Applications of Motif Discovery in Biological Data

by

Mark Philip-Walter Styczynski

B.S. Chemical Engineering
University of Notre Dame, 2002

Submitted to the Department of Chemical Engineering
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Chemical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2007

Author .....
Department of Chemical Engineering
May 29, 2007

Certified by .....
Gregory Stephanopoulos
Herbert H. Dow Professor of Chemical Engineering
Thesis Supervisor

Accepted by .....
William M. Deen
Professor of Chemical Engineering
Chairman, Committee for Graduate Students

Applications of Motif Discovery in Biological Data

by

Mark Philip-Walter Styczynski

Submitted to the Department of Chemical Engineering
on May 29, 2007, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Chemical Engineering

Abstract

Sequential motif discovery, the ability to identify conserved patterns in ordered datasets without *a priori* knowledge of exactly what those patterns will be, is a frequently encountered and difficult problem in computational biology and biochemical engineering. The most prevalent example of such a problem is finding conserved DNA sequences in the upstream regions of genes that are believed to be coregulated. Other examples are as diverse as identifying conserved secondary structure in proteins and interpreting time–series data. This thesis creates a unified, generic approach to addressing these (and other) problems in sequential motif discovery and demonstrates the utility of that approach on a number of applications.

A generic motif discovery algorithm was created for the purpose of finding conserved patterns in arbitrary data types. This approach and implementation, name Gemoda, decouples three key steps in the motif discovery process: comparison, clustering, and convolution. Since it decouples these steps, Gemoda is a modular algorithm; that is, any comparison metric can be used with any clustering algorithm and any convolution scheme. The comparison metric is a data–specific function that transforms the motif discovery problem into a solvable graph–theoretic problem that still adequately represents the important similarities in the data.

This thesis presents the development of Gemoda as well as applications of this approach in a number of different contexts. One application is an exhaustive solution of an abstraction of the transcription factor binding site discovery problem in DNA. A similar application is to the analysis of upstream regions of regulons in microbial DNA. Another application is the identification of protein sequence homologies in a set of related proteins in the presence of significant noise. A quite different application is the discovery of extended local secondary structure homology between a protein and a protein complex known to be in the same structural family. The final application is to the analysis of metabolomic datasets. The diversity of these sample applications, which range from the analysis of strings (like DNA and amino acid sequences) to real–valued data (like protein structures and metabolomic datasets) demonstrates that our generic approach is successful and useful for solving established and novel problems alike.

The last application, of analyzing metabolomic datasets, is of particular interest. Using Gemoda, an appropriate comparison function, and appropriate data handling, a novel and useful approach to the interpretation of metabolite profiling datasets obtained from gas chromatography coupled to mass spectrometry is developed. The use of a motif discovery approach allows for the expansion of the scope of metabolites that can be tracked and analyzed in an untargeted metabolite profiling (or metabolomic) experiment. This new approach, named SpectConnect, is presented herein along with examples that verify its efficacy and utility in some validation experiments. The beginning of a broader application of SpectConnect's potential is presented as well.

The success of SpectConnect, a novel application of Gemoda, validates the utility of a truly generic approach to motif discovery. By not getting bogged down in the specifics of a type of data and a problem unique to that type of data, a broader class of problems can be addressed that otherwise would have been extremely difficult to handle.

Thesis Supervisor: Gregory Stephanopoulos
Title: Herbert H. Dow Professor of Chemical Engineering

Acknowledgments

This thesis would not be complete, and these past five years would not have been survived, without the help and support of innumerable people. Always up for a challenge, though, I will attempt to enumerate at least a good chunk of them.

The first people that ought to be acknowledged are my collaborators. Kyle Jensen was my most important collaborator, and I will forever be greatly indebted to him in so many ways, both personally and professionally. His goals and ideas were both an inspiration and a jumping–off point for much of our work together, and this thesis reflects that. I also owe thanks to Joel Moxley, whose vision and goals helped steer SpectConnect to become what it is. Finally, I also am grateful for the help of both Lily Tong and Jason Walther — they both were collaborators on the SpectConnect work, but also helped me go pursue other projects that helped round out my approach to metabolomics.

I am also obviously indebted to my advisor, Gregory Stephanopoulos. Aside from financial support, he has also been a great help in navigating the obstacles of graduate school and figuring out how best to pursue my professional goals. He gave me lots of freedom in pursing projects and coming up with new ideas, which as we all know is both a powerful and dangerous thing; I thank him for this freedom and all I have learned from it.

Thanks are also due to other professors, including Joanne Kelleher and my thesis committee, comprising Dr. Rigoutsos and Profs. Lodish, Trout, and Bertsekas. Prof. Wittrup has been a particularly great help in the past year, giving great (and plentiful) advice. Profs. Brennecke and Stadtherr back at Notre Dame have also left their mark on me and continued to help me well into graduate school, and for this I will forever be indebted.

More indirect contibutors to this thesis include the Stephanopoulos group (both past and present), my office mates from other groups, and all of my classmates — including Jared Johnson, Amy Lewis, and Ginger Chao. I'm also grateful to my parents, Walter and Diane, and my sister Tracey, whose support has always meant

the world to me.

Finally, I absolutely must acknowledge all of my gratitude, love, and appreciation for my wondeful fiancee, Megan Cole. She put up with me, she guided me, and she inspired me. I'd hate to think where I'd be without her, and I'm perpetually excited about where I'll be next with her. I owe her everything and could never show it or say it enough.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

With the significant advances in high–throughput experimental techniques that have developed in biology within the past ten years, there has been rapid development in a relatively new field: bioinformatics. Broadly speaking, bioinformatics may be construed as any type of computational approach to biological systems. Some definitions may go so far as to include approaches that are more commonly associated with "computational systems biology", including differential equation–based modeling of systems and metabolic flux analysis. A more precise definition of bioinformatics might focus more on the data–driven aspects of the field. As such, the most obvious examples of bioinformatics are in sequence analysis, e.g. DNA and amino acid sequences. Other prominent examples of bioinformatics include the statistical analysis of DNA microarrays, gene network inference from microarray and other data, and literature mining for associations and correlations between different biological entities. One can note the consistent theme among these more "classical" examples of bioinformatics: a data–driven approach to analyze and interpret the biology underlying experimental results.

One interesting aspect of bioinformatics as a discipline is the continuing struggle between experimental and computational power as a driving force for further developments. As noted above, before the development of many high–throughput ex-

perimental techniques, there was not a strong drive for the development of algorithms and routines for biological data analysis. As this need arose, it first prompted *ad hoc*, heuristic approaches that were computationally feasible but not always completely accurate in their predictions. Of course, computing power is continually increasing, such that problems that could at first only be solved heuristically soon have a provably exhaustive and accurate solution at a reasonable computational cost. Algorithms must then adapt to reflect these possibilities. Frequently, though, even better experimental methods are being simultaneously developed, providing better, more detailed, or just a lot more data than previous high–throughput methods, thus preventing newly–developed accurate methods from being brought to bear on this newer, more complex dataset. This prompts the development of new computational analysis algorithms, and the cycle continues. Such seems to be the nature of bioinformatics in its infancy, and with the continuing development of both experimental techniques and computational power, it seems likely that this pattern may continue for some time.

Contextually, this thesis lies squarely in the middle of such a cycle. There are several well–established methods for some of the types of analyses pursued in this work. However, the potential for exhaustive, extremely powerful searches of certain problems has not yet been exploited. This thesis looks to harness this unexploited potential while unifying several diverse problems into one consistent, coherent, generic definition of motif discovery. The software produced as a result of this goal was ultimately used to perform novel, useful analysis of datasets that just a few years ago were not even being created. This then provides the potential to help bring an otherwise nascent field, metabolomics, closer to the level of acceptance of other "-omics" fields like proteomics and transcriptomics.

1.2 Thesis Objectives

The objectives of this thesis are the following:

1. To develop a motif discovery algorithm that is both as exhaustive and generic (or "data–agnostic") as possible. Exploiting recent advances in computational

capabilities, it is possible to develop a bioinformatic approach that can avoid heuristic shortcuts in many cases. This approach should not be specific to certain kinds of data, but rather should be generic enough that with minor modifications it can be applied to arbitrary problems with arbitrary data types.

2. To apply this approach to existing problems in bioinformatics. There are many outstanding problems in bioinformatics that have not yet been solved exhaustively or optimally. This approach should be capable of solving such problems such that end users (i.e., biologists) could use it to solve relevant problems in their research.

3. To apply this approach to novel problems in bioinformatics. While the many extant problems in bioinformatics are sufficient fodder for an enormous amount of work, this approach should ideally also be applied to problems that are not otherwise known about or prominently analyzed.

1.3 Thesis Organization

This document will address the different aims in turn. After a broad overview of background material (focusing on motif discovery and metabolomics) in Chapter 2, Chapter 3 will focus on the creation and development of a truly generic approach to motif discovery in sequential biological data. A few examples of its potential will be briefly addressed in this chapter as well. Chapter 4 will present a more in–depth example of how our generic approach can be applied to a relevant DNA sequence motif discovery problem. This chapter will also expand upon this sample problem to make it even more biologically relevant and faithful to the intricacies and deficiencies of the experimental data that would be used in the analysis. Chapter 5 moves into another example of how this approach can be applied, this time in the context of real–valued data obtained from metabolomic GC–MS analysis. Chapter 6 discusses the initial work towards a project looking to use these motif discovery approaches for the purpose of tracking as many metabolites as possible in numerous experimental

conditions. Chapter 7 presents some side projects that were done in the process of completing the primary thesis objectives. These projects focus on broader, higher–level aspects of amino acid sequence analysis. Chapter 8 presents some conclusions and provides suggestions for future work that can be done to improve the algorithms developed in this thesis as well as some work that can exploit the progress made by this thesis.

Chapter 2

Background

2.1 Motif discovery in sequential data

As has been widely noted, the availability of vast amounts of biological data has prompted great interest in the development of tools for the analysis and mining of this data.

For instance, the advent of advanced genome–sequencing techniques, from shotgun sequencing [112, 12, 168] to the most recent developments in approaches such as 454 sequencing [110], has been supplying a steady stream of whole–genome sequences available to the public. In fact, the technology has become so commonplace that genome sequencing sometimes becomes "just another step" in the pursuit of understanding certain organisms [151].

Protein structure determination is also continuing along, though perhaps at a slower pace. While standard methods for elucidating protein crystal structure (NMR and X–ray crystallography) have not advanced unusually rapidly in the past five years, there has still been significant growth in the number of protein structures that are available online via the Protein Data Bank (PDB) [25, 24]. With well over 42,000 structures available as of the time of publication of this thesis, there is a significant amount of data available in the PDB.

However, the mere presence of all this data is not nearly sufficient for the advancement of science. While a quick glance at a bacterial growth curve or a reactant

concentration curve can give an idea of what state an organism is in or what the kinetic constant of an enzyme is, the vast amounts of data acquired through "–omic" techniques require much more in–depth analysis in order to be useful. The most promising route to understanding this data is by identifying repeated elements that most likely have significant meaning. Just as a string of letters may seem meaningless to someone who has never seen the words that constitute a given language, a large portion of why our genomic and proteomic datasets are so confusing is because we just do not know what important repeated elements we should be looking for to understand the underlying structure of the data.

Merely knowing the sequence of hundreds of organisms, while in itself interesting, is not sufficient for understanding how genetic, metabolic, and cell–cycle regulation work in those organisms. While we may know very well where genes will start and stop, and we may have a very good idea of how promoter sequences work and how exons in genes will splice together, there are still many unknown aspects of genome sequences that remain to be elucidated. On a very fundamental level, even transcription factor binding sites (places where proteins called "transcription factors" bind to DNA and cause a cascade of events that results in the gene being transcribed) have insufficient understanding for engineering applications. Certainly, we understand how these binding sites work and that certain residues have more plasticity than others in allowing for binding; however, we do not yet have the capability to identify binding sites *a priori* in a sea of genomic data, nor do we have the ability to rationally engineer these binding sites to suit the whims of metabolic engineers and synthetic biologists. We also have relatively little understanding of how complex sequence elements found in higher eukaryotes, like enhancers and insulators, work to effect transcriptional regulation. It is the understanding of all of these elements that will bring us much closer to an understanding of genetic data.

Similarly, knowing only the structures of proteins is not nearly sufficient for the scientific and engineering purposes towards which we currently strive. Being able to qualitatively describe certain substructures as alpha helices or beta sheets is helpful for classifying proteins into families, but has relatively little utility outside of that

application. Ideally, we would like to be able to correlate overall structure or specific substructures to protein functions; we would also like to be able to predict protein structure strictly from the amino acid sequence of the protein. While there will certainly be the opportunity to employ thermodynamic and computational chemistry methods to advance these goals, the discovery and tracking of conserved substructures within amino acid sequences and protein structures will be a key step in unraveling the mysteries of protein folding and function.

Thus, the discovery of repeated patterns in sequential data is an extremely important problem in biology, biological engineering, and biochemical engineering. The potential applications of such knowledge are innumerable and include a variety of industrially useful applications like the engineering of microbes for more effective biosynthesis of valuable chemical feedstocks.

Before moving on, it is also worthwhile to clarify some terminology used in this document. The words "motif" and "pattern" can be used more or less interchangeably; they both refer to a repeated element in data that is believed to have some correlative or functional role. This thesis will frequently refer to its goals in the context of "motif discovery"; I use this in place of an alternative phraseology, that of "pattern recognition", to underscore one key aspect of this work. Generally speaking, "pattern recognition" can be construed as referring to any sort of classification problem. There are many algorithms and tools that have been developed to identify occurrences of known motifs based on the motif representations that will be discussed below, which some may refer to narrowly as "pattern recognition". Depending upon the motif model, this is often a difficult problem in and of itself. However, the "motif discovery" problems addressed here go one step beyond those problems: in motif discovery, one does not know what the motif is while trying to find it in the data. The only assumption made is that the set of sequences that one is searching are somehow related and thus likely to have some substructure or subsequence that characterizes the data. Using only the data, one attempts to enumerate the important repeating elements that give the data some sort of underlying structure. Thus, motif discovery is much more difficult than what some may narrowly refer to as pattern recognition;

one could say that specific kind of pattern recognition is to motif discovery what using the "Find" function in a word processor is to deducing the words that make up a foreign language based only on the letters in that document.

2.1.1 Types of data

Ordered vs. unordered data

In its most general sense, motif discovery can be performed on any arbitrary type of data. A common and intuitive type of data to be mined is "unordered" data, where the data points have no sequential nature. A mathematical example of unordered data would be taking a string of numbers from a pseudo–random number generator and trying to deduce from those numbers the underlying prior distribution of the generator. In this case, though the individual values drawn will be of use in deducing the prior distribution, the order in which they are drawn has no use in the inference. Some examples of unordered data are illustrated in Figure 2-1.

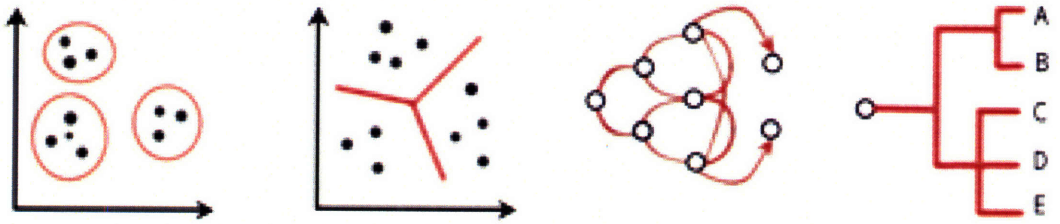


Figure 2-1: Some examples of unordered data analysis. Unordered data can be cast into arbitrary coordinate systems; in certain coordinate systems, some motifs may become obvious. The first schematic demonstrates how one might use a clustering algorithm to break the data into smaller groups. The second schematic shows how one might define a linear discrimination approach to predict the classes or clusters of future data points. Unordered data points can also be cast as either undirected or directed graphs (as in the third schematic) or displayed clustered together in a hierarchical manner (as in the fourth schematic). The unifying theme here is that while there may be motifs somewhere in the data, they are not necessarily likely to occur in the data's native coordinate system — it is in casting the data into an alternate coordinate system that one is most often successful at unordered data classification or motif discovery.

A more real–world example of unordered data can be found in online commerce: the "market basket" problem. A customer of an online vendor of books that purchases two novels is frequently presented by that online vendor with suggestions for other books that might be of interest. This is actually a motif discovery problem. The vendor takes the two books being purchased by the customer and looks in all of its previous transactions to identify other customers who purchased one or both of the books that the current customer is purchasing. Based on those previous transactions, the vendor attempts to guess the most likely other books that the current customer would consider purchasing. The vendor is thus finding "motifs" that help to describe purchasing habits and hopes that these motifs may give the customer ideas about other books to purchase. Of course, it is extremely difficult to search all prior transactions effectively and efficiently. As such, heuristic shortcuts are often necessary to make the problem computationally tractable. These heuristic shortcuts, while computationally useful, frequently cause the suggestions to be less helpful than they otherwise might be, leading to the occasional odd suggestions that one might encounter from online vendors.

This last aspect of the market basket problem speaks to a larger issue in motif discovery in unordered datasets: computational tractability. Since there is no sequential nature to the data, there is no reason to expect that data points that are "near" each other (whether in sampling order or some other natural coordinate system) are part of a motif. With such delocalized similarities, the search space grows combinatorially with the size of the dataset. For large datasets, this is obviously intractable to search exhaustively, necessitating (as in the market basket example) heuristic shortcuts that sacrifice accuracy in making the problem feasible.

Ordered datasets, on the other hand, lend themselves much more easily to exhaustive searches. In these datasets, the data points have a sequential nature such that the order in which they occur is relevant. By virtue of this fact, one expects to find more localized similarities within the data; that is, one expects that important motifs are likely to span a small region of the sequence rather than draw uniformly from the sequence. While this approximation is not valid in all cases (one of the

most prominent being RNA sequences where rather distal portions of the sequence physically touch to form secondary structure), it is useful for quite a few relevant biological problems that will be discussed shortly.

Since similarities are more localized, search algorithms can restrict their search to motifs that are composed of approximately continuous regions on the sequence. This restriction then allows for a drastic reduction in the size of the search space; what would otherwise be a search space that is combinatorially growing in the size of the dataset is instead much closer to linear growth in the size of the dataset. For example, if one were to search an unordered dataset consisting of ten items for all possible motifs that were composed of five of those items, there would be 252 possible motifs to analyze. In contrast, an ordered dataset of the same size where only consecutive runs of five items were interesting would have exactly 6 possible motifs. This reduction is almost two orders of magnitude; in larger datasets, the reduction would be amplified even further. It is then clear that ordered datasets are much more easy to search exhaustively than are unordered datasets. However, that is not to say that ordered datasets are always easy to search exhaustively; as the size of these sequences climbs into the hundreds, thousands, and higher, they quickly become difficult (and sometimes intractable) to search thoroughly. The ordered, or sequential, motif discovery problem thus represents a difficult, but reasonable, problem to address.

String– vs. real–valued data

Another distinction worth noting is that between string–valued and real–valued data. Each letter in a DNA sequence can only take one of four values for each of the four possible bases: A for adenine, G for guanine, C for cytosine, and T for thymine. Each letter in a protein sequence can take one of twenty values for each of the twenty common amino acids (there are some uncommon amino acids, but we will ignore those for the purposes of this discussion). In this sense, DNA and protein sequences can be described as a string (using the computer science sense of the term) because each letter is chosen from a predetermined, finite set of possibilities. There exist

many types of sequential data, though, that are not restricted to a finite subset of possibilities: for instance, a series of integers could have any of the infinite set of integers at any given data point. The same can be said for decimal–valued numbers, as well. These two types of datasets are illustrated in Figure 2-2. In this work, I will refer to the set of sequences composed from an infinite set of possibilities at each data point as a "real–valued" dataset.

The importance of this distinction lies in the feasible approaches that one can use to uncover motifs in the data. Given a finite number of choices for each letter in DNA or protein sequences, one can exhaustively enumerate all of the similarities between the physical entities that each letter represents and come up with a similarity metric based on those properties that best define how similar one letter is to another. For real–valued data, the creation of such a simple, finite comparison metric is not nearly as straightforward. In addition to this fact, and perhaps as a consequence of it, there are a large number of sequence analysis tools (that will be discussed below) geared towards string–valued data and relatively few tools geared towards real–valued data. Given this circumstance, it is not an uncommon or unreasonable approach to attempt to recast the real–valued data problem as a string–valued problem so that existing string–valued sequence analysis tools can be brought to bear on the real–valued data. However, in performing this transformation — moving from an infinite set of possible values for each data point to a finite set of possible values — one necessarily loses quite a bit of information. While in some cases this approximation may be successful, it is certainly not ideal, as will be discussed later in this chapter.

2.1.2 Motif representations

As noted above, the working definition of a "motif" for the purposes of this thesis is a repeated element believed to have some correlative or functional role. For a motif to be functional, though, it does not need to be identical in all of its occurrences. One obvious example of this plasticity in motifs is in stock trends. If one were to find that multiple stock prices followed very similar, but not identical, prices over the course of a year, one could easily (by eye) identify a motif that describes their

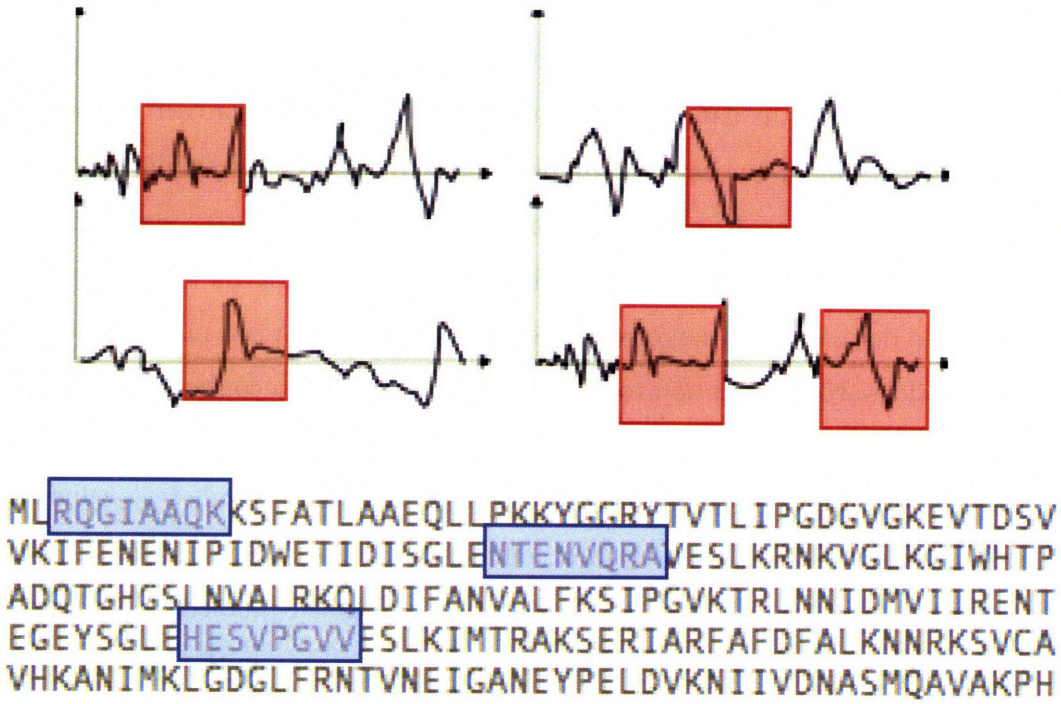


Figure 2-2: Two different "classes" of ordered data. The ordered data on the top is real–valued; that is, any given point in the sequence can have essentially any of an infinite number of possible values. The ordered data on the bottom is string–valued; that is, any given point in the sequence can only take one of a finite number of possible values (in this case, letters representing the different natural amino acids). While there are obvious (even visual) differences between the two types of data, there is still an underlying similarity, hinted at by the same treatment of arbitrary shaded boxes searching for motifs.

behavior. It is not necessary for us to see that their prices were exactly the same over the course of the year; we understand that there is noise inherent to stock prices and the overall financial system that causes slight deviation, so we can see through this noise to identify the underlying motif. In much the same way, noise is tolerated within biological systems such that a sequence of nucleotides or amino acids has some degree of plasticity; that is, a certain amount of change from the "real" motif does not prevent it from having its appropriate function and being identified as an instance of the motif.

Since there is some noise and uncertainty inherent in motifs, it is desirable to attempt to capture this uncertainty in a comprehensible representation of the motif. Different motif representations have different levels of approximations and thus varying levels of precision in their representations. A few representations will be explored briefly below, as these terms will be used frequently throughout this thesis.

Consensus sequence

The simplest way to represent a set of patterns is that of the consensus sequence. Constructing an arbitrary consensus sequence is a rather straightforward endeavor. Figure 2-3 presents an example from the literature that illustrates this point [134]. The six sequences on the top represent promoters that behave similarly. No single pattern of non–wildcard characters can be used to represent all six sequences. A consensus sequence is an attempt to represent all six sequences with only one sequence. The first candidate consensus sequence is `TATAAT`. When one uses the term "consensus sequence", this is the type of representation that is most commonly implied: where each letter in the sequence represents the most dominant possibility amongst all motif instances. However, if one were to search the genome (in this case, that of *E. coli*) allowing no mismatches with the consensus sequence, one would only find two of these sites. If one mismatch were allowed, three of the sites would be identified. Only if two mismatches were allowed would all six sites be identified. However, in that case one would also find a "match" about every 30 base pairs of random genomic sequence. Thus, many non–promoter regions would be identified as promoters. Even though the

sensitivity (ability to find known true positives) of such a sequence would be 100% if two mismatches were allowed, the specificity (ability to avoid false positives) would be quite low, making that a very poor representation of the site. Nonetheless, for stronger motifs this kind of consensus sequence is still commonly used.

An alternative consensus sequence using wildcard characters is also presented: `TATRNT`, where `R` stands for `G` or `A` and `N` stands for `A`, `C`, `G`, or `T`. Such a sequence with no mismatches identifies four of the sites with a reasonable specificity. Allowing one mismatch identifies all six sites, but again at a loss of specificity. Thus, one can clearly see that while creating an arbitrary consensus sequence is simple, finding the sequence and cutoff value that is optimal for identifying the existence of new (undiscovered) sites is very difficult [156]. Other literature sources [38] have already compared quite a few methods for constructing sequences and cutoff values to reach that optimality.

Regular expressions

Regular expressions — also known as regular grammars or type–I grammars — are simple rules specifying allowed arrangements of characters within a sequence [37, 88]. The most important aspect of regular expressions is that they permit wildcards, sets, and repeats which allow for (respectively) any character to match at a position, a set of characters to match at a position, or multiple instances of a character (or set) to match at a position.

Regular expressions can be seen as a more complicated version of consensus sequences. The possible wildcard values in the consensus sequence above are an important aspect of regular expressions. Instead of using a new letter to denote a subset of letters, though, regular expressions use brackets to delineate which letters may possibly exist at a given location. So, using the consensus sequence example above, the regular expression representation of the consensus sequence `TATRNT` would be `TAT[AG][ACGT]T`. Alternatively, since `[ACGT]` really means that any possible letter (from the predefined finite set) is acceptable, one can use the wildcard character "`.`" to represent complete indifference to what is present at that position. One could then use `TAT[AG].T` to represent the above regular expression. Finally, repeats can

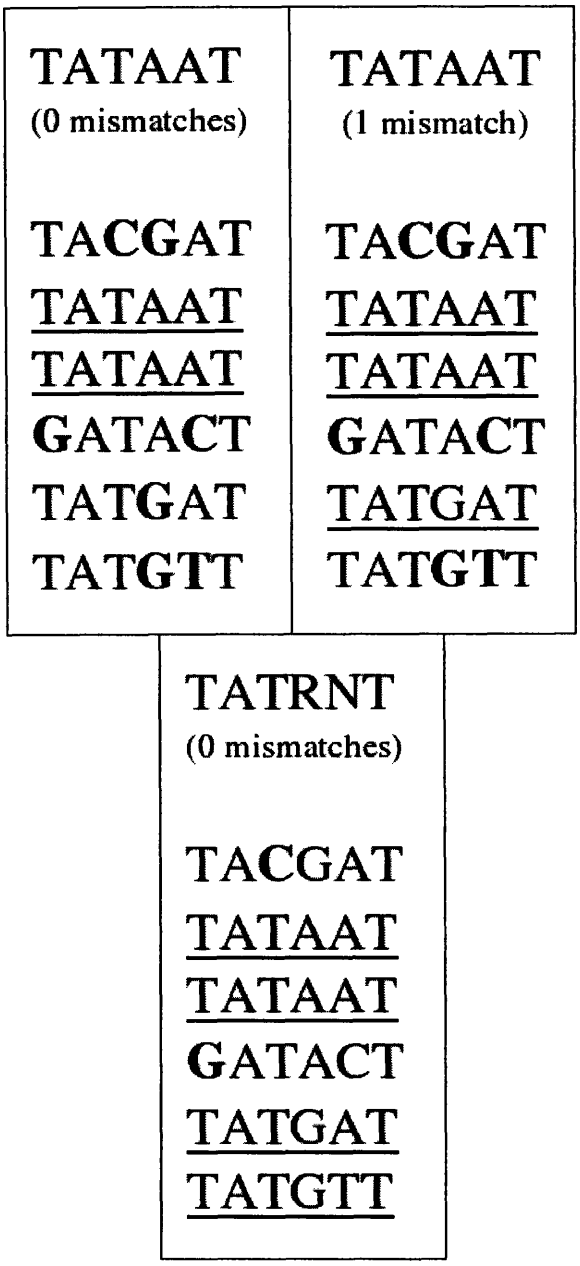


Figure 2-3: Six promoter regions [134] can be represented with either of these two consensus sequences. In the latter sequence, `R` means {`A` or `G`}, while `N` means {any nucleotide}. Allowing more mismatches with a consensus sequence lets more target sequences be found but ultimately causes a loss of statistical significance, as many more non–target sequences will also be found.

be represented in regular expressions. If the letter T can occur anywhere from four to nine times in a sequence, it would be represented as T$\{4,9\}$. If it could occur some undetermined number of times, from zero to infinity, it would be represented as T*.

By using these operators, regular expressions are more powerful than consensus sequences. Regular expressions can describe more complex motifs whose size and internal repetitiveness may vary significantly, which is not possible with consensus sequences. Also, regular expressions can more easily describe the multiple possible elements that can exist at any given location in a motif. However, it is worth noting (as will be discussed below) that many motif discovery approaches that use regular expressions will only consider a subset of these operators, thus limiting their capabilities in describing motifs.

Position weight matrices

Though consensus sequences and regular expressions have their merits in representing motifs (primarily their simplicity), they have significant shortcomings. Their biggest flaw is in the assumption that the matching of each sequence position contributes equally to the validity of the motif instance. That is, both of these methods assume that there is not a continuous distribution of "preferences" for letters at any given position. The consensus sequence acknowledges the fact that one letter may be preferred at a given position for a subsequence to be a motif instance. Regular expressions acknowledge that multiple different letters may be viable at any given position. Neither approach, though, addresses the fact that while there may be a first preference for a letter at a given position in a motif, there may be a quantifiable second, third, or higher preference at the same position. Biologically, this makes some sense; I will use DNA–binding proteins as an illustrative example. If a protein binds to a specific DNA sequence, there may be some leeway in what the binding site sequence may be. The binding capability is defined by the kinetics and thermodynamics of the protein–DNA interaction. Changing one specific base from, say, A to T may change the binding kinetics less than changing it from A to C. This in turn means that the binding site is more likely to have A in that position than any other nucleotide, but it

is also more likely to have T than C. These differences can then be quantified, based on (ideally) the thermodynamics of the system or (realistically) the frequency with which one sees each letter at each position in the binding site.

Position weight matrices (PWMs) help to express the unequal contribution of different sequence positions in the site to the binding energy of the transcription factor. For each location in a potential binding site, there is a corresponding column in the PWM. There are four rows in the PWM corresponding to the four bases (or twenty rows if one is dealing with proteins). In order to find the PWM score for a given sequence, one selects the element in each column of the matrix corresponding to the base pair at the respective position. These elements are then summed for the final score. The site is deemed a binding site if it is below (or above, depending upon the sign convention used) a threshold value usually determined such that the results meet some level of statistical significance. An example is given in Figure 2-4, where the PWM score for the sequence is found by adding the circled matrix elements.

Defining the PWM weights is a relatively simple task, though it requires a substantial amount of experimental data. PWMs are currently available in multiple databases, most notably TRANSFAC [178]. One can also create a new PWM based on new data or subsets of data. One tool to do so is a Perceptron, which is a simple neural network [157]. The method commonly used today is similar to that presented by Staden [153], which is to assign a weight equal to the negative logarithm of the frequency of the base at each position in a set of binding sites.

Other representations

Other representations have also been explored, though they are much less common than the previously delineated representations. One such approach is a natural extension of PWMs to Markov chains rearranged so that statistically interdependent positions are placed next to each other in the chain [51]. Hidden Markov Models (HMMs) are also another useful type of motif representation; prominent examples of such an approach include models of transmembrane helices [99] and exon/intron splice signals [35]. Neural networks and Bayesian networks can also be used to represent

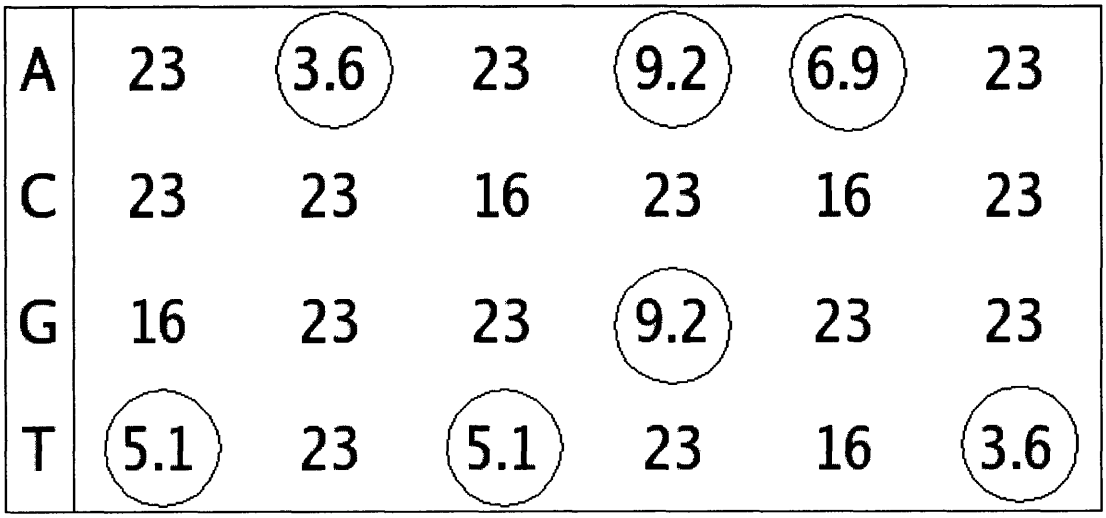


Figure 2-4: A position weight matrix (PWM) based solely on the six promoter sequences in Figure 2-3. This PWM uses a simple method of weight determination where the weight is equal to the negative logarithm of the frequency of a base's appearance at a given position (plus pseudocounts), times ten. Pseudocounts are additional "occurrences" added to the frequency of each letter so that no letter's frequency is zero. This is done because although a letter may be extremely rare in a certain position in the motif, it is likely that there may be some promoter sequence with that letter in that position and that we just haven't "sampled" enough promoter sequences to have seen it happened. It thus prevents an "improbable" event from being deemed as "impossible" merely due to a finite training sample. In the limit of extremely large training samples, the change in weights caused by these pseudocounts is negligible. The circled bases represent (potential) consensus sequences.

motifs. The main difficulty in using most of these representations is in the relatively large body of experimental evidence and motif examples that must be used in order to create accurate representations of motifs using these highly descriptive models. In the above HMM examples, a large amount of data is likely not a limiting factor, but for transcription factor binding sites one does not expect to have hundreds to thousands of example motif instances that will help train these models appropriately.

2.1.3 Standard tools for sequence analysis

Given the maturity of sequence analysis and the relative sophistication of our understanding of different sequence elements in both protein and DNA data, it is not surprising that there exist a few commonly–used and well–developed tools for analysis of string data. These tools range from workhorses for identifying homologies between a query string and a large database to programs that can be used by any biologist to attempt to find recurring motifs in DNA and protein sequences. While each tool has a significant contribution to the discipline, there still remains a significant gap in capabilities that motivates the research in this thesis. A few tools will be described herein, followed by an analysis of some of the shortcomings common to most or all of those tools.

Alignments and homology detection

This class of sequence analysis tools focuses on finding similarities between sequences. One approach is to find a "global" alignment between two sequences — that is, an alignment that spans the entire length of both sequences (with some blank spaces, or "gaps", inserted) and provides the best overall similarity. This approach is most effective for sequences of very similar length and that are not very evolutionarily diverged. For more diverged and diverse sequences, one frequently uses "local" alignment methods. These local methods focus on the small regions of similarity that may exist within larger stretches of dissimilarity and evolutionary divergence. Frequently, these short regions of homology can provide significant insight and characterization

of the sequences under study. Local alignments can then be used to search databases for homology.

In multiple sequence alignment, one seeks to align a set of sequences together in such a way that homologous patterns occur at the same place. This is done (as in pairwise alignments) by changing the start position of a sequence with respect to another and (perhaps) allowing for insertions of gaps in some sequences to facilitate alignment. A number of tools are available for multiple sequence alignment, including ClustalW [163] and T–COFFEE [124]. Various other tools usually used for other problems can also be used for multiple sequence alignment, including `TEIRESIAS` [140], MEME [17], and others.

Below I will go into greater detail about some of the more prominent algorithms and tools for alignment and homology detection within the field of bioinformatics. Most, if not all, of these tools were used over the course of this thesis, whether for benchmarking studies or because they were the most appropriate tools for the task at hand.

Smith–Waterman alignment The Smith–Waterman [152] algorithm is an approach to local sequence alignment. It is based on the concept of "dynamic programming", an inductive algorithm that aims to find optimal solutions by solving the problem as several smaller, stepwise problems. Implementations frequently entail the creation of a matrix that allows the tracking of all possible solutions of a problem. Within the context of pairwise sequence alignment, the matrix has a row for each letter in one sequence and a column for each letter in the other sequence. The two sequences will be aligned by inserting short gaps to help similar parts of the two sequences coincide. Each gap that is inserted in a sequence has a specific scoring penalty associated with it. Each "alignment" of two letters has a score associated with it as well; this is defined by a "substitution matrix", a set of values that reflect the relative *a priori* likelihood that a certain letter would appear in one sequence given the corresponding letter in the other sequence. (For proteins, these substitution matrices indicate the likelihood that one amino acid residue will change

to another over the course of evolution.)

The Smith–Waterman algorithm is based on the Needleman–Wunsch algorithm [119] for global alignment, which will now be described. To align the sequences, one moves stepwise through them in a "greedy algorithm" approach. Based on the gap penalties, it is clear what the score would be for an initial gap compared to a letter in either sequence, as well as for multiple initial gaps in one sequence compared to the other. It is also known what the score would be for a direct alignment of the first letter of each sequence (by the scoring, or substitution, matrices discussed above). Using this knowledge, the algorithm then attempts to compute the best possible score for the alignment of the first two letters in the first sequence with the first letter in the second sequence. This alignment can come from either an initial gap in the second sequence followed by alignment of two letters, two initial gaps in the second sequence followed by a gap in the first sequence, or the alignment of both sequence's first letters followed by a gap in the second sequence. Since the scores from all of those first–step scenarios are known, it is possible to deduce which would give the best possible alignment score at the current step, and that score is recorded as the appropriate matrix entry. Next, the same computations are done for the alignment of the first three letters of the first sequence with the first letter of the second sequence. These computations are based on the three possible prior (shorter) alignments that could possibly lead to that alignment. This process is then repeated for each possible sub–alignment (based on the three prior sub–alignment scores), until the entire matrix of values is filled out.

The major difference between the Needleman–Wunsch algorithm just described and the Smith–Waterman algorithm is that if at any point during the Smith–Waterman algorithm the penalty or score of a certain step would push the cumulative score below 0, then that score is just defined as 0. The optimal local alignment is then selected as the highest–scoring entry in the matrix, and the alignment can be recovered by tracing backward through the matrix using the known gap penalties and substitution/scoring matrix.

While this approach provides a guaranteed optimal local alignment, it does so at great computational expense. The computational complexity of a straight–forward

implementation of this algorithm is $O(mn)$, where m is the length of the first sequence and n is the length of the second sequence. The space requirements are also $O(mn)$. While these costs are trivial for the pairwise comparison of most gene or protein sequences, the approach does not scale well to straight–forward database searches since they require $O(p)$ pairwise comparisons, where p is the number of sequences in the database. For searches in large databases, a much more efficient approach is required for search time and space to be tractable.

BLAST BLAST [10], which stands for "basic local alignment search tool", was developed as a heuristic approach to finding homologies between two strings. It is a computationally tractable alternative to more exhaustive approaches like Smith–Waterman and dynamic programming algorithms for the purposes of searching large databases for local homologies.

Briefly stated, BLAST uses small clusters of similarity identified by comparisons to a pre–hashed database to identify the potential beginnings of homologies. That is, for a specific data type (whether DNA or protein sequences), a minimal "word" size is defined. For proteins, this size is frequently 3, while for DNA sequences, this size is often 11. For the database of sequences that is to be searched, all possible words are enumerated and stored. Each possible word in the query sequence (supplied by the user) is then compared to all of the words that exist in the database. Using some appropriate scoring metric or "substitution matrix", any word in the data that is sufficiently similar to a word in the database is noted and called a high–scoring pair, or HSP. These HSPs are then further analyzed and extended to discover longer regions of homology that are of greater biological relevance.

The central idea behind this approach is that if there is some long stretch of homology between the user's query and some sequence in the database, there is likely to exist in that stretch some extremely high–scoring pair of words. Frequently, there will exist more than one such pair. If a set of such pairs, or at least one pair, can be identified, then it can be extended to enumerate most (or all) of the homology between the query sequence and the database sequence.

Owing to the nature of its approach (which essentially amounts to multiple hash lookups in a stored database), BLAST is an extremely efficient algorithm. It is still used today with relatively few substantive modifications from its initial core searching algorithm. It can search multiple genomes for homologies to one's query within seconds. The only limitation that prevents it from being used natively on home computers is the physical memory required for storing the entire hash lookup table. This problem has long been obviated by the availability of central servers for performing BLAST searches on entire multi–genome databases.

BLAST has been further developed and optimized for many specific applications. Direct comparisons of two sequences (though truthfully unnecessary since Smith–Waterman searches can easily handle such alignments) are readily available on BLAST servers. Optimal parameters have been developed for searches looking for small, extremely well–conserved stretches of sequence. An iterative version of BLAST, known as PSI–BLAST (position specific iterative BLAST) [11], has been developed for the identification and extension of protein families using position–weight matrices. All of these versions and extensions of BLAST find frequent use in both the literature and the laboratory, making it the pinnacle example of a successful, well–integrated bioinformatics tool.

Motif discovery tools

This class of tools focuses on identifying conserved subsequences, or patterns, within sets of sequences. In a sense, these are more directly related to multiple sequence alignment tools than to pairwise sequence alignment or homology searching tools because they look to characterize many sequences simultaneously. The key difference between these tools and multiple sequence alignment tools are the noise and evolutionary relationship in the data being analyzed. The most common application for multiple sequence alignment tools is to better identify the similarities and differences between sequences (say, proteins) that are believed to be evolutionarily related. In the case of proteins that are rather closely related, multiple sequence alignment aims to insert small gaps to make the obviously similar parts of each sequence line up. For

more distantly related proteins, multiple sequence alignment helps to identify larger gaps between related portions of the proteins under study. The tools discussed in this section, however, are not necessarily expected to be closely related. The problem is not to find the best placement of gaps to make the sequences line up well with each other; rather, it is almost the inverse, where one needs to identify only those small regions within the sequences that actually are similar while disregarding the vast regions of dissimilarity in the rest of the sequences. In a sense, these tools are looking for multiple–sequence local homologies — essentially the most difficult respective aspects of multiple sequence alignment and homology detection.

The problems these tools look to solve are very difficult. As such, the approaches that these tools use are frequently very complex — both computationally and conceptually. The results, though, can frequently convey much more insight into the dataset than a basic alignment or homology search can. Below I will review the basic underpinnings of some of the motif discovery tools that are most relevant to this research or most prominent in the field.

`TEIRESIAS` `TEIRESIAS` is a tool created by Rigoutsos and Floratos [140] to exhaustively search the entire space of sequences for patterns meeting certain user–specified criteria. Our research group has used `TEIRESIAS` for a variety of bioinformatic purposes including peptide design, binding site discovery, and substitution matrix construction. A cursory overview of the tool is given below; a more thorough and rigorous explanation can be found in the literature [140]. In general, `TEIRESIAS` works on any character sequences, whether amino acids, nucleotides, or discrete numbers. Given a set of character sequences and a set of integer parameters L, W, and K, `TEIRESIAS` will find all patterns having at least L non–wildcard characters over a span of any W characters and occurring at least K times. In essence, L/W reflects a minimum "density" of non–wildcard characters in the pattern, while L reflects the minimum pattern length. Any pattern longer than L must meet the L/W criterion for any given window of W consecutive characters. `TEIRESIAS` returns a list of patterns, the number of occurrences of those patterns, the number of different sequences in

which the patterns occur, and (optionally) "offset lists" indicating the location of each occurrence of each pattern.

The noteworthy aspect of TEIRESIAS is the way by which it obtains these results. Rather than enumerating and searching for every single possible pattern, it only initially enumerates "elementary" patterns; that is, only those patterns satisfying the L/W constraint and with exactly L non–wildcard characters are enumerated. Longer patterns are then created by "convolution", which allows for elementary patterns to be combined provided that there is sufficient support and overlap between them. This strategy eliminates a great deal of computation time when compared to other brute force methods. Ultimately, the method leads to the following three properties:

1. All maximal patterns are reported.

2. Only the maximal patterns are reported.

3. Running time is quasi–linearly dependent upon the output (the number of patterns present in the data).

The impact of the first and second properties is that a complete, non–redundant set of patterns is returned for a given set of parameters. Such a pattern–space search is thus exhaustive, and one can be assured that given the parameters supplied to TEIRESIAS, all patterns have been discovered. The impact of the third property is that patterns of any length can be discovered. Since computation time is only output–dependent, no restrictions are placed on the maximum pattern length as is done in some other routines to make problems tractable.

Of course, the algorithm does have distinct disadvantages which must be worked around or tolerated. For instance, some *a priori* user knowledge is required in the form of input parameters. If the general character of the pattern being sought is known, this requirement for *a priori* knowledge may be tolerable; in general, though, this creates some degree of bias in the results. Such bias can be minimized by choosing parameters conservatively or searching through pattern space with multiple sets of parameters. Also, since computation time is output–based, it increases greatly when

simple pattern repeats (e.g., poly–A blocks) are present. A problem that is otherwise tractable can quickly become intractable if numerous simple repeat sequences must be analyzed for patterns. This issue can be circumvented by using a filtering program that "masks" (eliminates) regions of low complexity or information content. These steps are rather *ad hoc*, though, leading to a potential loss of generality in the results.

Despite some disadvantages, `TEIRESIAS` provides a novel, feasible way to exhaustively discover patterns of assigned density, length, and support across an arbitrarily large set of sequences.

Expectation maximization Several works are available that discuss expectation maximization in different contexts and with different implementation specifics, including a stochastic dictionary model of Gupta and Liu [62], the expectation maximization algorithm used in MEME by Bailey and Elkan [17], and others [105] . The explanation contained herein will focus mainly on the guiding principles behind these works. As already stated, the goal is to find an arbitrary motif contained across a set of sequences. To perform expectation maximization for motif discovery, one begins with an initial guess of PWM weights, either from preexisting physical knowledge or from an arbitrary guess. Each sequence is then searched to find the motif starting position that maximizes the score given the current weights. Given those new starting positions, new weights are calculated such that they create the maximum likelihood of producing the patterns represented by the current binding site starting locations. Given these new PWM weights, the process repeats itself by changing the site starting positions so as to maximize the PWM score again. The process iterates until convergence, thus simultaneously yielding both the binding site locations and the PWM weights corresponding to the maximum likelihood of producing the binding sites found by the algorithm. Within the step of obtaining maximum likelihood parameters, manipulations involving Bayesian estimation and associated priors may be utilized.

Efforts at finding cooperative binding sites, where two proteins might bind and interact to have an unusually strong regulatory effect, are also a novel application of

the expectation maximization method [61]. In this case, the goal is to identify two weak patterns below some fixed cutoff distance from each other as one strong pattern.

Each of the aforementioned implementations of expectation maximization is a vast improvement over needing extensive pre–existing knowledge about the binding site being sought. Obviously, these methods are much more computationally intensive than simpler string–based methods; there are also a few other disadvantages from which they suffer. For instance, the results are initialization–dependent, demanding multiple initializations in order to gain confidence that one has obtained the correct site pattern and locations. These methods are also gradient–based, so they are highly susceptible to local minima [114]. This can be overcome with some difficulty by implementing other search methods within the algorithm (e.g., simulated annealing). Finally, one loses quite a bit of information about the data in these methods. One cannot, for example, provide confidence intervals along with the results. It then seems that it may be better to attempt to work with the entire probability distribution associated with the binding sites rather than just a point estimate as one uses in expectation maximization.

Gibbs sampling Gibbs motif sampling has been widely used to discover binding sites for transcription factors [1, 175]. Tools implementing this strategy range from AlignACE [79] to BioProspector [104] with only slight variations. The method is derived from a more general Gibbs sampler philosophy that was altered to work for motif discovery [100, 103]. The most important aspect of Gibbs sampling is that it works with the entire probability distribution, not just a point estimate, by describing a complex probability distribution in terms of a Markov chain built with the simpler marginals of the distribution [114]. Thus, it gives more accurate results at the cost of increased computational complexity.

The general Gibbs sampler philosophy is described in the following example in three variables [114]. We suppose that these three variables are described by the probability distribution $P(x_1, x_2, x_3)$. Gibbs sampling consists of sampling $x_1(i+1)$ according to $P(x_1|x_2(i), x_3(i))$, then sampling $x_2(i+1)$ according to $P(x_2|x_1(i+$

$1), x_3(i))$, and finally sampling $x_3(i+1)$ according to $P(x_3|x_1(i+1), x_2(i+1))$. This process is then repeated indefinitely until convergence. At convergence, this Markov chain becomes the joint distribution, denoted by the chain operator:

$$P(x_1, x_2, x_3) = P(x_1|x_2, x_3) \circ P(x_2|x_1, x_3) \circ P(x_3|x_1, x_2) \tag{2.1}$$

These variables can then be grouped or collapsed so as to allow faster convergence. Liu proved [102] that for the case of the gene regulatory problem, which meets the conditions of reasonably fast Markov chain convergence and simplicity of drawing from the conditioned components, a collapsed Gibbs sampler

$$P(x_1, x_2, x_3) = P(x_3|x_1, x_2)\,[P(x_1|x_2) \circ P(x_2|x_1)] \tag{2.2}$$

converges faster than either the grouped or basic Gibbs sampler. This is then directly applicable to the motif discovery problem, as x_1 represents the motif probability matrix and background model, and x_2, x_3, and many subsequent variables would represent the alignment positions of the sought patterns, all found conditional upon the sequences initially given. Again, much Bayesian estimation and manipulation of priors is contained within the steps of the algorithm; for more details, consult the papers by Lawrence et al. [100] and Liu et al. [103].

(The tools mentioned at the beginning of this section only alter the collapsed Gibbs sampling strategy slightly. AlignACE uses a fixed single nucleotide background model and retrieves only one motif at a time, while BioProspector uses a higher order Markov model.)

The overall process can be visualized in a way very similar to the explanation used for expectation maximization described above. One key difference is that in Gibbs sampling, the likelihood of choosing a specific site to be the motif location in a specific sequence is merely a probabilistic function of the score of that site given the current PWM model. Thus, the Gibbs sampler is non–deterministic in terms of the results that it returns. This is particularly relevant for the initial stages of MEME/Gibbs sampling, when the motif is not well–characterized. Given the same initialization

guess, MEME will consistently identify the same motif locations in each sequence for the next step in the process, while the Gibbs sampling algorithm will choose motif locations for the next step randomly. If the first guess is relatively uninformed, one expects little variation in the scores of the potential motif sites, and thus the Gibbs sampler will choose sites with an approximately uniform random distribution. If these next sites are truly random, one expects a similarly uninformed PWM to result, and this process will continue until a few highly related motif sites are selected. This increase in similarity will help to drive the sampler to convergence. This converged answer will also hopefully be more robust with respect to initial guesses.

Phylogenetic footprinting The actual implementation of phylogenetic footprinting is rather simple: given a number of sequences, use multiple sequence alignment with existing tools to find unusually conserved regions. Algorithms may be based on either local alignments (as in BLAST [11] and PIPMaker [148]) or global alignments (as in AVID [29] and VISTA [45, 111]); both methods have proven effective in detecting unusually conserved regions. The Bayes Block Aligner, which focuses on aligning highly conserved and ungapped blocks, has also proven useful in phylogenetic footprinting [175].

The key behind phylogenetic footprinting is its underlying evolutionary hypotheses. Orthologous regions are DNA sequences from different species that arose from a common ancestral gene during speciation and are likely to be involved in similar biological function [106]. One seeks to identify the best–conserved motifs in orthologous regulatory regions from multiple species with the belief that evolutionary selective pressure causes functional elements to evolve at a slower rate than that of nonfunctional sequences. This then gives a distinct advantage over the multiple sequence alignment strategy using sequences only from the same genome: one can identify regulatory elements specific even to a single gene, and it does not require a method such as clustering to assemble a collection of coregulated genes [28].

This method seems to be relatively robust with respect to errors in the phylogeny. It can also be applied as a filtering step in a larger algorithm that uses other site

discovery techniques (e.g. using PWMs to discover sites and then restricting analysis to those sites which are well–conserved through phylogenetic footprinting [106]). However, there are some requirements that must be met in order for the method to be fruitful. Assuming that the foundational hypothesis (that regulatory elements are unusually conserved across evolution) is true, it is also necessary that the surrounding elements have changed significantly. That is, if the genomes being aligned are too closely related, then the regulatory element does not stand out in the alignment, and the alignment is not informative [28]. The general problem with alignments mentioned above also applies here: if the regulatory regions are short and the regions being aligned are long, then the regulatory elements may be lost in sequence noise, thus making the method uninformative.

Summary and analysis

Clearly, there is a large number of sequential data analysis tools available for use, ranging from simple alignment and homology detection to motif discovery based on sequences believed to be related. The above description of relevant and important sequence analysis tools is not intended to be exhaustive; it does, however, give a sense for the variety of sequence analysis problems that exist and the further variety of algorithms and tools to handle those problems. Of most relevance to this thesis are the motif discovery tools, as the goals of this thesis lie in that area of research.

There are three shortcomings in existing motif discovery tools that are relevant to this thesis: lack of exhaustiveness, parameter requirements, and lack of versatility for adaptation to new problems. Each of these will be addressed in turn.

The exhaustiveness/representation balance Looking at the variety of motif discovery algorithms available, one can broadly categorize them by the search strategies and motif representations they employ. Search strategies range from probabilistic searches (e.g., Gibbs sampling) and deterministic non–exhaustive searches (e.g., MEME) to exhaustive searches (e.g., `TEIRESIAS`). The choice of search strategy is closely linked to the choice of motif representation, e.g. regular expression or position

weight matrix.

In general, there is a trade–off: more descriptive motif representations frequently make exhaustive searches computationally infeasible. For example, TEIRESIAS is one example of an exhaustive motif discovery approach. To achieve this goal, TEIRESIAS uses regular expressions to represent its motifs. In fact, TEIRESIAS and other similar tools (e.g., Pratt [85]), actually limit their scope to only a subset of the regular expression language that can be enumerated exhaustively. For example, TEIRESIAS uses bracketed expressions — such as [KRF], which says that K, R, and F are acceptable characters — but does not use the "Kleene star", *, which means "zero or more" of a particular character. Thus, even though regular expressions are more informative than simple consensus sequences, some approaches are still not even able to capture all of the details that a full–featured regular expression implementation would contain.

Position weight matrices are more desirable than regular expressions (especially when compared to the limited implementations of regular expressions as in TEIRESIAS), as they capture the variability at a given position quantitatively rather than only qualitatively. As a result, PWMs tend to be a frequently used representation as they capture more of the "information" in the motif without over–fitting or requiring large amounts of training data. The parameters in a PWM are sampled from a continuous distribution, which in turn makes it impossible to enumerate the effectively infinite number of all possible PWMs that could describe a motif. Generally, tools that employ PWM representations tend to return only one "optimal" PWM or just a handful of such matrices; the methods used to find these optimal matrices are heuristic or probabilistic rather than exhaustive.

In this sense, the existing motif discovery tools for sequence analysis can be placed on a spectrum that ranges from exhaustive tools using simple motif representations to non–exhaustive tools using more complex representations. The majority of tools can be found at the extreme ends of the spectrum, with tools that exhaustively enumerate regular expressions (or single consensus sequences) at one end and probabilistic, PWM–based tools at the other. Table 2.1 gives a more detailed view of the different kinds of motif representation, their relative benefits and pitfalls, and potentially

appropriate applications and tools using those representations.

Given these competing demands of descriptiveness and exhaustiveness, it is certainly understandable that many tools do not attempt to be exhaustive. For extremely large datasets, motifs that are nearly indistinguishable from random noise (the "twilight zone" of motifs [94]), and other extremely complex problems, a provably exhaustive approach may not be computationally tractable. For example, a Smith–Waterman search of all known and deposited protein sequences is still beyond the current computational capabilities of an average researcher. However, given recent technological advances, there are certainly a large class of problems that are well within the reach of provably exhaustive search methods and that are not currently being addressed as such. One can imagine quite a few problems that are uniquely amenable to solution by an exhaustive approach (one will be presented in Chapter 4 of this thesis).

At the same time, we would like to increase our potential for complex motif representations in these exhaustive searches. While the regular expression approach of tools like `TEIRESIAS` is suitable for quite a few problems, it would be more desirable to uncover PWMs or other more descriptive representations that are better suited for certain classes of problems. As seen in Figure 2-5, one can imagine that by uncovering the motif instances, one can represent the motif in an arbitrary fashion. This then appears to be one aspect of motif discovery that has significant room for improvement: finding motifs instances in such a way that one is not restricted to certain motif representations.

Parameter requirements Another common shortcoming of motif discovery algorithms is the requirement for accurate parameter values. Again, this is in many senses a reasonable requirement — one cannot expect to find the relevant motifs in a dataset if one has absolutely no idea what those motifs may look like. A search that is exhaustive in parameter space, even if not exhaustive in sequence search space, is computationally intractable. In order to make motif discovery feasible, one needs to have some idea of the characteristics of the sought motif. However, the degree of

Table 2.1: Common motif representations.

model	description
exact "words"	Simply a string of nucleotides or amino acids. There is no allowance for more than one letter at a particular position in the motif.
IUPAC notation	A string of nucleotides or amino acids along with letters that represent more than one nucleotide or amino acid. This representation is a subset of regular expressions.
Regular expressions	A representation that allows multiple characters at each position and variable length motifs. Typically, motif discovery tools use a limited subset of the operations available in regular expressions.
Position weight matrices	A matrix where each row–column pair represents the probability of finding a particular letter at a particular position in the matrix. PWMs can more accurately capture the "biology" behind *cis*–regulatory elements and are, as a generalization, more specific than regular expressions.
Complex grammars	A representation similar to regular expressions, but allowing for dependencies between the positions in a motif. Complex grammars, such as context–free grammars, are commonly used to represent RNA folding motifs.
Markov models	Allows probabilistic dependencies between arbitrary positions in a motif. These models have many more parameters to train than PWMs and therefore require much more data.
Bayesian networks	A representation in which positions can depend on many other positions in a probabilistic manner. Like Markov models, Bayesian models require many instances of a motif for accurate training.

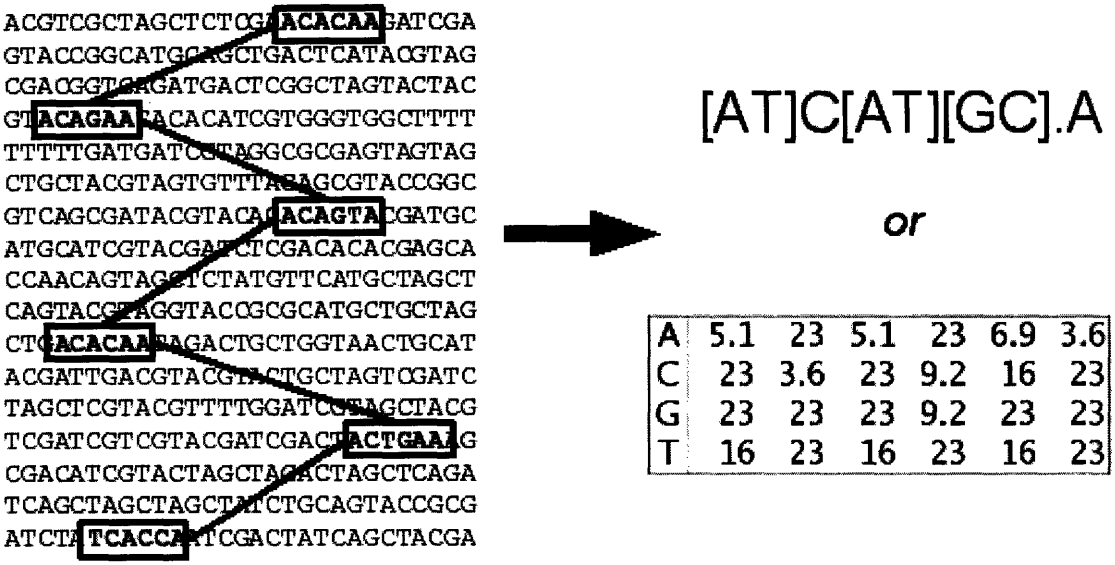


Figure 2-5: Pattern discovery in genomic sequences. The occurrences that are found are pairwise similar, but they are not identical. TEIRESIAS would represent this pattern with the regular expression in the top right, whereas one may instead represent those same motif instances by the position weight matrix in the bottom right. If one can uncover the motif instances directly, they could be arbitrarily expressed by either method, or any other motif representation.

specificity that is required by many motif discovery algorithms is sometimes burdensome or unrealistic. Even the most widely used tools require the user to provide an estimated length of the motif being sought, or at least a narrow range of possible lengths. In some cases, such as the search for binding sites of a transcription factor with a well–characterized structure and that binds to well–known lengths of DNA, this request may be reasonable. But if the transcription factor is not well–known, or if the length of its binding region is not well–characterized, then the input parameter given by the user will at best be an educated guess. Since many tools depend strongly upon these input parameters, this is an undesirable situation. While the development of an effective and efficient parameter–free motif discovery algorithm would be an overwhelming task, it is desirable to develop a tool that requires less specific information that a user is more likely to be able to provide without extensive *a priori* knowledge of the motif's origins.

Lack of versatility One final shortcoming of motif discovery algorithms is their distinct lack of versatility. In looking at the different tools, algorithms, problems, and data types discussed above, it is clear that there are hundreds of different tools used for the analysis of sequential data. Notably, though, most tools are useful for only a very small number of data types (and frequently just one data type) and a small number of problems. Sifting through the numerous available tools for the appropriate algorithm for a particular problem is an onerous task, and one that may not even be fruitful: since many tools are well–tuned to detect one kind of motif or solve one kind of problem, if one does not know enough about the dataset under study, it is difficult to know *a priori* what the most effective tool would be.

It would be of great use to have a versatile motif discovery tool that is capable of handling a variety of data types and a variety of different problems within those data types. The obvious disadvantage to this approach is that a more versatile or generic approach will necessarily give up exploiting specifics of a problem that can lead to decreased computational cost. While this is certainly a consideration, it is important to consider the many potential benefits of a generic approach. One obvious

benefit is the unification of various computational techniques under the umbrella of one algorithm. For instance, numerous clustering methods have been developed and applied to solving diverse motif discovery problems. If one could easily access multiple such methods in a versatile tool, it would be easier to determine the best approach for a specific problem. Additionally, a versatile approach would greatly help the process of solving new problems or addressing new types of data. When an algorithm is so finely tuned to solving a specific problems, it is often difficult to recast a new problem in such a way that an existing method can handle it. One such example has already been presented: the translation of real–valued data into string–valued data so that existing string analysis tools can be used. Some data types are even less amenable to such recasting, meaning that an entirely new tool may need to be developed to handle these data. A truly versatile tool would at least give an easy first attempt at handling these new problems, and may even be sufficient to handle most of the necessary analysis for these problems.

2.2 Metabolomics

Changing gears significantly from motif discovery, it is also important to understand the background of another field relevant to the work presented in this thesis: metabolomics.

Metabolites play an important role in numerous areas of academic, industrial, and medical interest. The definition of a "metabolite" that will be used in this work is a small–molecule cellular intermediate, typically of molecular weight less than 600, and in the polar phase of a cellular extract (which excludes lipids, typically considered to be a different field of study known as "lipidomics".) Industrially, metabolite accumulation is used as a route for chemical synthesis [36]; the ability to predict or refine an organism's production of a metabolite will have a significant impact on process cost and feasibility. Academically, metabolites are known to be of great relevance. A few well–known examples of metabolite regulatory interactions, including the effects of lactose and glucose [83] in the bacterium *Escherichia coli* and galactose [167] in

Saccharomyces cerevisiae, are used to teach basic regulatory principles in introductory biology courses. At the same time, though, much remains unknown about the functions of most metabolites as anything but intermediate products in the cell. Finally, metabolites are also known to be potential indicators of diseases and disorders in humans [155]; the ability to understand what these metabolites indicate and how best to assay for them may play a huge role in future medical diagnostics. For these reasons, it has become evident that a study of all metabolites, or metabolomics, has the potential for significant scientific impact.

Existing knowledge about metabolism varies by system and is typically useful but incomplete. One way of viewing metabolism (see Figure 2-6) is as a complex reaction network, similar to (for a chemical engineer) a convoluted reaction scheme in a chemical reactor or the network of combustion reactions that occur in a flame. There are many different intermediates that exist, some of which are in detectable pools while others are transient. Reactions may go almost to completion or may be equilibrium reactions. A great deal of information is known about the central reactions and pathways in the network (see Figure 2-7). In these cases, enzymes are known and kinetics are often known. In central carbon metabolism, it is even well–known which carbon atoms will end up in which products for reactions that break the sugar down. These well–studied examples of metabolism constitute extremely useful *a priori* knowledge for metabolomic studies. However, there are also portions of metabolism that are not as well characterized. Individual reactions may seem to be "missing" due to proteins that have not been characterized as performing those functions. Occasionally, a metabolic flux measurement can reveal that there is some unknown pathway or reaction that contributes to a known reaction flux. In these cases, metabolomics using existing techniques is more difficult; these difficulties will be addressed in Chapter 5.

Metabolomics focuses on large–scale analysis of small molecule products (metabolites) in cells and organisms, an extremely difficult experimental task. The most obvious reason for this difficulty is the chemical diversity of metabolites. DNA and proteins involve subunit–based chemistries; that is, they are copolymers of four or

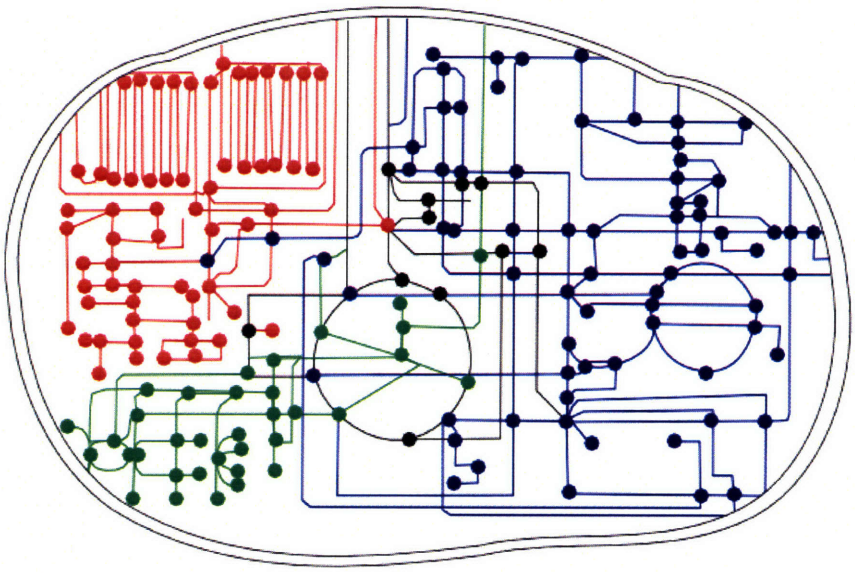


Figure 2-6: A metabolic network overlaid on a cell. Dots represent metabolites (substrates/products), while lines connecting those dots represent reactions. Different pathways or modules within the network are different colors. Reactions or metabolites where pathways intersect are may be of great significance.

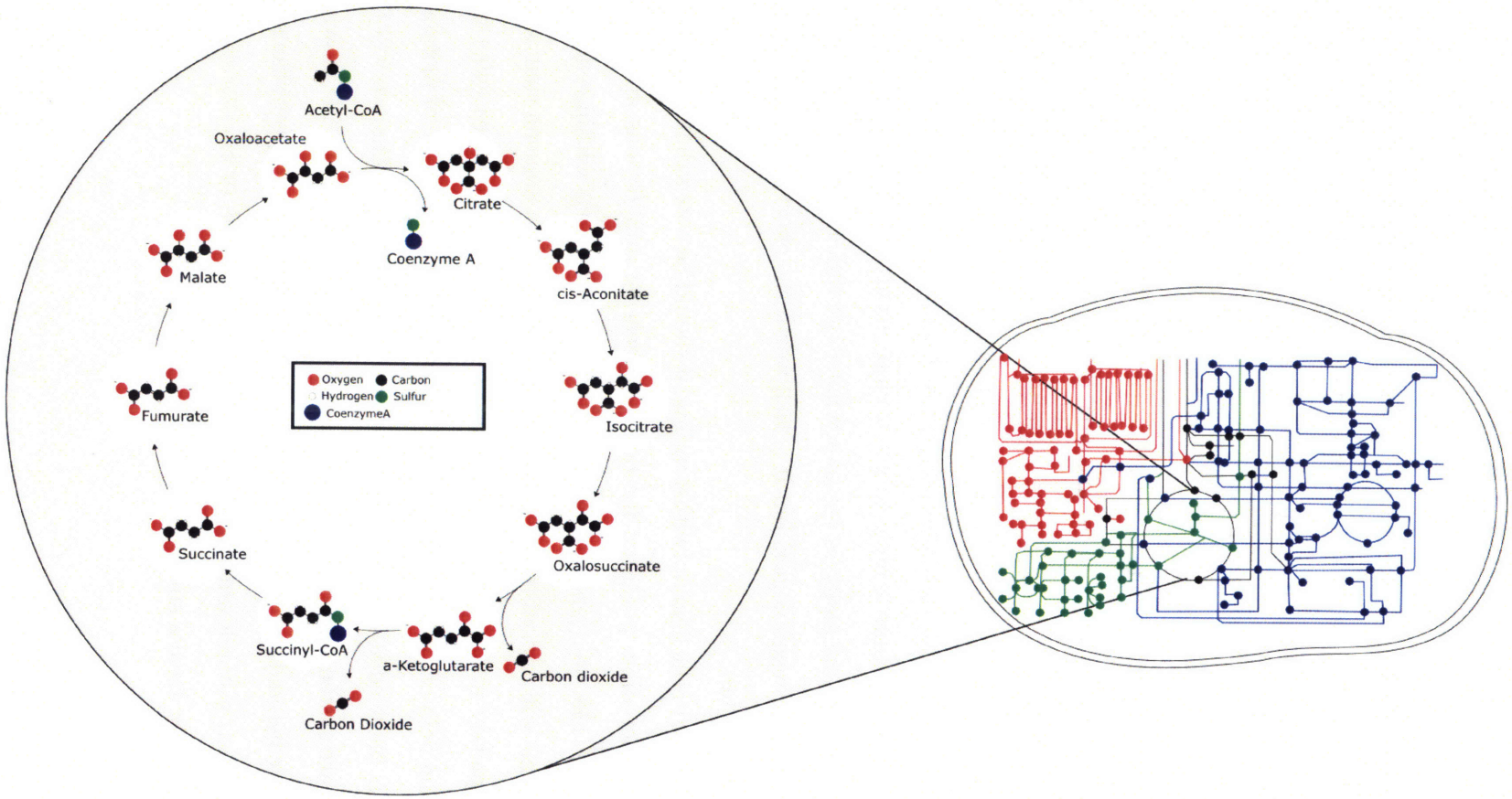


Figure 2-7: The tricarboxylic acid cycle, or TCA cycle. This portion of central carbon metabolism is extremely well–studied and well–characterized. Even the specific carbon atoms that are removed or added in various steps in this cycle are known. This is an example of a metabolic system where *a priori* knowledge can be extremely helpful in profiling metabolites and in determining the flux of different reactions. Even though the TCA cycle is well–known, there are other pathways that are not as well–studied and thus do not have such detailed information about them available.

twenty (respectively) unique monomers. Assaying and manipulating these polymers, compared to metabolites, is relatively straightforward. Even though a staggering number of DNA molecules or proteins can be formed from those subunits, a single experimental method can frequently exploit properties common to all of the subunits so as to act on millions of different polymers simultaneously. For metabolites, there is no common basis for such assays. Metabolites are diverse in their structure, size, hydrophobicity/hydrophilicity, charge/polarity, and potential for chemical reactions. As such, assays in metabolomics must typically strike a balance between how sensitive they are, how informative they are, and how diverse the metabolites analyzed can be [125]. No single analytical technique is optimal for measuring all metabolites. In many senses, metabolomics is an excellent complement to the well-established "omics" fields, proteomics and transcriptomics (see Figure 2-8). All three can feasibly be performed on the same sample to get a more accurate snapshot of cellular state. As noted already, these analyses are somewhat more difficult for metabolomics given the current state of technologies, but it is well worth the effort for the potential insight that it may provide.

The current hurdles in the field notwithstanding, it should be noted that metabolomics is a field that has incredible potential for both engineering applications and more fundamental science [138, 55]. Knowledge of metabolite interactions within cells and within larger systems will undoubtedly elucidate pathways and phenomena that were previously incompletely understood or even inaccessible. These advances will then further the growing field of systems biology. By expanding our knowledge of metabolites, and thus our knowledge of metabolism, regulation, and interactions, we will be better able to create systems-level models of biological systems that provide us with macroscopic knowledge and the ability to predict some phenomena. Applications of these models to industrially relevant organisms like *Escherichia coli* or *Saccharomyces cerevisiae* will no doubt have a substantive impact on society.

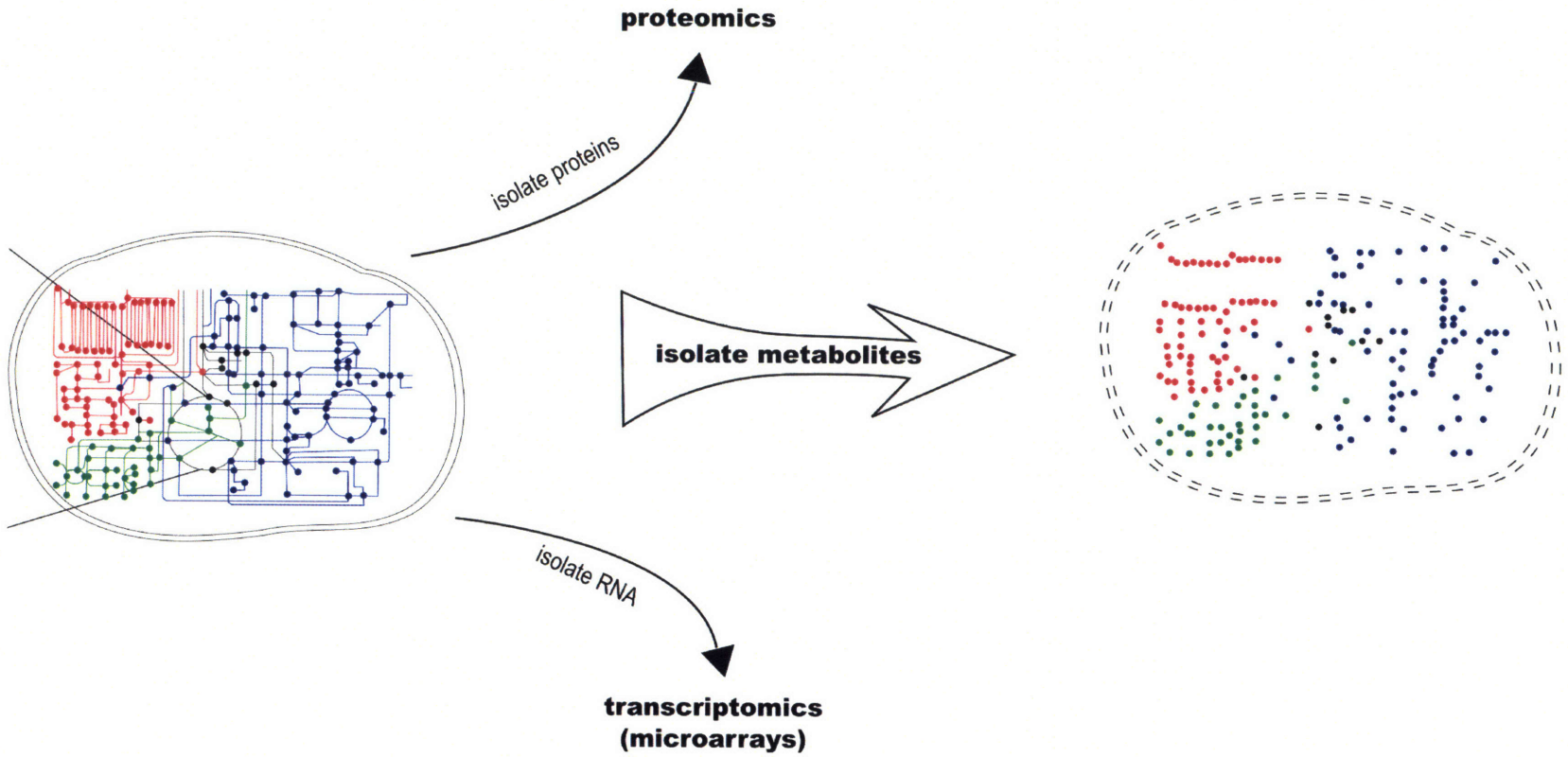


Figure 2-8: A multi–omics approach. Metabolomics can easily be serialized with transcriptomics and proteomics for more complete analysis of a sample. The available technologies make metabolomics currently more difficult than the other "omics" approaches, but just as promising for their potential insight. A workflow that captures information on all three of these levels — transcriptomics, proteomics, and metabolomics — will provide an extremely accurate snapshot of the cell.

Chapter 3

Gemoda, a generic motif discovery algorithm

3.1 Overview

As noted in Chapter 2, motif discovery in sequential data is a problem of great interest and with many applications. However, previous methods have been unable to combine exhaustive search with complex motif representations.

This chapter presents a GEneric MOtif DIscovery Algorithm (Gemoda) for sequential data. Gemoda can be applied to any dataset with a sequential character, including string–valued (or "categorical") data and real–valued data. The algorithm proceeds in three steps: comparison, clustering, and convolution. Gemoda deterministically discovers motifs that are maximal in composition and length. As well, the algorithm allows any choice of similarity metric for finding motifs. Finally, Gemoda's output motifs are representation–agnostic: they can be represented using regular expressions, position weight matrices, or any number of other models. This chapter will also demonstrate a number of possible applications in a variety of settings, including the discovery of motifs in amino acids sequences and the discovery of conserved protein sub–structures.

Much of the research and discussion in this chapter is drawn from the publication that describes our generic motif discovery algorithm:

- Jensen KL, Styczynski MP, Rigoutsos I, and Stephanopoulos GN. "A generic motif discovery algorithm for sequential data." *Bioinformatics* 22 : 21 – 28 (2006).

This chapter will frequently use "we" to refer to the collective authors of the manuscript and their contributions to the work.

3.2 Introduction

Chapter 2 discussed in great detail the wide variety of motif discovery methods used to find recurrent trends in data. In bioinformatics, the two predominant applications of motif discovery are sequence analysis and microarray data analysis. Less common applications include discovering structural motifs in proteins and RNA [75, 116]. Motif discovery in sequence analysis typically involves the discovery of binding sites, conserved domains, or otherwise discriminatory subsequences, while motif discovery in microarray data typically involves the discovery of gene expression motifs that are predictive of important phenotypes like cancer [63, 139].

Behind sequence alignments and homology detection methods, motif discovery is arguably the second most common form of sequence analysis. But, in contrast to sequence searching methods — which are dominated by a few, widely–accepted tools such as Blast [11] and FastA [131] — the landscape of motif discovery tools is much more fragmented, with dozens of specialized tools for as many different purposes. From a computational standpoint sequence searching is distinct from motif discovery; indeed, the latter is an NP–hard problem whereas the former can be solved in polynomial time [59].

There are many publicly available tools that are each quite adept at addressing a specific subclass of motif discovery problems. Some of the more commonly-used tools for motif discovery in nucleotide and amino acid sequences include MEME [17], Gibbs sampling [100], Consensus [72], Block Maker [70], Pratt [85], and TEIRESIAS [140]. More recently introduced, but less-widely used tools include Projection [34], MultiProfiler [94], MITRA [52], and ProfileBranching [135]. This list of tools is not

intended to be exhaustive; however, it is indicative of the wealth of options available for solving such problems.

All of the existing motif discovery tools for nucleotide and amino acid sequences can be classified on a spectrum that ranges from exhaustive tools using simple motif representations to non–exhaustive tools using more complex representations. The majority of the tools can be found at the extreme ends of the spectrum, with tools that exhaustively enumerate regular expressions (or single consensus sequences) at one end and probabilistic tools, based on position weight matrices (PWMs), at the other. This partitioning of tools is due to a computational trade–off: more descriptive motif representations such as PWMs frequently make exhaustive searches computationally infeasible.

Depending on the task at hand, a specific type of motif discovery tool may be more useful than others. For example, the PWM–based tools excel at finding *cis*–regulatory binding elements [166], whereas the regular expression–based tools are well–suited to finding conserved domains in large protein families [141]. Generally, it can be difficult to know *a priori* which motif discovery tool will be right.

3.3 Algorithm

3.3.1 Overview

Gemoda was designed to meet the demand for complex motif representations, like PWMs, while still being exhaustive. The philosophical underpinnings of the Gemoda algorithm can be traced back to TEIRESIAS [140]; Winnower [132]; the algorithm by [108]; and a variety of algorithms for association mining [183, 182]. In particular, Gemoda shares some of its logical steps with the TEIRESIAS algorithm while incorporating a more flexible definition of "similarity" and allowing motif representations other than regular expressions.

Gemoda's design goals can be summarized as follows: *exhaustive discovery* of all *maximal motifs* in a way that allows flexibility in *motif representation*, incorporation

of a variety of *similarity metrics*, and the ability to handle diverse *sequential data types*. Each point of emphasis can be explained as follows:

- **Exhaustive discovery:** Gemoda's combinatorial nature provides an algorithmic guarantee that all motifs meeting certain criteria are deterministically discovered.

- **Maximal motifs:** Gemoda returns only motifs that are maximal in both length and composition with respect to the similarity and clustering functions.

- **Motif representation:** The motifs discovered by Gemoda are reported as short multiple sequence alignments (in the case of motif discovery in nucleotide and amino acid sequences) and can be modeled using regular expressions, PWMs/PSSMs, Markov models, or any other representation.

- **Similarity metrics:** Any criterion, ranging from sequence alignment scores to geometric functions, may be used to compare sequences.

- **Sequential data types:** The nature of Gemoda's computations is not unique to any specific type of data, and thus can be used on any data with a sequential character — that is, data in which there is a natural left–to–right order, such as a sequence of nucleotides or amino acids. In the most general sense (and the one adopted in this work), sequential data also include real–valued series data, such as a stock price or the ordered (x, y, z) triplets of an alpha–carbon trace in a protein structure.

The algorithm has three distinct logical phases: comparison, clustering, and convolution. During the comparison phase, short overlapping windows in the data set are compared. During clustering, these windows are grouped together to form elementary motifs. Finally, during convolution, these motifs are "stitched" together to form maximal motifs (see Figure 3-1). In the following sections, we give some brief definitions and nomenclature, then describe each of the algorithm's three phases in detail. Finally, we illustrate a few applications of Gemoda.

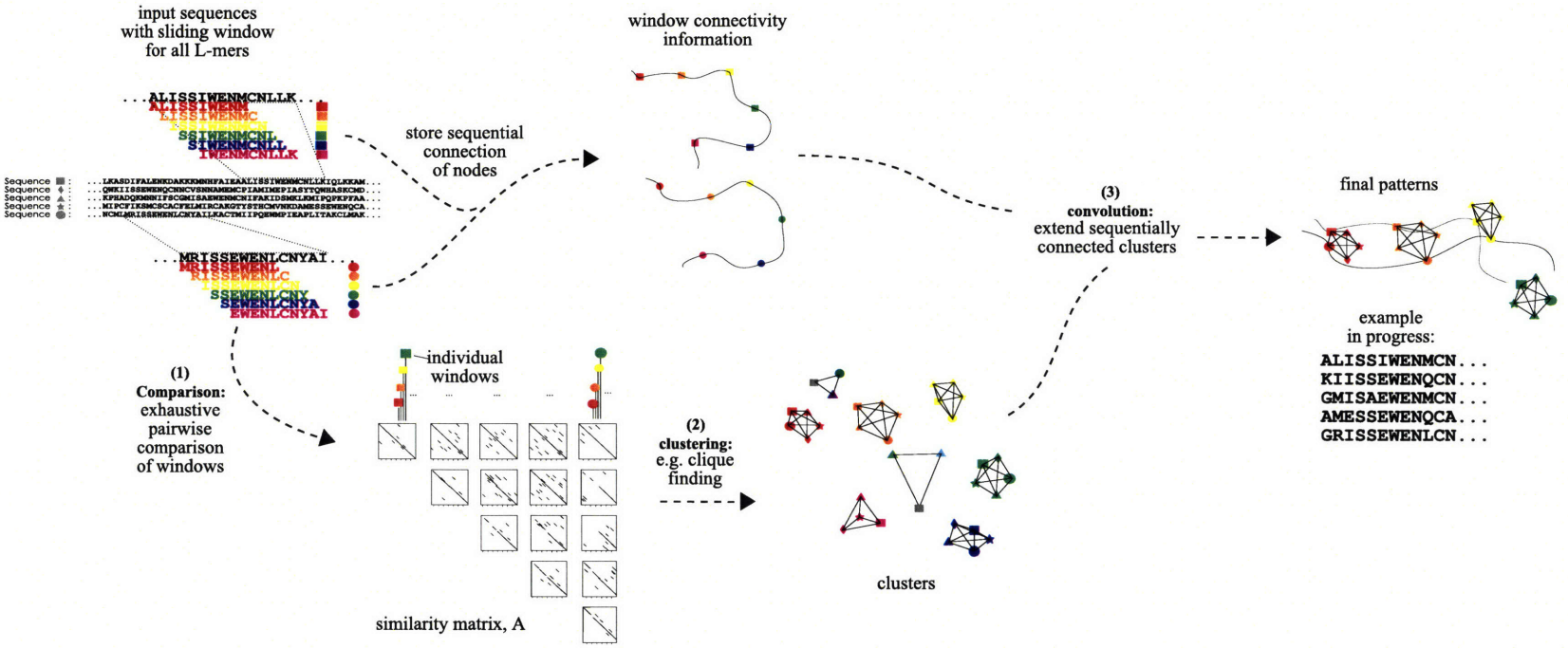


Figure 3-1: A sketch showing the flow of the Gemoda algorithm for an example input set of protein sequences. The various colors in the input sequences are used to indicate the sequential ordering of the L–residue windows. The various shapes are used to indicate a particular window's sequence of origin. (1) In the comparison stage, each window is compared to each other window on a pair–wise basis. Here we show the similarity matrix, A, where the values in the matrix have been thresholded. Those pairs of windows in A that have a similarity score above the threshold are colored black. Note that the graph looks very similar to a standard dot plot. (2) In the clustering phase, groups of windows are clustered together. Here, we show the clusters as cliques, or maximal fully–connected subgraphs in the thresholded matrix A. (3) Finally, these clustered are "stitched" together in the convolution phase using the sequential ordering of the windows to reveal the maximal motifs. A similar process applies for any kind of sequential data analyzed by Gemoda.

3.3.2 Preliminary definitions and nomenclature

The input to Gemoda is a set of sequences of data points $S = \{s_1, s_2, \ldots, s_n\}$, where sequence s_i has length W_i. So, for example, the j^{th} member of the i^{th} sequence is denoted by $s_{i,j}$. Each $s_{i,j}$ is a primitive, or atomic unit, for the data that is being analyzed. For time–series data, $s_{i,j}$ may be a point sampled from $\mathbb{R}^K$ (with K arbitrary), whereas for a DNA sequence it would be one of the characters {`A,T,G,C`}.

Typically, one seeks motifs of a minimal, domain–dependent length. We denote this minimum length by L and we define a matrix A of size $N \times N$, where $N = \sum_{i=1}^{n}(W_i - L + 1)$. That is, A is a matrix with one row and one column for each window of size L in our entire sequence set. For example, the 10^{th} window of size L in the 5^{th} sequence would be expressed as $s_{5,10:10+L-1}$, where "$10 : 10 + L - 1$" denotes "position 10 through position $10 + L - 1$, inclusive." To keep track of which window corresponds to which index in A, we define the one–to–one function $\mathscr{M}(s_{i,j:j+L-1}) \mapsto q \in [1, N]$. (For simplicity, we define $(s_{i,j} + 1)$ to be $s_{i,j+1}$, unless $s_{i,j+1}$ does not exist, in which case $(s_{i,j} + 1)$ is undefined.) Similarly, $\mathscr{M}^{-1}(q) \mapsto (s_{i,j:j+L-1})$ such that $i \in [1, n]$ and $j \in [1, W_i - L + 1]$.

We also define a similarity function $\mathscr{S}(s_{i,j:j+L-1}, s_{q,z:z+L-1})$, that takes as arguments two arbitrary windows and returns a real–valued number indicating the level of similarity between the two windows. In the most simple case, $\mathscr{S}$ may use the identity matrix to count how many DNA bases two windows have in common; for real–valued data, the function may return the sum–of–squares error between two windows or any other measure of similarity.

We define a motif p as a data structure with two features: a width $\mathscr{W}(p)$ and a list of locations in the data where the motif occurs, $\mathscr{L}(p)$. A motif has the property that the locations in $\mathscr{L}(p)$ meet some predefined clustering requirements (discussed below) based on the similarity function $\mathscr{S}$ for each window of length L within the motif. The support of a motif is equal to the number of its occurrences (or "embeddings"), $|\mathscr{L}(p)|$.

We say a maximal motif is a motif which has the following properties:

- The motif's width cannot be extended in either direction (left or right) without

producing a motif with fewer embeddings (i.e., without $|\mathscr{L}(p)|$ decreasing); and

- The motif is not missing any instances, i.e. $\mathscr{L}(p)$ includes the locations of all instances of the motif.

These two criteria can be summarized qualitatively by stating that a maximal motif is not "missing" any locations and is as wide as possible, and thus it is as specific and sensitive as possible.

Given these explanations and definitions, we can now detail the computations involved in each phase of the Gemoda algorithm.

3.3.3 Phases in Gemoda

Comparison phase

In the comparison phase of the Gemoda algorithm, the sequences are divided into overlapping windows of size L which are then compared to each other in a pairwise manner to produce a similarity matrix, A (see Figure 3-1). Formally, $A_{i,j}$ is equal to $\mathscr{S}(\mathscr{M}^{-1}(i), \mathscr{M}^{-1}(j)) = \mathscr{S}(s_{i,j:j+L-1}, s_{q,z:z+L-1})$.

A is then, quite simply, a similarity matrix for all N windows based on the similarity function $\mathscr{S}$. In most cases, $\mathscr{S}$ is commutative (and the A matrix is symmetric); however, this is not a requirement.

Clustering phase

The purpose of the clustering phase is to use the similarity matrix A to group similar windows in clusters. These clusters will become "elementary motifs" from which the final, maximal motifs will be constructed.

We define a clustering function $\mathscr{C}(A) = c^L = \{c_1^L, c_2^L, \ldots, c_Z^L\}$ where each c_i^L is a set of indices in A and $c_i^L[q]$ is the q^{th} member of c_i^L. Note that $\mathscr{C}$ can be any function; common clustering functions include hierarchical clustering, k–nearest–neighbors clustering, and many others. We call each c_i^L an "elementary motif" of length L. We note that a clustering function may assign each node (window) to one

or more groups. In the latter case, each c_i^L may have a non–null intersection with any c_j^L.

Convolution phase

The purpose of this phase is to "stitch together" the elementary motifs to generate the final, maximal motifs [140]. For the purposes of Gemoda (and consistent with the above concept of convolution), we say that a motif h of width $\mathscr{W}(h) > L$ meets the similarity criterion if for each window of length L completely within the motif, all instances participate in a cluster together based on $\mathscr{S}$ and $\mathscr{C}$. In this manner, we can piece together longer continuous motifs from smaller motifs that all meet the similarity criterion over windows of length L.

Next we define the "directed intersection" of two elementary motifs, $c_i^L \curvearrowright c_j^L = c_r^{L+1}$, where c_r^{L+1} is the set of those indices q in c_i^L such that $\mathscr{M}(\mathscr{M}^{-1}(c_i^L[q]) + 1)$ is in c_j^L. That is, c_r^{L+1} is the set of indices in c_i^L that are located, in the sequences S, one position earlier than the indices in c_j^L. c_r^{L+1} is then a motif of length $L + 1$.

We define the operation "$\sqsubset$" as follows: $c_i^L \curvearrowright c_j^L \sqsubset c^{L+1}$ is true if the set of indices $c_i^L \curvearrowright c_j^L$ is a subset or a superset of the indices in any member of c^{L+1}. This operation compares a convolved motif of length $L + 1$ to all previously–convolved motifs of length $L + 1$ to identify significant overlap: if the list of locations in the proposed motif is a superset or subset of the list for any other motif, the result of this operation is true. With this step, Gemoda can identify and eliminate redundant and non–maximal motifs.

If $c_i^L \curvearrowright c_j^L \sqsubset c^{L+1}$, then all super– or sub–sets of the proposed convolved motifs are removed from c^{L+1}; these windows are then taken together with the proposed motif, and the union of those sets of windows is returned to c^{L+1}.

Our objective is to find all the maximal motifs in the sequence set using the elementary patterns. We do this by performing $c_i^k \curvearrowright c_j^k$ for all i and j at each length $k \geq L$ until c^k is empty ($|c^k| = 0$). We then define the set of maximal motifs comprising c^k for all k as P, the final set of motifs that are returned to the user. This simple induction scheme guarantees that all (and only) the maximal motifs are in P

given appropriate clustering functions (see Section 3.3.4).

3.3.4 Implementation

Choice of clustering function

Gemoda can use any clustering function; however, as the size of the input sequence set increases, storing the matrix A can become practically difficult. In these cases, it can be easier to store true/false, or "similar"/"not–similar", values in A, where the value is true if the similarity score between two windows is better than a user–defined threshold g. The matrix A can then be viewed as an unweighted, undirected graph with a vertex for each window and edges between those nodes with pairwise similarity scores better than g (see Figures 3-1 and 3-2). When constructed as such, we have found that clustering functions based on finding either cliques[1] (see Figure 3-3 or connected components (maximal disjoint subgraphs) can be effective for motif discovery in diverse applications.

In the case where the clustering function $\mathscr{C}(A)$ is chosen such that each c_i^L is a clique in the g–thresholded A matrix, the Gemoda algorithm has a guarantee of compositional and length maximality, relative to the threshold g. (This is due to the nature of the clusters that are created: the result of convolving any two cliques will always be a clique because all members are pairwise similar, so the convolution process will necessarily produce maximal cliques.) That is, Gemoda will discover all motifs where each pair of instances has a similarity score better than g over every window of size L, there are no "missing" instances having this property, and the motif cannot be extended either to the left or right (see Section 3.3.4).

Clique enumeration is NP–complete [59, 165]; however, in practice this complexity is usually not an issue because the density (the ratio of the number of edges to the number of vertices) of graphs is usually low for datasets of nucleotide or amino acid sequences (with reasonable choice of g).

[1] We define a clique as a maximal, fully–connected subgraph. It may be alternatively defined without the requirement for maximality, thus making the clusters we discuss "maximal cliques". We use the former definition for the sake of brevity and clarity when discussing the maximality of extending motifs.

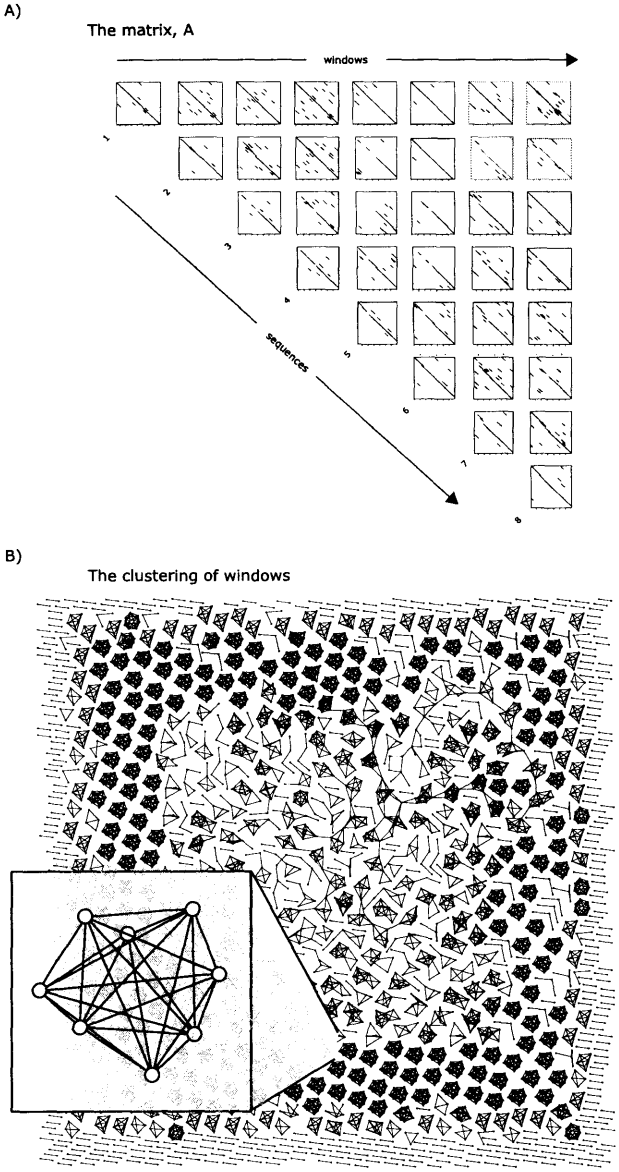


Figure 3-2: The similarity graph for the 3.1.7.2 enzyme example. (A) is the similarity matrix A, which contains one row and column for each window of 50 residues in the set of input sequences. Entries in the matrix have been thresholded such that pairs of windows that can be aligned with a bit–score greater than 20 are given a black dot and all others are white, producing the familiar dot–plot appearance of the matrix. (B) is a graph representation of A. Each vertex represents a window, and two vertices are connected with an edge if they have a black dot in the top image. The breakout shows a clique of size eight, which represents a set of windows that participate in the motif shown in Figure 3-6. In general, as the bit–score threshold is lowered, the number of edges in the graph increases, making the clustering stage more computationally intensive. When using clique–based clustering with too small of a threshold, computational expense may make the problem infeasible. At these thresholds the "signal" cannot be distinguished from the "noise." However, with the parameters used in this example, the clustering phase is quite easy, which is intuitive given the number of disjoint subgraphs shown in the bottom image.

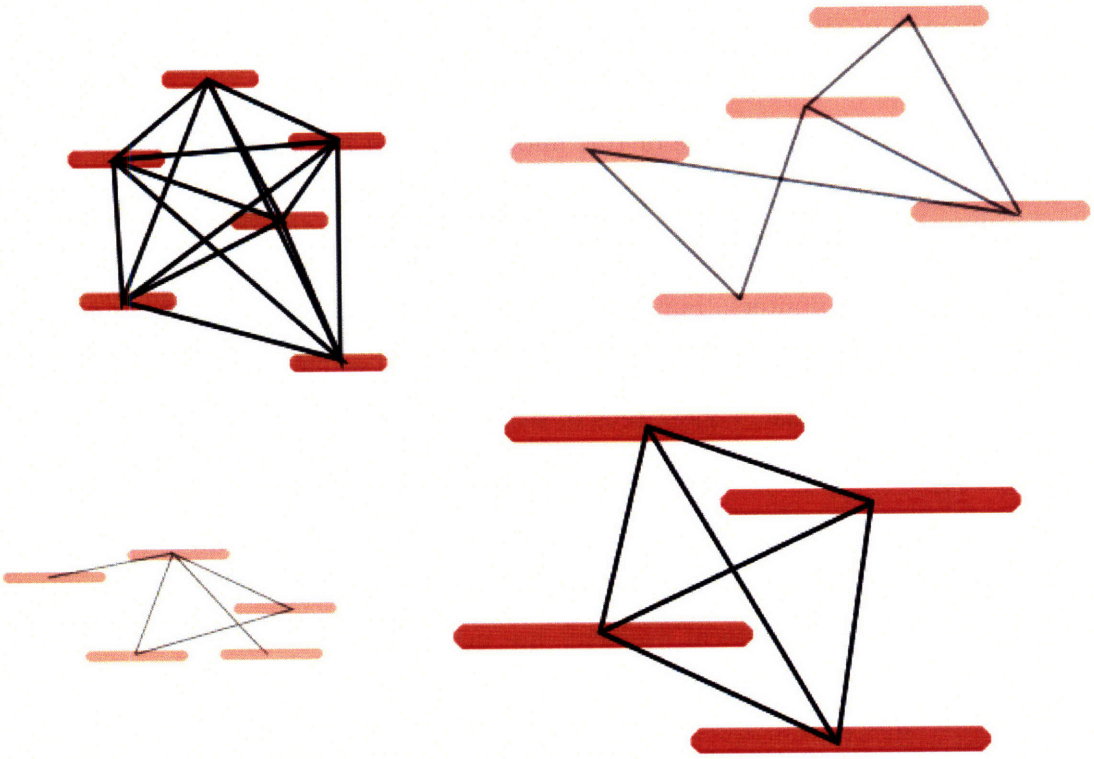


Figure 3-3: Representation of some clique and non–clique subgraphs. Filled ovals represent nodes, while lines between them represent edges. The fainter subgraphs are not cliques because each node is not pairwise connected to every other node in the subgraph. The bolder graphs have every node pairwise connected, forming cliques of different sizes.

Of course, as mentioned earlier, any clustering algorithm can be used for $\mathscr{C}$. In the case where the clustering function $\mathscr{C}(A)$ is chosen such that each c_i^L is a maximal disjoint subgraph in the g–thresholded A matrix (i.e., c^L represents the connected components of A), the computational complexity for the clustering phase is significantly less than for clique–based clustering. As well, in the case where Gemoda is applied to nucleotide and amino acid sequences, the motifs from this "connected components" method may be more intuitive than motifs found using clique–based clustering.

The space and time usage of this implementation is not unreasonable. In most cases, memory usage is not a limiting factor. For instance, the peak memory usage for a large sequence set containing 65,000 characters is 1 GB, well within the reach of many personal computers. Furthermore, the examples given in this work can all be done in reasonable execution times. The amino acid sequence example and protein structure example take at most tens of seconds on 1.5 GHz Pentium PCs, while the DNA sequence example takes about two hours on a 2.0 GHz Xeon processor. These times are more than reasonable given the exhaustive guarantees provided by the algorithm. While larger input sets and lower similarity thresholds may extend execution time, we believe that typical problems are in the range of reasonable computation time.

Inductive proof of exhaustive maximality

When using clique–finding as the clustering function, each elementary pattern of length L is a clique in our similarity graph. That is, the elementary pattern is a set of windows that are all similar on a pairwise basis and there is no other window that can be added to the set.

When the algorithm enters the convolution stage, it starts by convolving each length L elementary motif with all of the others. An elementary motif that is *non–maximal* can be convolved with another elementary motif to yield a motif at level $L+1$ that has the same cardinality. All such motifs are marked as non–maximal. Those elementary motifs that remain unmarked cannot be extended on either side

without losing support; since they are cliques we know they cannot be made greater in cardinality. Thus, all such unmarked cliques of length L can be labeled as maximal motifs and saved for output. In this way, we know that only maximal motifs will be returned to the user, and all such motifs will be returned.

When the "$\sqsubset$" operation is performed on two elementary motifs of length L that are being convolved, it ensures that no identical motifs of length $L+1$ exist and that no motif of length $L+1$ is a subset of any other. Additionally, since we have exhaustively compared a complete list of elementary motifs, and all such motifs are cliques with maximum cardinality, we are certain that all possible comparisons between motifs are being made. That is, no unique motifs of length $L+1$ could be created that are not subsets of motifs created by our exhaustive comparison. Finally, it is important to note that the result of convolving any two cliques will always be a clique. We know this because we take the set of all instances that can be extended (so the subgraph is maximal) and because all instances that are extended were pairwise similar in both windows being convolved (thus meeting our definition of similarity over multiple windows).

Thus, since Gemoda exhaustively generates *all possible* cliques of length $L+1$, and every added motif of length $L+1$ is maximal in support, we then know with certainty that c^{L+1} is an exhaustive list of motifs, or cliques, of length $L+1$. The induction step is then trivial, as setting L equal to $L+1$ at each step gives an exhaustive list of cliques just as when we started with c^L. This allows for a continual guarantee of exhaustiveness and maximality in output. The obvious termination condition for the algorithm is when $|c^i| = 0$. The following pseudocode sketch faithfully encapsulates the inductive algorithm described above.

```
begin
  n := 0
  while |c^n| ≠ 0 do
      for i := 0 to |c^n| step 1 do
          ismaximal := true
          for j := 0 to |c^n| step 1 do
              f := c_i^n ↷ c_j^n
              if |f| ≠ 0
                if f ⊏ c^{n+1} = false
                  c^{n+1} := c^{n+1} ∪ f
              else
                  choosemaximal(f, c^{n+1})
                fi
                if |f| = |c_i^n|
                  ismaximal := false
                fi
              fi
          od
          if ismaximal = true
            P := P ∪ c_i^n
          fi
      od
      n := n + 1
  od
end
```

Estimation of motif significance

The absolute significance of motifs depends strongly on the choice of the similarity metric and clustering function and is difficult to derive *a priori*. However, for a specific pair of similarity metric and clustering function, the *relative* significance can be easy

to calculate. For the clique–based clustering function described above, the relative significance can be estimated solely from the matrix A using a bootstrapping method. Such significance calculations are equally valid for many different motif discovery problems (e.g., nucleotide sequences or protein structures) because the calculation method uses only the matrix A: it is data–type agnostic.

Each pair of nodes in a similarity graph can be described with two different quantities: $\eta_{i,j}$, the number of neighboring nodes (including each other) that the two nodes have in common, and $\chi_{i,j}$, the number of consecutive windows starting from each of those nodes that are connected to each other. For instance, if window 1 is similar to windows 1, 10, 25, and 36, and window 10 is similar to windows 1, 10, 25, and 37, then these two nodes have three neighbor nodes in common and $\eta_{1,10} = 3$. If window 1 is similar to 10, 2 is similar to 11, and 3 is not similar to 12, then there are two consecutive similar windows and $\chi_{1,10} = 2$.

By analyzing each node as above, we can accumulate a matrix of graph statistics, Φ, such that

$$\phi_{i,j} = |\{(x,y) : \eta_{x,y} = i, \chi_{x,y} = j, 0 \leq x, y \leq N\}| \tag{3.1}$$

(where the vertical bars indicate the cardinality of the set, or the number of ordered pairs) and

$$\Phi_{i,j} = \sum_{a=i}^{\infty} \sum_{b=j}^{\infty} \phi_{a,b} \tag{3.2}$$

These statistics can then be used in the following calculation for $p_{rel}(q,r)$, the relative likelihood of an output motif of length q and support r given the calculated similarity matrix:

$$p_{rel}(q,r) = \binom{N}{r} \left[\prod_{i=0}^{r-2} \left(\frac{\Phi_{i,1}}{\Phi_{i,0}} \right)^{r-i-1} \right] \left(\frac{\Phi_{r,q-L+1}}{\Phi_{r,1}} \right) \tag{3.3}$$

In this equation, the combinatorial factor represents the number of different ways that windows can be sampled in groups of r, the cumulative product represents the necessary conditions for the formation of a clique of length L, and the last factor represents the likelihood of extending a clique of support r to be length q. In this way, the relative likelihood measure attempts to represent the expected number of

motifs of length q and support r that would occur at random given the calculated similarity matrix. Notably, this significance is based solely on the similarity matrix A, and so it can be used for either categorical or real–valued sequence data clustered with the clique–finding method.

Summary of user–supplied parameters

The input to Gemoda is a set of sequences (categorical or real–valued), a window length, a similarity function, and a clustering function. Various clustering functions may require other parameters. For example, the clique–finding and connected components clustering algorithms discussed above require both a threshold parameter g and, optionally, a minimal support parameter k. g is as defined earlier, and k is the minimum number of times that a motif must occur for it to be returned to the user. Other parameters can be easily incorporated into various clustering functions, such as a "unique support" parameter p that limits returned motifs to those that occur in at least p different sequences.

Availability

We have written open source programs implementing the Gemoda algorithm that are publicly available at the following URL: `http://web.mit.edu/bamel/gemoda`. The software includes a number of "helper" applications for interoperability with common bioinformatics tools. For example, applications are included that allow users to model Gemoda's output motifs (in the case of nucleotide or amino acid sequences) as PSSMs — using the pftools package available via the Prosite database [74] — or as hidden Markov models, using the popular HMMer software [48].

The implementation is distributed in two variants, each with a different comparison stage of the algorithm. The gemoda–s variant is for motif discovery in FastA–formatted text strings, typically nucleotide or amino acid sequences. The gemoda–r variant is used for motif discovery in sets of multi–dimensional, real–valued sequences. The gemoda–s variant is distributed with a number of similarity functions based on various nucleotide and amino acid substitution matrices. The gemoda–r variant is

distributed with similarity functions based on the root mean square deviation, with options for optimal translation and rotation.

3.4 Simple examples of Gemoda's motif discovery process

To better understand the entire process necessary to use Gemoda for motif discovery, I will present two brief examples. The first will go into great detail about the specific steps a user would have to take on her computer. The second example will expand beyond the simplicity of the first to investigate the effects of different comparison and clustering methods.

3.4.1 A trivial alphabetic example

We first imagine that we would like to find the motif(s) present in two given sequences, `ABCDEFG` and `ABCEFDG`. These would be represented with by following Fasta–formatted file:

```
> Sample 1
ABCDEFG
> Sample 2
ABCEDFG
```

We will use a window of length 3, a minimum similarity of 1, a clique–finding clustering method, and the similarity function defined as the identity matrix, where only exact matches are considered to be similar; that is, in comparing two sequences, if the first letters are both `A`, then we add a one to the cumulative similarity score; if the first letter of one sequence were `A` and of the other were `B`, then we would add nothing to the cumulative similarity score for the two sequences. The hypothetical command–line argument for the software implementation of Gemoda provided by the authors would look something like this:

```
$ gemoda-s -i testSeqs -l 3 -g 1 -k 2 -m identity_aa
```

Given this command, Gemoda finds the maximal motif `ABC..FG` (using a simple regular expression representation). How this happens is illustrated in Figure 3-4.

Windows 1 and 6 are identical, while windows 2 and 7 are 66% identical. Windows 3 and 8 have their first letter in common, allowing them to meet the similarity threshold. Windows 4 and 9 have their last letter in common, allowing them to meet the similarity threshold. In the case of a 2–clique as in this problem, convolution reduces graphically to following diagonal "streaks" of similarity that are not on the main diagonal. This streak is evident in part b of the figure.

Giving the above–mentioned input data and parameters to Gemoda, we get back not only the motif that can be represented as `ABC**FG`, but also two other motifs that may not have been readily obvious. The complete output of Gemoda is as follows:

```
pattern 0:      len=7   sup=2   signif=1.000000e+00
    0    0       ABCDEFG
    1    0       ABCEDFG

pattern 1:      len=5   sup=2   signif=5.000000e+00
    0    1       BCDEF
    1    2       CEDFG


pattern 2:      len=5   sup=2   signif=5.000000e+00
    0    2       CDEFG
    1    1       BCEDF
```

These additional motifs are due to the low similarity threshold; one letter of similarity is sufficient to make three consecutive windows all meet the threshold.

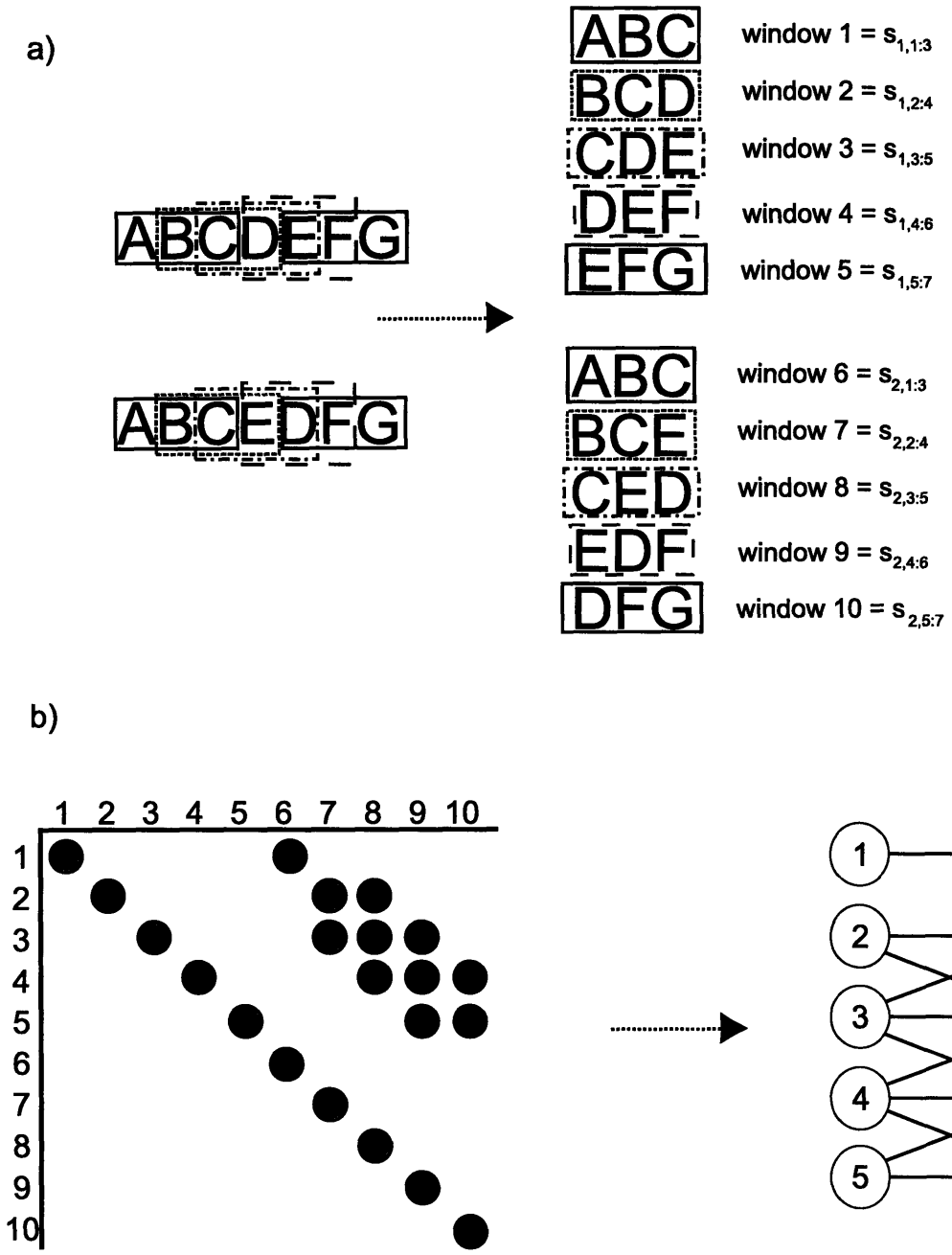


Figure 3-4: An extremely simple example demonstrating how Gemoda will find motifs in string–valued sequences. All possible windows are enumerated in (a) and pairwise compared; all pairs that have at least one letter at the same position have a dot placed in the appropriate entry in (b). This similarity matrix can also be represented as an unweighted, undirected graph.

3.4.2 A natural–language example

To demonstrate exactly how various steps in the algorithm work, we now provide a simple, natural–language example along with a description of the actions Gemoda would take at each step. Suppose we have a set of three words,

MOTIF

MOTOR

POTION

and we would like to find the motifs that some of these words share in common. Further, suppose that we are only interested in motifs that are at least four letters long and for which at least three of the four letters are "similar" between the windows. In this example, each word is a sequence, and the parameter L is 4. Thus, there are 7 possible windows that are taken sequentially from the three input sequences, numbered as shown in figure 3-5.

If we choose a similarity function based on the identity matrix with a threshold of three — that is, for two windows to be similar, at least three letters must be the same — then we find that only the following pairs of windows are similar: $(1,3)$, $(1,5)$, and $(2,6)$. Importantly, we note that though window 1 is similar to both windows 3 and 5, windows 3 and 5 are not similar to each other.

If, on the other hand, we choose a similarity function based on a matrix that distinguishes only between vowels and consonants — that is, any vowel is considered similar to any other vowel, and the same goes for any consonant — we would see different results for the same threshold value. In this case, we would find the following set of similarities: $(1,3)$, $(1,5)$, $(3,5)$, $(2,4)$, $(2,6)$, and $(4,6)$.

Given these similarity matrices for the different similarity functions, we can now cluster the graphs. Using the similarity matrix from the identity function, a clique–finding algorithm would find no cliques larger than size 2; that is, the only cliques that exist are the pairs of similar nodes. Since window 3 (`MOTO`) is not similar to window 5 (`POTI`), they cannot be in the same cluster.

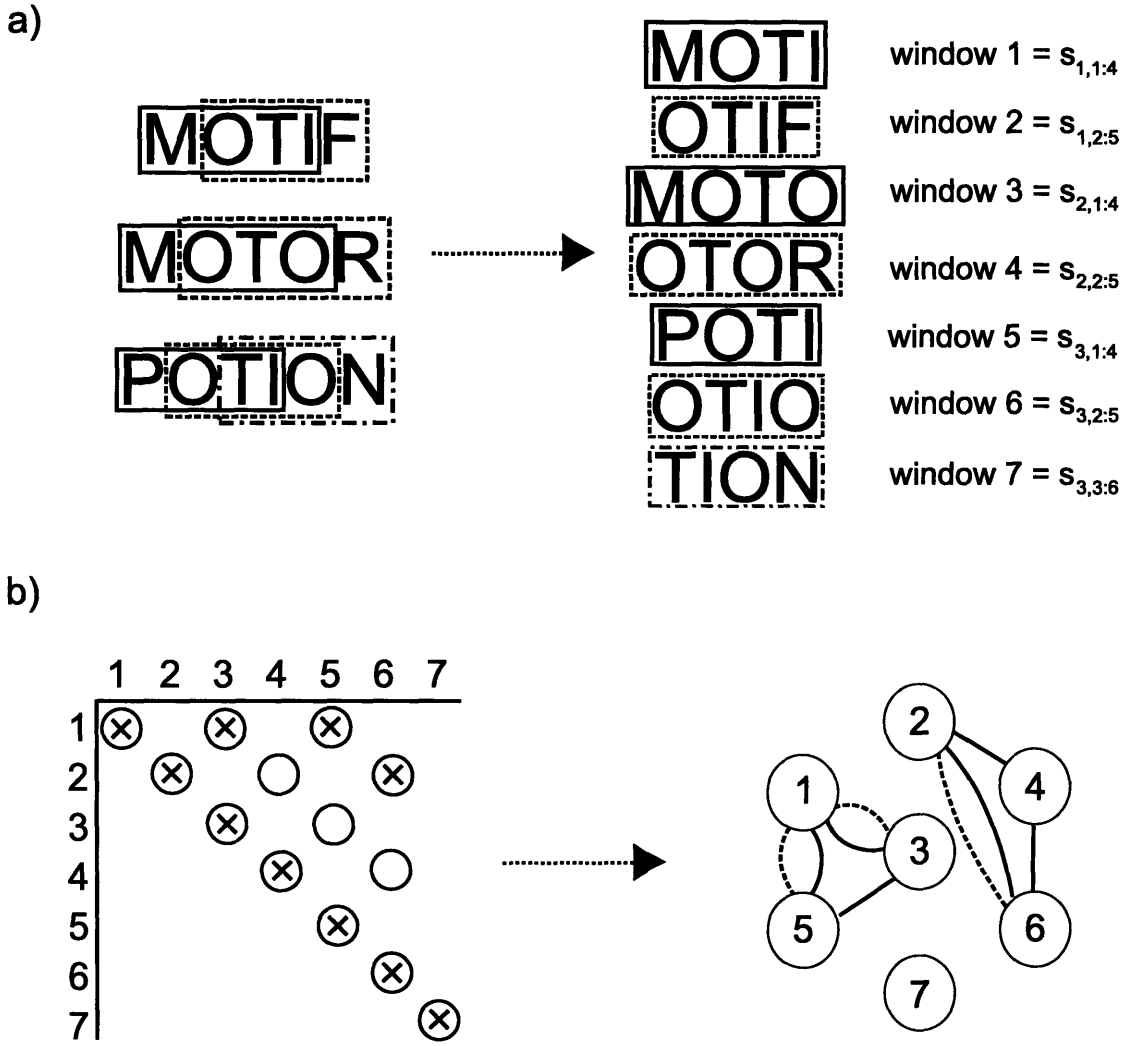


Figure 3-5: A natural language example illustrating the steps that Gemoda takes to discover motifs. In a), we see the three words, or sequences, being broken into overlapping windows of four letters each. Gemoda would then compare each of these windows to each other using either of the similarity metrics described in the text. In b), we see the resulting similarity matrix and how it looks when drawn as a graph. In the matrix, two nodes are similar by the identity metric if there is an "X" at their intersection, while they are similar by the vowel/consonant metric if there is an "O" at their intersection. Making each window a vertex and connecting vertices with an edge if the windows are similar, we obtain the graph on the right. Dotted lines indicate similarity by the identity metric, while solid lines indicate similarity by the vowel/consonant metric. In this representation, it is clear what the results of both clique–finding and commutative clustering methods will be.

However, if we use the similarity matrix produced by the weaker vowel/consonant function, we will find exactly two cliques of size 3: $\{1,3,5\}$ and $\{2,4,6\}$. Though there exist pairs of nodes that are similar, none of them is a clique because they are not maximal — that is, each individual pair of nodes that is similar (e.g., $(1,3)$) can have another node added to its set (5) without violating the pairwise similarity constraint, so only the larger set is a clique.

We also note that applying a connected components clustering function to the matrix created by the identity function would give still different results. In the connected components clustering function, the fact that windows 3 and 5 are not similar would not prevent them from being in the same motif; the function finds all disjoint subgraphs and defines them as the motifs. The motifs for such a case would be $\{1,3,5\}$ and $\{2,6\}$, which we will call motifs c_0^L and c_1^L, respectively.

Finally, we perform the convolution step. Using the last set of motifs described (with connected components clustering and the identity similarity function), we perform the convolution operation on each ordered pair of motifs; in this case, it means performing $c_0^L \curvearrowright c_1^L$, $c_1^L \curvearrowright c_0^L$, $c_1^L \curvearrowright c_1^L$, and $c_0^L \curvearrowright c_0^L$. For the first operation, we find the windows immediately after each of the windows in c_0^L, which is the set $\{2,4,6\}$. The intersection of this set with motif c_1^L is the convolved motif of length $L+1$, which is $\{2,6\}$; we can call this c_0^{L+1}. In performing $c_1^L \curvearrowright c_0^L$ and $c_1^L \curvearrowright c_1^L$, we note that no windows exist "after" windows 2 and 6, because their respective sequences end. In this case, the first set to be intersected is null, so the intersection is null. The final self–convolution operation also yields a null set. We now have only one motif for the new round of convolution, c_0^{L+1}. Performing $c_0^{L+1} \curvearrowright c_0^{L+1}$ results in a null set, meaning that there are no more motifs. At this point, we terminate convolution. It is worth noting that c_0^L is returned as a maximal motif because window 4 cannot be extended, but c_1^L is not because all of its instances were convolved in one direction.

Thus, we get different sets of motifs for different similarity and clustering functions. For identity similarity and clique–finding clustering, the final list of motifs is $\{\{\texttt{MOTIF}, \texttt{POTIO}\}, \{\texttt{MOTI}, \texttt{MOTO}\}\}$. For identity similarity and connected components clustering, the final list of motifs is $\{\{\texttt{MOTIF}, \texttt{POTIO}\}, \{\texttt{MOTI}, \texttt{MOTO}, \texttt{POTI}\}\}$. For

vowel/consonant similarity and either clustering method, the final list of motifs is {{MOTIF, MOTOR, POTIO}}.

3.5 Applications

In this section, we demonstrate Gemoda's capability by presenting several sample applications. Specifically, we address motif discovery in amino acid sequences, in nucleotide sequences, and in protein structures.

As discussed previously, the clustering and convolution stages of the Gemoda algorithm are generic — they are independent of the nature of the input data. However, the comparison stage is data–specific. In what follows, we discuss how the comparison stage is changed for each kind of data and outline the types of results Gemoda is capable of finding.

3.5.1 Motif discovery in amino acid sequences

To use Gemoda to find motifs in amino acid sequences, the comparison stage needs to reflect the notion of "similarity" for amino acid sequences. Specifically, we choose a window comparison function $\mathscr{S}$ that returns a sequence alignment score, such as the bit–score from an amino acid scoring matrix (e.g., the popular BLOSUM matrices [66]).

The bit–score is a measure of similarity of two aligned sequences: higher scores indicate greater similarity. For example, the alignment

```
AKTF
:..:
APKF
```

has a bit–score of $4 - 1 - 1 + 6 = 8$ with the BLOSUM–62 matrix, where the AA pair contributes 4 units, the KP pair contributes -1 unit, and so on. There are many different amino acid scoring matrices, each of which measures the similarity between

residues differently [69].

Here, we demonstrate how Gemoda can be used for motif discovery in amino acid sequences by "discovering" known protein domains in the (ppGpp)ase family of enzymes. These eight enzymes catalyze the hydrolysis of guanosine 3',5'–bis(diphosphate) to guanosine 5'–diphosphate (GDP) and are classified by the Enzyme Commission (EC) number 3.1.7.2 [19].

We used Gemoda to identify motifs in these eight (ppGpp)ase enzymes using the BLOSUM–62 scoring matrix as the basis of our similarity function $\mathscr{S}$ and the clique–based clustering function described previously. Specifically, we sought motifs that occurred in all eight sequences, were at least 50 residues long, and had a pairwise bit–score of at least 50 bits over a window of 50 residues.

With these parameters, Gemoda discovers four motifs in this set of eight sequences; the longest motif, with a length of 103 amino acids, is shown in Figure 3-6 as an alignment of the regions that correspond to instances of this motif (see also Figure 3-2).

A comparison with the known protein domains in the NCBI Conserved Domain Database (version 2.02) [109] reveals that this motif captures the RelA_SpoT domain (CDD PSSM–id 15904). The RelA_SpoT domain has an unknown function, but is found in both SpoT proteins (the 3.1.7.2 family) and RelA proteins (EC 2.7.6.5). The RelA enzymes catalyze the synthesis of guanosine 3',5'–bis(diphosphate) from ATP and GTP (or GDP), suggesting that the RelA-SpoT domain is important in both the synthesis and degradation of guanosine 3',5'–bis(diphosphate).

The remaining three motifs are not present in the CDD database. However, further inspection using the tools available from the PFAM database [23] revealed that they composed the left, middle, and right regions of the HD domain [14]. The HD domain is typical of enzymes with phosphohydrolase activity, particularly those enzymes involved in nucleic acid metabolism and signal transduction. In the SpoT enzymes, this domain has a number of insertions and deletions that give rise to gaps such that Gemoda identified and reported individually the left, middle, and right regions of conservation of the HD domain.

In this example, the BLOSUM–62 matrix was chosen as the similarity metric because it is optimized for detecting distant homologs. The Gemoda input parameters $L = 50$ and $g = 50$ were chosen to enforce a one–bit–per–amino–acid score, which should rise above random "noise" since, by design, the expected bit–score for two aligned amino acids is negative for the BLOSUM set of scoring matrices. We found from subsequent experiments that similarities at this threshold do not occur frequently at random (e.g., such similarity in a default BLAST search would have an E–value of about 10^{-5}). Use of more stringent parameters would lead Gemoda to return fewer motifs that are better conserved, and lowering the pairwise similarity threshold or decreasing the minimum length would change the number of motifs returned as well. As with any motif discovery tool, the optimal choice of parameters depends on the problem at hand and may affect the computational difficulty of the problem, in this case via the characteristics of the similarity matrix A. Gemoda's dependence on parameter selection is acceptable, as a slight change in the parameters used for this search does not drastically alter the results that are obtained.

In order to test the sensitivity of these results to noise, we conducted an experiment to determine the degree to which these (ppGpp)ase motifs could be found if obscured by noise caused by adding random spurious sequences to the 8 enzyme sequences. We randomly selected a single sequence from Swiss–Prot (Release 45.0) [20] and included the sequence with the (ppGpp)ase sequences when applying Gemoda with the same choice of parameters. If no "noise" motifs were found (i.e. motifs not from the (ppGpp)ase enzymes), then another random sequence was added to the set of sequences; this step was iterated until at least one "noise" motif was found. We repeated this process ten times and found that motif detection remained unhindered until at least 62 random sequences were added to the initial set of 8 enzymes, a notable detection performance. That is, the target motifs could be detected in an 8–fold majority of spurious sequences given the chosen input parameters.

3.5.2 Identifying co–regulated genes

The discovery of motifs in nucleotide sequences is most commonly used in the search for *cis*–regulatory elements. Previous work in this area is voluminous, encompassing methods ranging from statistical calculations on small enumerated "words" to the use of various pattern discovery tools to locate potential regulatory elements.

An abstraction of the binding site discovery problem has been made and is discussed in greater detail in Chapter 4. Here we present a brief summary of Gemoda's ability to find transcription factor binding sites (*cis*–regulatory elements) in experimentally generated data.

For some regulons in *E. coli* with mild to strong consensus sequences, Gemoda returns results that are similar to or improve upon the results from commonly–used motif discovery tools. For instance, using the set of upstream regions (400 base pairs upstream and 50 base pairs downstream of the translation start site) for the 9 operons believed to be regulated by LexA [145], Gemoda's top–scoring motif was used to generate the sequence logo found in Figure 3-7. This motif closely matches the literature PWM for the LexA binding site and represents 80% of the literature–found binding sites with no false positives. problems.

The parameters used for this search, based on simple heuristics, were $L = 20$, $g = 10$, and $k = 6$. The length was selected based on the knowledge that the DNA–binding domain of LexA is a helix–turn–helix variant, and so it was likely to be a relatively long motif. The similarity threshold was chosen as one–half of L, which we know from the (l,d)–motif problem (Chapter 4 ought to be approximately sufficient to prevent the graph from being too dense (and thus expensive to cluster). The support threshold was chosen to be about two–thirds the total number of sequences, allowing for some noise in the data. Of course, the judicious selection of parameters is an outstanding problem in binding site discovery. It is worth noting that most of these selections were simple and intuitive, with the least intuitive selection (L) happening to be a parameter for which Gemoda was fairly tolerant of slight perturbations.

It should be noted that the above example is not intended to be indicative of every

```
GKIKYKSEQAENYRKLILATAEDPRVILLKLSDRLDNVKTLWVFREEKRKKIAKETMEIY  SPOT_AQUAE
HNKTRSIKEANTISKMFFAMTHDIRIIIIKLADKLHNMTTLSYLPKNRQDRIAKDCLSTY  SPOT_BORBU
KFRDKKEAQAENFRKMIMAMVQDIRVILIKLADRTHNMRTLGSLRPDKRRRIARETLEIY  SPOT_ECOLI
KFRTRQEAQVENFRKMILAMTRDIRVVLIKLADRTHNMRTLGSLRPDKRRRIAKETLEIY  SPOT_HAEIN
LKNKKENLNLKSFVNIAINSQQEINVMVLKLADRLDNIASIEFLPIEKQKVIAKETLELY  SPOT_MYCGE
LNRKKEDLNLKSLVNIAMSSQQEVNALVLKLADRLDNISSIEFLAVEKQKIIAKETLELY  SPOT_MYCPN
AKENRTQIKAQYLRKLYLSMAKDIRVIIVKLADRLHNLKTIGYLKPERQQIIARESLEIY  SPOT_SPICI
NFSSTTEHQAENFRRMFLAMAKDIRVIVVKLADRLHNMRTLDALSPEKQRRIARETKDIF  SPOT_SYNY3

APLAHRLGVWSIKNELEDWAFKYLYPEEYEKVRNFVKESRKNLEE  SPOT_AQUAE
VPIAERLGISSLKTYLEDLSFKHLYPKDYKEIKNFLSETKIEREK  SPOT_BORBU
SPLAHRLGIHHIKTELEELGFEALYPNRYRVIKEVVKAARGNRKE  SPOT_ECOLI
CPLAHRLGIEHIKNELEDLSFQAMHPHRYEVLKKLVDVARSNRQD  SPOT_HAEIN
AKIAGRIGMYPVKTKLADLSFKVLDLKNYDNTLSKINKQKVFYDN  SPOT_MYCGE
AKIAGRIGMYPVKTQLADLSFKVLDPKNFNNTLSKINQQKVFYDN  SPOT_MYCPN
SAIAHRLGMKAVKQEIEDISFKIINPVQYNKIVSLLESSNKEREN  SPOT_SPICI
APLANRLGIWRFKWELEDLSFKYLEPDSYRKIQSLVVEKRGDRES  SPOT_SYNY3
```

Figure 3-6: The RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences.

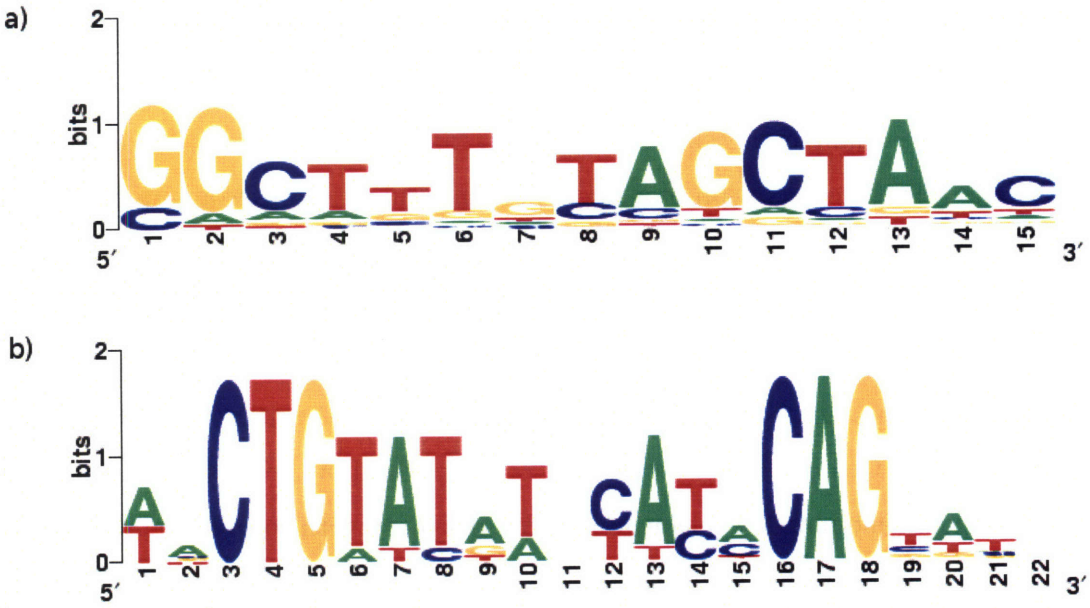


Figure 3-7: The sequence logo for a) the motif implanted in each sequence for the (l,d)–motif problem (see Chapter 4 and b) the LexA binding site motif generated from the highest–scoring motif returned by Gemoda.

motif discovery problem in DNA, as noise, dataset quality, and other factors all affect these problems. Parameter selection has always been and remains an important consideration for this kind of discovery problem; generally, the identification of optimal values may depend on a fundamental understanding of the problem domain. Problems that plague all motif discovery tools also have an impact on Gemoda, including the lack of a good "gold standard" for accuracy and the difficulty in finding motifs that a "gold standard" defines as being extremely weak. Despite these problems, our preliminary examples have shown a great deal of promise.

3.5.3 Motif discovery in protein structures

The detection of 3–dimensional motifs in sets of protein structures is another problem type that Gemoda can address. Often, homologs that are related through a distant lineage show little to no sequence similarity, particularly at the nucleotide level [50]. However, these homologs frequently show conserved tertiary structures [42], making motif discovery in protein structures often revealing in situations where there appears to be no similarity at a sequence level.

There are a number of well–developed tools for the pair–wise comparison of protein structures or the comparison of a single protein structure to precomputed structural motifs; these have been reviewed elsewhere [50]. Some of the more popular tools include SSAP [127], VAST [107], Dali [76], Mammoth [128], and FoldMiner [149]. The Gemoda algorithm, when used for structural motif discovery, is most similar to the Sarf algorithm [2, 4, 3] and, to a lesser degree, algorithms by [80] and [86]. Conceptually, Gemoda could be thought of as a hybrid of the Sarf and TEIRESIAS algorithms, combining 3–D elementary motif discovery with convolution. To the best of our knowledge, Gemoda is the only tool that can compare an arbitrary number of protein structures simultaneously and produce an exhaustive set of maximal motifs.

To discover motifs in protein structures, Gemoda compares L–residue windows of the proteins' alpha–carbon trace using the minimized RMSD similarity metric (one of many possible metrics for comparing protein sub–structures [97]). Here we use "minimized" to indicate that the protein structures are optimally super–imposed

via rigid–body rotation and translation [78, 15]; occasionally this term is implicit. (When using the Gemoda algorithm for analyzing other types of real–valued data, this super–imposition is occasionally not desired.) Other possible metrics include the unit–RMSD, which is used both by Mammoth and by Krasnogor's "universal similarity metric" [98].

To facilitate the use of the software, Protein Data Bank (PDB) [25] can be used as the initial data input file for using Gemoda. Scripts are supplied to transform these structures into a series of (x, y, z) coordinates representing the positions of the alpha–carbons along the backbone of the structure: the alpha–carbon trace.

Using the clique–finding clustering algorithm, Gemoda finds motifs that are sets of alpha–carbon traces (in a set of protein structures) that can be super–imposed with an RMSD less than g $\mathring{A}$ over each window of L residues on a pair–wise basis. Similar to the amino acid and nucleotide applications of Gemoda, these structural motifs are maximal in both length and support.

As an example, we demonstrate how the Gemoda algorithm can be used for structural motif discovery by "discovering" known 3–dimensional motifs that are conserved between distant homologs. We attempted to detect the structural homology between the human galactose-1-phosphate uridylyltransferase (PDB id 1HXQ) [177] and fragile histidine triad proteins (PDB id 3FIT) [101], originally reported elsewhere [77]. Using Gemoda, we looked for motifs of at least 30 residues, occurring in at least three chains, that had a pairwise RMSD of 1.5 $\mathring{A}$ or less (based on superposition of the alpha–carbon backbone) over each window of 30 residues.

This search returns 4 motifs, the longest of which is 66 residues (see Figure 3-8). This motif has one embedding in the 3FIT protein and two, in different chains, in the 1HXQ protein. As shown in the figure, the motif is an alpha helix followed by a beta sheet.

Thus, we see that Gemoda is generic in that it can even be applied to real–valued data. Other examples of pattern discovery in such real–valued data include time–series stock price data and gas chromatography–mass spectrometry (GC–MS) data. Each of these data sets needs only to be expressed as a series of arbitrary–dimensional

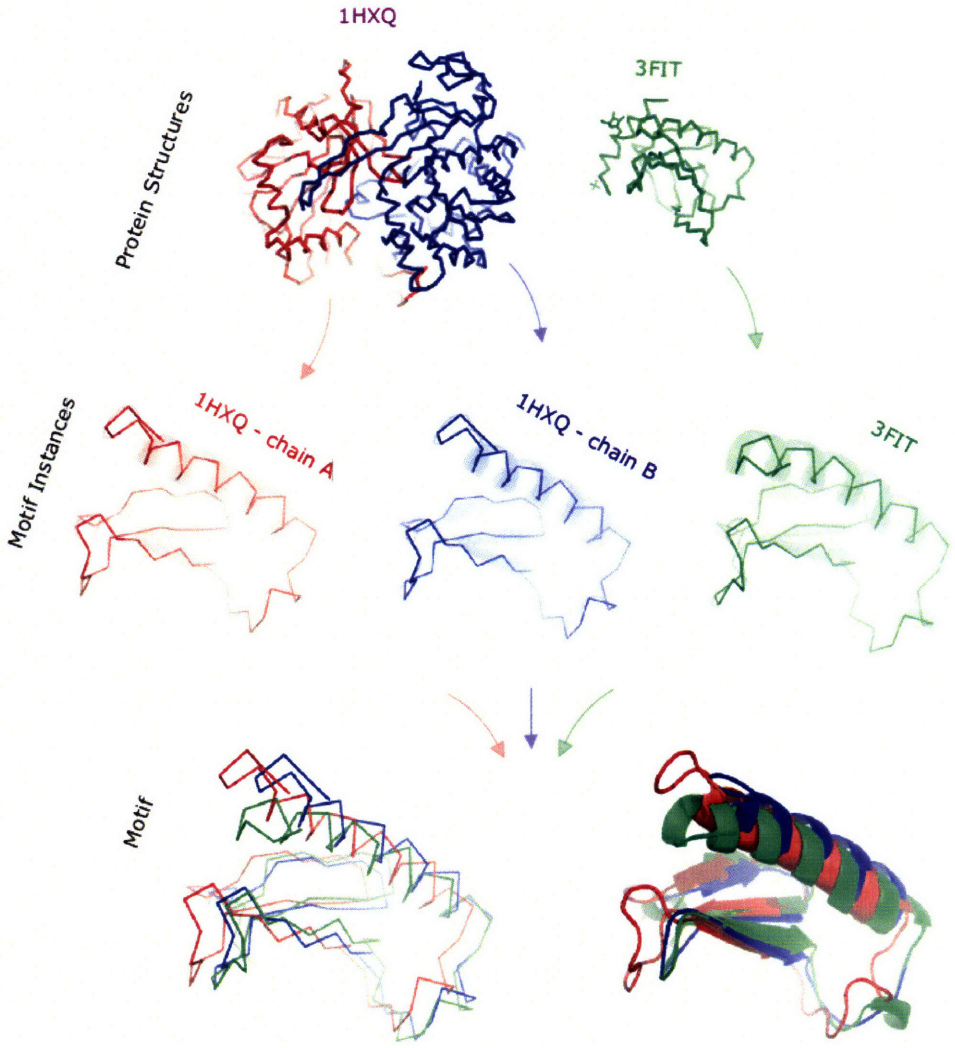


Figure 3-8: A motif showing structural conservation between the human galactose-1-phosphate uridylyltransferase and fragile histidine triad proteins originally reported by [77]. All subunits are treated as individual proteins, and the resulting protein chains are then compared one window at a time to identify conserved secondary structure. The motif, as shown here, was discovered using the Gemoda algorithm along with three other, smaller, structural motifs that are highly conserved between the two proteins. Notably, the proteins show little sequence similarity over the region displayed in the structural motif above. Graphics created using PyMol (DeLano Scientific, San Carlos, CA, USA).

points to be suitable for analysis by Gemoda. In addition, one must use (or define) an appropriate similarity function; for instance, while a translational (but not rotational) RMSD may be appropriate for stock data, a more appropriate similarity metric must be used for GC–MS data.

3.6 Discussion

Gemoda makes four contributions. First, the algorithm is generic in that it is equally applicable to any variety of sequential data. Second, Gemoda allows arbitrary similarity metrics. In the examples shown here, we chose relatively simple metrics (scoring matrices and RMSD–base metrics); however, similarity metrics can be easily changed or added. For example, in the case of amino acid sequences, one can easily define hybrid metrics incorporating primary, secondary, and tertiary structure features. In the case of nucleotide sequences, the metric may be changed to incorporate methylation information. The third contribution is that Gemoda returns motifs that are not tied to any particular motif representation. In the case of amino acid sequence motifs, it is easy to model Gemoda's motifs using regular expressions, hidden Markov models, or position–specific scoring matrices. Finally, when used with the clique–finding clustering algorithm, Gemoda returns an exhaustive set of maximal motifs. To the best of our knowledge, Gemoda is the only motif discovery algorithm incorporating the above features.

As mentioned in the introduction, Gemoda integrates the best characteristics from a number of previously published motif and association discovery algorithms. For specific problems, Gemoda's performance can be improved further, though at the expense of generality. For example, a window sampling approach such as that used by Blast [11] would be useful in applications where speed is more important than completeness of results. For protein structure comparisons Gemoda could also be altered to use contact maps like those used by Dali [76]. The convolution stage could also be made faster by using heuristic, non–exhaustive convolution methods.

Furthermore, the core Gemoda algorithm could be modified to produce gapped

motifs. As currently formulated, Gemoda is not able to automatically identify motifs with large or variable–size gaps; if a gap causes a motif to fail to meet the similarity threshold when it is extended, then it is not extended. One may note that this is the case for many motif discovery algorithms, particularly ones that are not specialized to solve the gapped motif problem. One alternative may be to alter the convolution step to allow for gapped motifs. Another option is to analyze the current output of Gemoda for such motifs: they may occur as multiple distinct maximal motifs. In this case, *post hoc* analysis by the user will still allow the discovery of gapped motifs, either by visualizing and identifying their occurrences or by post–processing the motifs in a process analogous to Gemoda's convolution.

Thus, we see that Gemoda's generic nature makes it readily applicable for many problems. In the protein sequence application, Gemoda's exhaustive search using a scoring matrix as a similarity metric identified multiple motifs. It provided an accurate representation of these domains in as much as an eight–fold excess of spurious sequences. In the DNA motif discovery application, Gemoda identified an otherwise unintentional result in a synthetic dataset and satisfactorily described a motif embedded in a genomic dataset. In the protein structure application, Gemoda demonstrated that it can compare multiple arbitrary–dimensional structures simultaneously and return results previously shown in the literature. Gemoda can also be directly applied to other diverse types of sequential datasets, or it can be extended to address problems not yet considered.

Chapter 4

Gemoda for DNA sequences: solving the (l,d)–motif discovery problem

4.1 Overview

As discussed in Chapter 3, Gemoda is capable of finding *cis*–regulatory sites in DNA where transcription factors may bind given a set of sequences upstream of genes believed to be coregulated. The example given in Chapter 3 used freely available experimental data. While the utility of this algorithm on such realistic data is certainly useful, it is also desirable to know whether Gemoda is conceptually able to handle a well–curated gold–standard dataset and find all known motifs. While gold–standard experimental data would be an ideal test for this goal, the reality of the situation is that even the available well–curated data available may not necessarily be viewed as "gold–standard". To that end, a useful test of Gemoda would be with completely synthetic data where all binding sites are known because they have been manually inserted.

The (l,d)–motif challenge problem, as introduced by Pevzner and Sze [132], looks to provide such a test as a mathematical abstraction of the DNA functional site dis-

covery task. In this chapter I will explain what the (l,d)–motif problem is, solve it using Gemoda, and then expand it to more accurately model the experimental difficulties that confound the creation of sets of coregulated genes. According to the theory explained earlier and based on the problem statement of the (l,d)–motif problem, Gemoda is guaranteed to find all (l,d)–motifs in a set of input sequences with unbounded support and length. We demonstrate the performance of the algorithm on publicly available datasets and show that Gemoda indeed deterministically enumerates the optimal motifs.

Much of the research and discussion in this chapter is drawn from the publication that describes the application of Gemoda to the (l,d)–motif discovery problem:

- Styczynski MP, Jensen KL, Rigoutsos I, and Stephanopoulos GN. "An extension and novel solution to the (l,d)–motif challenge problem." *Genome Informatics* 2004: 21 – 28 (2006).

Again, this chapter will frequently use "we" to refer to the collective authors of the manuscript and their contributions to the work.

4.2 Introduction

In 2000, Pevzner and Sze [132] noted that despite significant advances in pattern discovery, there were still gaping holes in our ability to identify and enumerate frequent patterns in biological sequences. Experimental noise and error were not the only significant issues, as the community was still incapable of solving certain problems with purely synthetic data and no worry of experimental or gross error. One such problem, defined below, was the (l,d)–motif challenge problem; it exposed the fact that certain motifs, despite having a strong consensus and being rather unlikely to occur at random in independent and identically distributed (i.i.d.) sequences, are extremely hard for most motif discovery algorithms to locate. The reason that these motifs are hard to locate is that even though they may deviate very little from a consensus sequence, their pairwise deviation tends to be rather large. Other false pairwise similarities are thus extremely likely to occur at random elsewhere in the dataset, and

this random noise obscures the true motif's signal. Pevzner and Sze [132] presented two algorithms that looked towards solving this problem; Buhler and Tompa [34] followed suit by presenting a more effective algorithm. However, the problem is still not completely solved per se; difficulties exist in obtaining the correctly refined motifs and instances even for this simplified model of biology. In addition, though existing algorithms move towards solving this simplified problem, they are not nearly as helpful in addressing the biological realities that computational biologists face.

The (l,d)–motif problem was designed to model a biological phenomenon that could not be addressed by previous computational methods. To that end, the abstraction served its purpose well; however, there is a fundamental disconnect between the system that the challenge problem models and the biological realities scientists face on a daily basis. It is this disconnect that has kept the motif challenge problem merely a theoretical, academic exercise up until now. It is important to create a connection between the abstracted problems we strive to solve and the systems to which those solution algorithms could ultimately be applied. While the (l,d)–motif problem described below is justifiably inspired by biology, it deviates from research conditions in a few key ways that ought to be modified to more accurately reflect experimental conditions and prompt the creation of truly useful search tools. This paper aims to point out those differences and reframe the problem in a manner more accurately reflecting the biological system being modeled.

Our objective, then, is to discover DNA sequence motifs in a way that requires as little *a priori* knowledge as possible. The challenge problem presented in previous works is a simplified model of this more general problem. We contend that Gemoda is of significant use in solving the open (l,d)–motif problem in a deterministic but computationally tractable nature.

In what follows, we will give a formal statement of the motif discovery problem as it applies to our algorithm. We will then detail our application to the (l,d)–motif problem and explain how our approach can make it a more biologically meaningful abstraction of the binding site discovery problem.

4.3 The expanded (l,d)–motif challenge problem

The original (l,d)–motif problem [132] can be paraphrased as follows:

> Within a set of random DNA sequences with i.i.d nucleotides, a parent motif of length l is embedded in each sequence in a random location. Each time the motif is embedded, it is mutated in d locations. The (l,d)–motif problem is to recover the locations of the embeddings, knowing only the parameters l and d and that each sequence contains exactly one instance of the motif.

At first, this seems to be a reasonable simplification of the phenomenon of binding sites and other functional sites in DNA. It is not uncommon to have some ancestral sequence from which each motif occurrence is some short evolutionary distance away. This model accurately captures the difference between instance–instance similarity and instance–ancestor similarity. That is, even though a motif instance may be a very short distance from its ancestor (say, four mutations out of fifteen bases), any two instances of the motif may be significantly different from each other (eight mutations out of fifteen bases). This low degree of instance–instance similarity can occur rather frequently in random i.i.d. nucleotide sequences, thus obscuring the true evolutionary relationship of the motif instances (the signal) with purely random relationships of background nucleotides (the noise) [34, 33].

As discussed by Buhler and Tompa [34], local search methods (such as the common ones mentioned before) using typical initialization strategies encounter an insurmountable amount of noise when searching for some sparse motifs described by the (l,d)–motif problem. We would ideally like to be able to recover such motifs, since they are expected to occur by chance in every sequence with rather low probability (approximately 10^{-7}) [34, 33].

In a more realistic scenario, a researcher may not know the size l of the motif *a priori*. Instead, it is more likely that she would know the evolutionary distance between motif instances, i.e. the rate of mutation d/l. It is also unrealistic to mutate the embedded motif *exactly* d times; rather, the researcher is more likely to be interested

in motifs that are d or fewer mutations away from each other. That is, in a real–world scenario, we would more likely have a reasonable estimate of the upper limit d/l of the mutation distance between embedded motifs. There may also be multiple, different motifs in the dataset. Finally, as experimental data are commonly rife with noise, it is likely that some of the sequences may be false–positive candidates for the motif; that is, some sequences may contain no motifs at all.

With these issues in mind, we define an extended (l,d)–motif problem as follows:

> Within a set of random DNA sequences with i.i.d. nucleotides, a parent motif of length $\geq L$ is embedded zero or more times in each sequence in a random location, such that the motif has been embedded a total of k times in the data set. Also, each time the motif is embedded it is mutated such that there are no more than d mutations over any window of l nucleotides (that is, the rate of mutation is d/l). This process is repeated for any number of parent motifs, each with the same l and d, but possibly different L. The extended (l,d)–motif problem is to recover the locations of the embeddings for every parent motif without any *a priori* knowledge of where they might be, but only knowing the parameters l and d.

We will refer to this formulation as the "extended" (l,d)–motif problem and the previous formulation as the "restricted" (l,d)–motif problem. In what follows, we detail an algorithm for solving both the extended and restricted (l,d)–motif problems.

4.4 Preliminary definitions and nomenclature

All of the same nomenclature from Chapter 3 will be utilized in this chapter. In addition, we will call the Hamming distance function $\mathscr{H}$, where $\mathscr{H}$ takes two windows of size l from our sequence set, and returns a real–valued number equal to the number of characters that differ between the two windows.

Using these definitions, we note that the extended (l,d)–motif problem can be solved another way. Given a sequence set and the function $\mathscr{H}$, finding all maximal

(l,d)–motifs in the sequence set will identify all possible planted motifs and enumerate the locations of all instances of those motifs.

4.5 Applying Gemoda to the extended (l,d)–motif problem

The input to Gemoda is a set of nucleotide sequences $S = \{s_1, s_2, \ldots, s_n\}$, where sequence i has length W_i. So, for example, the j^{th} member of the i^{th} sequence is denoted by $s_{i,j}$ and is one of the characters `A,T,G,C`.

The comparison phase proceeds as defined previously; we will briefly review what that entails for this problem. Typically, a user is interested in motifs that have a minimal length (i.e., motifs of length 1 are not interesting, whereas motifs of length 10 might be depending on the context). We denote this user–supplied length as L and we define a symmetric matrix A of size $N = \sum_{i=1}^{n}(W_i - L + 1)$. That is, A is a matrix with one index for each window of size L in our entire sequence set. The 10^{th} window of size L in the 5^{th} sequence would be expressed as $s_{5,10:10+L-1}$, where "$10 : 10 + L - 1$" denotes "position 10 through position $10 + L - 1$, inclusive". To keep track of which window corresponds to each index in A, we define the function $\mathscr{M}(s_{i,j:j+L-1}) \mapsto q \in [1, N]$. (For the sake of simplicity in later use, we define $(s_{i,j}+1)$ to be $s_{i,j+1}$, unless $s_{i,j+1}$ does not exist, in which case $(s_{i,j}+1)$ is undefined.) Similarly, $\mathscr{M}^{-1}(q) \mapsto (s_{i,j:j+L-1})$ such that $i \in [1, n]$ and $j \in [1, W_i - L + 1]$.

Thus, $A_{i,j}$ is equal to $\mathscr{H}(\mathscr{M}^{-1}(i), \mathscr{M}^{-1}(j))$.

At this point we have a matrix A that contains the pairwise distances between each window in the dataset. One could also say that that A is simply a distance matrix that could be used to plot the windows in some high–dimensional space. In that sense, we are interested in clustering these windows together to form "elementary motifs".

The clustering and convolution phases proceed as defined previously.

4.6 Solving the restricted (l,d)–motif problem

The input set for the (l,d)–motif problem is any arbitrary set of n sequences, each with length W_i nucleotides. Most bioinformatics literature treatments use $W_i = 600$ and $n = 20$. Different versions of this problem have been discussed at length; the most commonly discussed is the (15,4) problem, while the (14,4) and other associated, more difficult problems are also addressed in the literature.

It has been shown before that the most commonly used motif discovery algorithms, including CONSENSUS [72], Gibbs sampling [100], and MEME [17], are unable to solve the restricted (15,4) problem. Algorithms that are capable of solving the restricted (15,4) problem have been presented in the literature. While some of these, including Winnower and SP-STAR [132], are unable to solve the more complicated (14,4) problems, others are able to address this and other, more difficult, problems with some degree of accuracy. These latter algorithms usually leave the deterministic realm, though, and rely on probabilistic methods to find the planted motifs.

On the other hand, Gemoda allows for exhaustive, deterministic solution of these problems. The (l,d)–motif problem solved by the above–mentioned tools is a degenerate case of the extended problem that our algorithm was designed to solve. Thus, Gemoda is not optimally tuned for solving the restricted (l,d)–motif problem in the least amount of time. Nonetheless, solving a range of the restricted (l,d)–motif problems is still a valuable check on the utility of our tool to make sure it can solve at least some of them in a reasonable amount of time. In addition, our exhaustive search allows for one to see how many other false signals are in the data. This can facilitate the assessment of statistical significance of results, certainly an important step in analyzing any proposed signal.

4.6.1 Solution Method

Our approach requires three user input parameters: l, g, and k. l is the minimum motif size and the size of the sliding window used for judging similarity between two sequences. g is the similarity threshold for any two windows to be deemed instances

of the same motif; in this case, if two windows of length 10 are a Hamming distance of 2 away from each other, g would need to be 8 or less for the windows to be in the same motif. Finally, k is the support, or minimum number of motif occurrences required to report the motif to the user.

It is obvious that any two motifs of length l each being mutated d times from an ancestral sequence can differ at most at $2d$ locations. Thus, at least $(l-2d)$ locations must be preserved in the motif. This observation lays the foundation for discovery of the hidden motifs. Gemoda is run with parameters $l = 15$, $g = 7$, and $k = 20$ for the (15,4) problem. The discovery of the motif is then a straightforward combinatorial problem with deterministic discovery of the solution.

It is important to note, however, that our method will solve and return a superset of the restricted (l,d)–motif problem. That is, any group of d–mutants from a common ancestor can be described as having $(l - 2d)$ identical bases, but not all groups of sequences with $(l - 2d)$ identical bases can be used to synthesize an ancestor from which all group members deviate $\leq d$ bases. When there are a large number of "signal" motif members, there is usually sufficient overall deviation to prevent a $\geq d$–mutant from joining a motif group. However, at smaller support k, it is more likely to find motif instances that violate the d–mutant constraint. It is not desirable to immediately remove motifs with such members from the output, as they do still meet the constraints imposed by our parameter values; rather, we can use a simple post-processing method to note which motifs have readily obvious ancestors and thus are the most likely candidate signals.

4.6.2 Discussion

A few interesting observations can be made regarding the complexity of the algorithm and the quality of its solutions. First of all, the time to solution is not affected directly by the length of the motif to be discovered as in many other exhaustive methods. Rather, it is the sparseness or subtlety of the motif (or more accurately, the probability of the pairwise motif similarity occurring randomly) that has the most profound impact on the complexity of the algorithm. The most computationally

expensive step is the clique-finding function, which increases in computation time with the number of edges (NP–complexity at worst, though on average much better). For varying l and d, as two l-mers sampled randomly from the background are more likely to meet the threshold of similarity defined by l and d, there will be more false edges (similarities) in the graph, and thus the clustering algorithm will take longer. Motifs of widely different length may be (approximately) equally likely in the background distribution if d is set to a certain value for each. In this case, it would take almost exactly the same amount of time to find both motifs in the same input set. Of course, the size of the data set also has a significant impact on computation time, as for any algorithm; a larger input set causes more false occurrences of a potential motif, and the resulting distance matrix needs more time to be explored by our clique-finding algorithm. Figure 4-2 indicates the order of complexity of computational time for this problem; since the pairwise similarities that Gemoda is searching for occur at random so frequently within the sequence datasets, the graph is rather dense, making the increase in computation time approximately exponential, especially for the more difficult parameter values plotted. This is as one may expect from an NP–complete problem. Thus, as indicated earlier, there are certainly limits as to what size datasets Gemoda can handle — but the size of datasets that are of interest to at least some biologists and bioinformaticians may lie within the realm of those that are computationally tractable for our exhaustive approach.

Also, our method does not preclude discovery of more than one instance of a motif in any given sequence. Much like the re–framing of the (l,d)–motif problem presented above, this is more reflective of what one expects may happen in a real biological system: motifs of biological significance may occur more than once in a biosequence, and it behooves us to be able to discover all occurrences. In fact, in the original dataset for the (15,4)–motif problem used by Pevzner and Sze [132], there is actually an additional instance of the original motif that occurred completely by chance; this instance was discovered in our solution of the problem (see Figure 4-1). In addition, by virtue of its exhaustiveness, Gemoda is capable of finding multiple unique motifs that may exist in one dataset; other approaches are limited to finding

only the strongest motifs.

Finally, it is important to note the absolute accuracy of our results. In previous papers presenting algorithms to solve the (l,d)–motif problem, a metric called the performance coefficient is used to gauge the accuracy of the algorithms. This is defined as $\frac{K \cap P}{K \cup P}$, where K is the set of $l * s$ nucleotides representing the s motif instances each of length l and P is the set of $l * s$ nucleotides representing the s proposed motif instances of length l. Coefficients above .75 are usually deemed acceptable for these algorithms. Improved algorithms return results with coefficients of about 0.9 or 0.95. Examples of the performance of other algorithms are presented in Table 4.1. Clearly, Gemoda returns all coefficients of 1; that is, it will return the exact location of all motif occurrences. This is a notable improvement over other algorithms that may return approximate motif locations that then need to be verified and slightly adjusted or optimized by hand. In fact, in any given run of PROJECTION (the most accurate of the algorithms in Table 4.1), one will usually find that one or two (or even more) of the returned motif instances are not just imperfectly located, but are false positives.

Table 4.1: Performance on a range of (l,d)–motif problems with synthetic data. Data from other algorithms are from Buhler and Tompa [33]. GibbsDNA, WINNOWER, and SP-STAR are averaged over eight random instances, while PROJECTION is averaged over 100 random instances. Computation times for Gemoda are averaged over three random instances.

l	d	GibbsDNA	WINNOWER	SP-STAR	PROJECTION	Gemoda	Time
10	2	0.20	0.78	0.56	0.80	1.00	8 min
11	2	0.68	0.90	0.84	0.94	1.00	$<$ 1 min
12	3	0.03	0.75	0.33	0.77	1.00	10.5 h
13	3	0.60	0.92	0.92	0.94	1.00	10 min
14	4	0.02	0.02	0.20	0.71	1.00	$>$ 3 months
15	4	0.19	0.92	0.73	0.93	1.00	6 h
17	5	0.28	0.03	0.69	0.93	1.00	3 weeks

The computation time of our tool becomes unacceptable as the motifs become degraded beyond the (15,4) problem. This is to be expected for a deterministic algorithm as the probability of the signal reaches a level that causes many pairwise

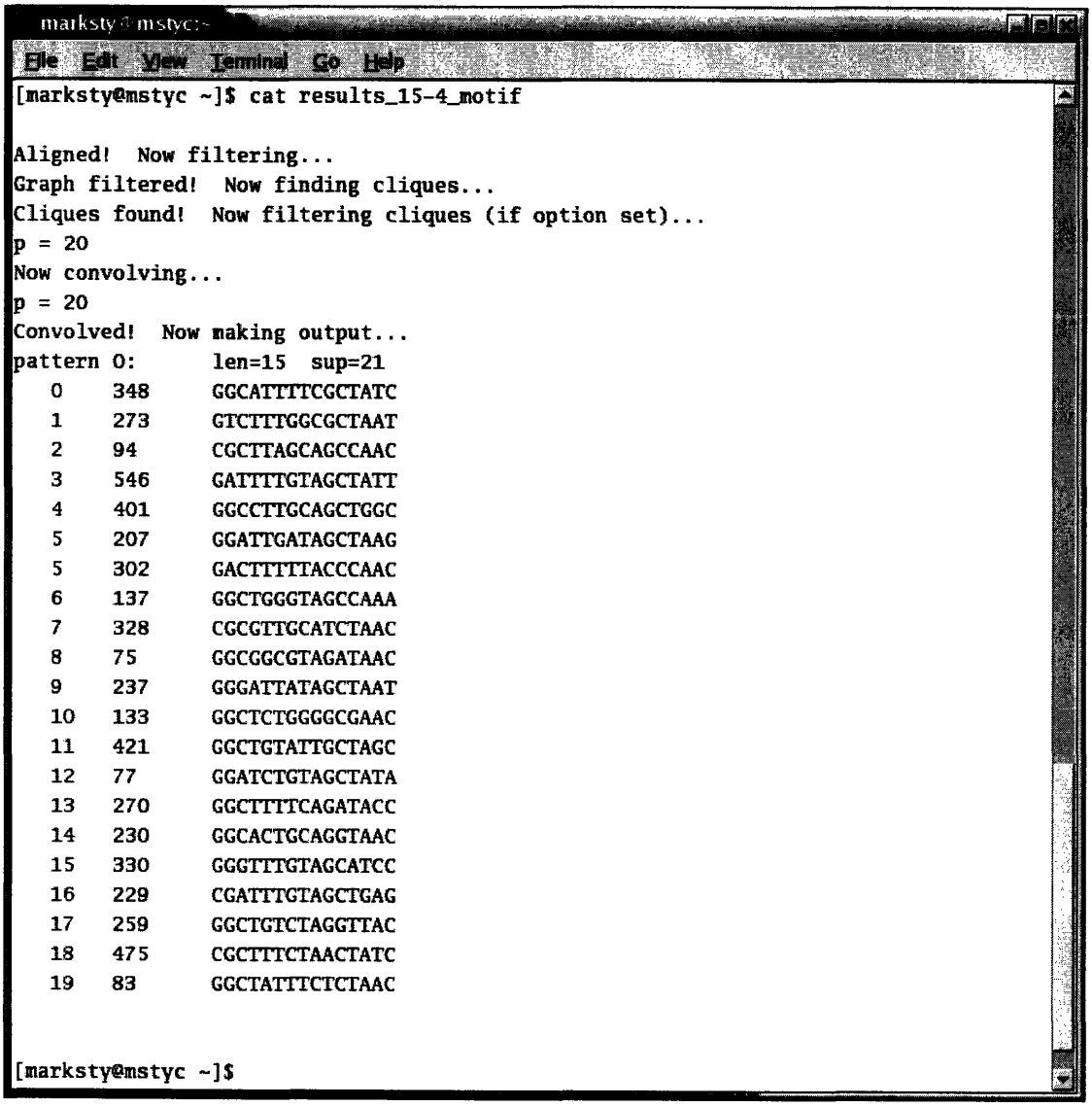


Figure 4-1: A screenshot indicating Gemoda's output for the (15,4)–motif problem. Note that all occurrences of the motif are returned to the user, and all have the same common ancestor. Also note that while there are only twenty sequences (numbered starting from zero), twenty–one occurrences of the motif have actually been found. This is not an intentional implantation of the (15,4)–motif, but it is indeed only four mutations from the parent motif. This is an interesting example of the potential of a provably exhaustive search of the sequence space.

similarities to occur by chance. Since our strategy is generalized and exhaustive, we expect the computation times to be suboptimal. Beyond this table, one would benefit from other probabilistic or heuristic algorithms in order to solve the more difficult (l,d)–motif problems in an acceptable period of time. Fortunately, it seems to not be a too frequent occurrence to search for a (18,6)–motif in each of 20 biological sequences, so Gemoda should be of significant utility for common applications.

We also note that the complexity of the restricted problem seems to be NP. Using the (15,4) problem and the (11,2) problem for analysis, we analyzed the impact of increasing the dataset size on the computational time required to solve the problem. As Figure 4-2 demonstrates, the (15,4) problem clearly shows a log dependence of computation time on sequence size, as one would expect from an NP–complete problem. The (11,2) problem seems to escalate more slowly in computational time (even on a log scale), but it is still likely NP in complexity.

4.7 Solving the extended problem

Of course, in a real biological problem, one does not have nearly the same certainty in the contents of each biosequence as is allowed by the (l,d)–motif problem. This becomes evident upon analyzing the situations that the (l,d)–motif problem is meant to analyze, the most salient of which being the discovery of transcription factor binding sites. In order to come up with the candidate coregulated sequences, the results of laboratory experiments are analyzed to find which genes are sufficiently coexpressed. However, much of this data is prone to noise. Some genes may not be coexpressed, though they may seem to be due to some experimental aberration. Of those that are actually coexpressed, they may or may not be coregulated by the same transcription factor; it is a distinct possibility (and quite frequently a reality) that genes appearing to be coexpressed are not bound by any common factor. The same analysis follows for other situations for which the (l,d)–motif problem is an otherwise reasonable approximation: experimental noise prevents certainty that all input sequences are truly.

Other methods meant to be robust enough to solve the restricted (l,d)–motif prob-

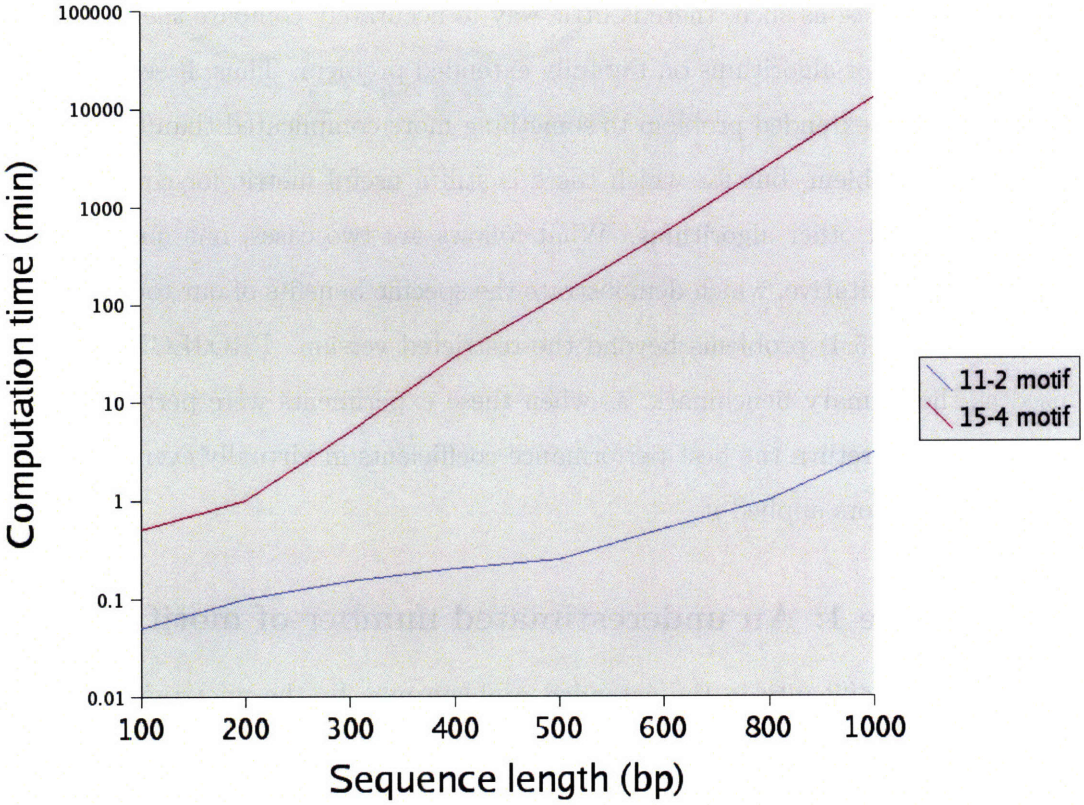


Figure 4-2: Computational complexity of the (l,d)–motif problem. Two different motifs were used: one (15,4) and one (11,2). The size of each of the 20 input sequences was varied to see the effect of dataset size on computational cost. Both problems exhibited the behavior one would expect from an NP–complete problem, indicating that the density of graphs caused by these problem definitions makes our clique–finding step's computational complexity NP.

lem will lose significant advantage in this more realistic, extended set of circumstances. Gemoda was designed specifically to deal with the issues addressed by the extended challenge problem. It discovers, in a provably exhaustive and deterministic fashion, all motifs described in the extended problem definition. Other algorithms discussed previously in this paper are just not constructed to deal with such uncertainty in motif characteristics; as such, there is little way to accurately compare the performance of ours and other algorithms on the fully extended problem. Thus, it seems intuitive to simplify the extended problem to something more complicated than the restricted (l,d)–motif problem, but for which there is still a useful metric for comparison between ours and other algorithms. What follows are two cases, one qualitative and the other quantitative, which demonstrate the specific benefits of our tool for pattern discovery on (15,4) problems beyond the restricted version. PROJECTION will be used as the primary benchmark, as when these experiments were performed it had been shown to return the best performance coefficients in virtually every problem to which the authors applied it.

4.7.1 Case 1: An underestimated number of motif instances

One source of difficulty in the extended problem may be the uncertainty as to the exact number of motif instances. For this case, we still restrict ourselves to windows of size l with d mutations from a consensus sequence. However, we allow for uncertainty in the number of motif instances. For this case study, we instruct algorithms to find motifs with instances in at least 15 sequences when in fact there is an instance in every sequence. If an algorithm such as WINNOWER were to search for cliques across 15 sequences when in fact all 20 sequences had a motif instance (as in this test), it would have a final graph with much more than the single signal that it usually hopes to obtain. PROJECTION's attempts to find 15 instances when 20 actually occur are similarly problem-ridden, returning different candidate motifs on different runs.

In empirical testing using the original data of Pevzner and Sze [**132**], our tool found that there are 379 different maximal motifs with support of at least 15, one of which was the optimal motif of support 21 and was returned as the most signifi-

cant motif. As one expects from a probabilistic algorithm, PROJECTION returned different candidate motifs on different runs. These results would sometimes have significant overlap with one of the 379 motifs, though at other times would have very little overlap. Most disturbingly, all of these proposed motifs would have approximately the same score, thus making it difficult to discern a truly useful motif from an artifact of background noise.

One workaround for the problem presented by this case study is to run the existing algorithms across a range of potential support values, with the most significant motif at the highest support being the motif returned overall. Obviously, this begins to erode away at the sole advantage that these other algorithms have over our algorithm: computation time. Multiple runs of PROJECTION begin to approach the computation time we use, and without any of our computational guarantees. Furthermore, the results of such an iterative approach would be difficult to analyze; the scores of motifs returned by PROJECTION, for example, are frequently quite similar, so it may be rather difficult to discern which motif is truly the "best".

Thus, it seems that in the case of underestimating the number of motif instances, a cursory qualitative analysis demonstrates our tool's potential utility. A further simplification of the problem will facilitate a quantitative study and demonstrate that our algorithm is truly the only one currently able to address the extended problem well.

4.7.2 Case 2: Zero-or-one motif instances

In this next case, we analyze the impact of there being zero or one motif instances in each sequence. To implement this simplification, we instruct each algorithm to find the exactly 15 motif instances that are implanted across 20 sequences. This makes the problem astonishingly similar to the (l,d)–motif problem, with the exception that not every sequence contains a motif instance. This problem setup is thus significantly more realistic, as one does not expect every sequence to have a motif occurrence in every pattern discovery problem. Of course, this is still a simplification of reality, as one would not expect to know the exact number of motif instances. However, not

even this gross simplification can salvage the efficacy of existing algorithms for the discovery of such subtle motifs.

Three different instantiation examples were used to assess the effectiveness of different algorithms. In one, fifteen sequences from the original Pevzner and Sze dataset [132] were supplemented with five sequences of background distribution (all bases equally probable). We ran our algorithm to analyze the contents of this new input set, and PROJECTION was run 100 times to find the average performance coefficient for this specific example. Similar analysis was performed for a new synthetic dataset in the second example. For the final example, PROJECTION was run 100 times with 100 unique input sets, and its average performance coefficient was returned. Note that our performance coefficient will be 1, so there is little need to run our tool on each of these 100 datasets except to gauge its computational expense. In each of five sample datasets, our algorithm took approximately 13 hours to return its solution set. Each set of 100 runs of PROJECTION took approximately 17 hours.

Table 4.2: Performance on an extended challenge problem, case 2. (15,4) motif instances were planted in 15 of 20 sequences. PROJECTION was used with a minimum bucket size of two, which returns more accurate results at the cost of computation time.

Example	Performance Coefficients		
	PROJECTION average	PROJECTION max	Proposed algorithm
1	.06	.43	1
2	.004	.03	1
3	.08	.76	n/a

In the first instantiation example, our tool found that there were 18 unique motifs that met its $g = 7$ constraint over a 15-base window. However, only one had an obvious common ancestor from which all could be obtained by only four mutations. (Since we did not exhaustively search to construct ancestors, it is technically possible, though unlikely, that a motif had a non–obvious ancestor. For this reason we only "rule in" as more likely motifs with obvious common ancestors rather than "rule out" others.) This single motif corresponded with the motif that was originally implanted

in the data, plus two additional instances that occurred purely by chance and were only four mutations from the common ancestor. Perhaps even more interesting is that there was a non–implanted, purely random motif that occurred with a support of 16. Though each pair of instances had similarity of ≥ 7, there was no obvious ancestor. In the second example, our tool found that there was exactly one motif that occurred at least 15 times in the input set; this was the exact motif implanted synthetically.

The results of PROJECTION can be seen in Table 4.2. Clearly, PROJECTION is poorly suited to solving this more realistic version of the challenge problem, while our algorithm returns accurate results, as promised. Even when run 100 times on the same datasets in examples 1 and 2, PROJECTION never had a reasonable performance coefficient, yet took 4 hours longer than our tool.

4.8 Conclusions

The benefit of using Gemoda for this problem is then obvious: deterministic and provably complete output even in the face of uncertainty in motif characteristics. The motifs could have been longer than 15 bases, could have had fewer mutations, or could have occurred in a variable number of sequences, and our tool would have found them. Its only obvious negative aspect is its computational expense. The restricted (15,4) problem took 6 hours, while the extended problem took 13 hours. Compared to the runtimes of algorithms like PROJECTION, which can be as low as five minutes for the restricted problem, these runtimes may seem extremely large. In practice, however, this computation time is far from unacceptable; one would not expect to often encounter the need to run motif discovery many times sequentially, particularly if the results being returned to the user are deterministically correct.

Perhaps even more importantly, we have reframed the challenge problem statement in a way that is more biologically meaningful; hopefully this new challenge will inspire other methods that outperform ours in some way. While a deterministic and exhaustive method is always welcome, for some problems it seems that a heuristic approach may provide a good balance between time and accuracy; we look forward

to seeing new tools that address our amended problem with sufficient accuracy.

Chapter 5

Applying Gemoda to GC–MS data

5.1 Overview

In Chapter 3, I described a generic approach to motif discovery and presented some brief examples of how it is successful at finding motifs both in string–valued and real–valued data. The more in–depth example of Chapter 4 showed how a string–valued problem could be solved, even if all of the potential of Gemoda wasn't harnessed (i.e., for the restricted (l,d)–motif problem, convolution isn't necessary to find the desired motifs). The important aspect of this example is that an existing problem, for which Gemoda was not specifically being designed to solve, was easily handled with our generic approach. This chapter will present another example where all of Gemoda's capabilities may not be harnessed to their fullest, but a problem that previously had little software available to handle it can be addressed by Gemoda very easily. By merely taking an external data format, formatting it for input to Gemoda, and running Gemoda iteratively, we were able to discover relevant motifs in gas chromatography–mass spectrometry (GC–MS) data. Again, this speaks to the potential of a truly generic approach that is not dependent upon specific data types, models, or problem nuances.

Analysis of metabolomic profiling data from GC–MS measurements usually relies upon reference libraries of metabolite mass spectra to structurally identify and track metabolites. In general, techniques to enumerate and track unidentified metabo-

lites are non–systematic and require manual curation. This chapter will describe a method and software implementation that can systematically detect components that are conserved across metabolomic samples without the need for a reference library or manual curation. We validate this approach by correctly identifying the components in a known mixture and the discriminating components in a spiked mixture. Finally, we demonstrate an application of this approach with a brief analysis of the *Escherichia coli* metabolome. By systematically cataloging conserved metabolite peaks prior to data analysis methods, our approach broadens the scope of metabolomics and facilitates biomarker discovery.

Much of the research and discussion in this chapter is drawn from the publication that describes the application of Gemoda to analyzing gas chromatography–mass spectrometry metabolomic data:

- Styczynski MP, Moxley JF, Tong LV, Walther JL, Jensen KL, and Stephanopoulos GN. "Systematic identification of conserved metabolites in GC/MS data for metabolomics and biomarker discovery." *Anal Chem* 79: 966 – 973 (2007).

Again, this chapter will frequently use "we" to refer to the collective authors of this manuscript and their contributions to the work.

5.2 Introduction

The goal of metabolomics — the metabolite analog of genomics and proteomics — is the measurement of concentrations (or "metabolite profiles") of as many cellular metabolites as possible, usually with applications to functional genomics [160, 53]. Certain aspects of metabolomics suggest that exhaustive metabolite profiling may be possible: the number of known metabolites present in many organisms (e.g., yeast) is tenfold to hundredfold fewer than the number of genes or proteins [57, 90, 113], and the cost of measuring these metabolites is by comparison significantly lower. To date, the coupling of metabolomic data with other cell–wide data has yielded valuable insight into underlying biochemical processes and has contributed to numerous advances in the area of functional genomics [138, 176, 73, 64]. However,

obstacles to exhaustive metabolite profiling persist, one of the most significant being the chemical diversity of metabolites. Unlike DNA or proteins, metabolites do not adhere to a subunit–based chemistry, so assaying for many metabolites (with many chemistries [89]) simultaneously is difficult. Gas chromatography–mass spectrometry (GC–MS) is one method frequently used to assay for a variety of metabolites, and the aim of this work is to improve the downstream analysis of this GC–MS data independent of upstream experimental protocols (see Figure 5-1). While we describe our approach in the context of GC–MS, there is no reason why this approach could not similarly be applied to other similar types of metabolomic data like liquid chromatography coupled to mass spectrometry (LC–MS). The role of our approach in the context of generic metabolomic analysis is illustrated

5.2.1 Common data processing methods

First, current methods for analyzing metabolomic GC–MS data must be understood. In targeted GC–MS analysis, when only the concentrations of a few specific compounds are desired, only certain regions of the chromatogram or certain m/z values of mass spectra may be considered relevant. For non–targeted metabolomic analyses, on the other hand, the entire chromatogram (for all m/z values) is important, prompting efforts to select experimental parameters that maximize metabolite peak accessibility [125]. However, accurate analysis of all of this data presents some challenges. Some "real" chromatographic peaks may be hard to distinguish from noise. Other peaks may contain mixtures of metabolites that are coeluting, so the individual MS scans of their peaks are not pure spectra of either component. Thus, the immediate processing steps after the storage of raw GC–MS data typically include peak enumeration (distinguishing "true" peaks from noise in a chromatogram) and, with increasing frequency in recent literature, spectral deconvolution (obtaining putative pure spectra from two overlapping peaks). These steps may be performed either with proprietary software for a specific manufacturer's apparatus or with publicly available software like AMDIS [154]. Whether or not spectral deconvolution is applied, the user is left with a set of chromatographic peaks that putatively represent individual com-

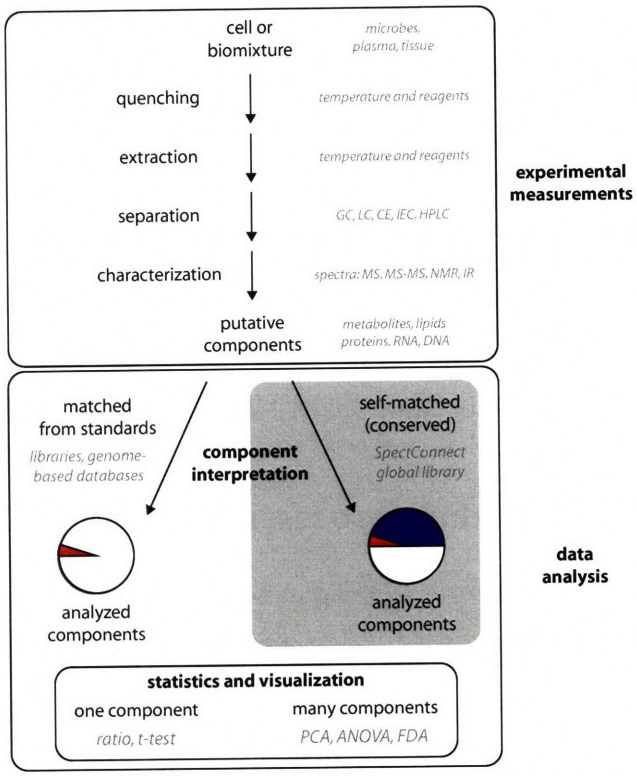


Figure 5-1: A typical metabolomic workflow. Generic steps are indicated in bold font, while italicized gray font gives specific examples of those steps. The goal of SpectConnect is to impact upstream experimental measurements as little as possible. Typically, a cell or some sort of biomixture is prepared through a number of steps to eventually yield a number of putative components as judged by the resulting data (whether GC–MS or otherwise). It is at the interpretation of these components that we aim to apply our approach, requiring only that upstream steps use replicate samples (whether biological or sample replicates). Instead of tracking and interpreting components by matching them to known standard data, SpectConnect uses an essentially self–matching approach to find conserved components that are unlikely to occur purely by chance as artifacts of noise. This approach then easily fits into standard downstream methods of visualization and statistical analysis that are typically only practiced on the known, matched–from–standards components.

ponents in the sample mixture. Often, multiple experimental conditions have been assayed and all of the results need to be analyzed and compared.

It is in the following steps that methods for metabolomic experimental data analysis are rather scattered. Few tools are available for easy comparisons of multiple experimental conditions; ChromaTOF (LECO, St. Joseph, MI, USA) is a representative example of some of the better proprietary software for performing this analysis. However, ChromaTOF can only be used with LECO mass spectrometers, so such a program is not useful for many scientists. Similar problems arise with other manufacturers' software packages; many are inadequate for whole–dataset analysis, and analysts typically do not have the luxury of buying new equipment strictly for its superior software. Alternatively, there are other proprietary programs available that work for a variety of manufacturers' hardware, such as MetAlign (Plant Research International, Wageningen, Netherlands). However, such programs are typically expensive and (as yet) have not seen wide use in scientific literature. There are a few freely–available packages that can perform some of this data analysis, such as mzMine [92] and SpecAlign [180], though these newer tools also have yet to see widespread testing in scientific literature. Rather, a much more common approach is to compare the spectra from a given run to a reference library of spectra so that metabolite peaks may be tracked by names.

In metabolomic experiments using this approach, the scope of most automated data analysis techniques is limited to those compounds which have been isolated, been purified, and had their spectra stored in a reference library. However, even the largest publicly available libraries [54, 122, 16] are often incomplete, leaving many metabolite peaks unidentifiable without additional experimental work [56]. To avoid such issues, one may create a customized reference library using standard reagents; however, this is prohibitively labor–intensive and the resulting library will necessarily be incomplete because many compounds of interest are not commercially available (e.g., only 200 of the 600 known yeast metabolites can be purchased [46]). Alternatively, supplementing a pre–existing library by adding every spectrum from every experimental run is an equally undesirable option: subsequent matches to this "complete" library have less

value, as spectra generated by noise and other biologically irrelevant factors would overtake the true set of cellular components.

5.2.2 Tracking of unidentified metabolite peaks

When a chromatographic peak's spectrum cannot be matched to any reference library entry, the analyst must then decide the disposition of this unidentifiable spectrum. It is not uncommon for such spectra to be discarded. In cases where some of these spectra are retained, this is usually the result of a labor–intensive, somewhat ad hoc process of comparing multiple chromatograms to a reference chromatogram. Table 5.1 enumerates recent GC–MS metabolite profiling studies, some of which relied on non–systematic, manually–driven curation of unidentified metabolite peaks. While previously mentioned tools such as ChromaTOF, Metalign, and mzMine are capable of allowing some tracking of unknowns, this process still requires significant intervention on behalf of the user and may not allow for completely consistent, automated tracking of unknown metabolite peaks. Perhaps more importantly, these methods typically entail comparison to one single reference spectrum. However, if a spectrum may contain significant noise (see discussion below and Figure 5-2), the use of any one given spectrum as a reference for all others may be undesirable, as the "true" low–concentration metabolites may be difficult to parse out from the data. Thus, while unidentifiable peaks are sometimes retained in GC–MS data analysis, an automated, systematic method for including these peaks in subsequent analysis and appropriately accounting for noisy spectra is as desirable a goal for GC–MS metabolomic analysis as it was for NMR analysis [138, 176, 73].

Another goal of metabolomics research is to enumerate metabolites known as biomarkers that discriminate classes of samples obtained from different cellular conditions by being absent, present, or differentially present. As noted above, there are presently few methods for comparing many sets of spectra in order to identify components characteristic of a sample or group of samples. A commonly used method is principal component analysis (PCA) [172, 47, 87]; however, the results of PCA (the loadings) are often not intuitive, and the optimized function (capturing variance) is

Table 5.1: Usage and cataloging of unidentified GC–MS peaks in metabolomic studies. A literature survey reveals many GC–MS metabolite profiling studies rely on non-systematic, manually assisted curation of unidentified metabolites. Recently published GC–MS metabolite profiling studies were curated to determine methods by which unidentified metabolites were handled, if at all. *, +, ++, ** Peaks are cataloged by applying the {AMDIS* , ThermoQuest-MassLab+, MSFACTs++, Xcalibur**} program to a reference chromatogram. As stated in the text, this class of methods fails to use a systematic conservation criterion and requires manual curation to scale to moderate to large data without becoming dominated by noise. Peaks are analyzed by MetAlign, a method described in our text. While the unknown peak data are systematically analyzed, the true metabolites are not cataloged.

Author	Total Peaks	Identified Peaks	Unidentified Peaks Used?	Catalog Method
Fiehn 2000 [55]	326	164	Yes	Non–systematic*
Roessner 2001 [142]	88	61	Yes	Non–systematic+
Taylor 2002 [162]	≥ 400	90	Yes	Non–systematic*
Roessner–Tunali 2003 [143]	?	73	Yes	Non–systematic+
Verdonk 2003 [169]	?	62	No	-
Prithiviraj 2003 [137]	?	253	No	-
Duran 2003 [47]	?	?	Yes	Non–systematic++
Morris 2004 [115]	60	27	Yes	Non–systematic**
Barsch 2004 [22]	200	65	Yes	Non–systematic**
Strelkov 2004 [158]	330	164	Yes	Non–systematic*
Vikram 2004 [170]	1081	49	No	-
Bino 2005 [26]	7500	320	Yes	Non–systematic@
Devantier 2005 [41]	?	29	No	-
Kaplan 2005 [91]	416	81	Yes	Non–systematic*
Desbrosses 2005 [40]	500	87	Yes	Non–systematic*
Tarpley 2005 [161]	?	21	No	-
Herebian 2005 [71]	¿200	130	Yes	Non–systematic*
Tikunov 2005 [164]	322	100	Yes	Non–systematic@
Villas–Boas 2005 [173]	?	45	No	-
Brosche 2005 [31]	?	22	No	-
This work	554	51	Yes	Systematic

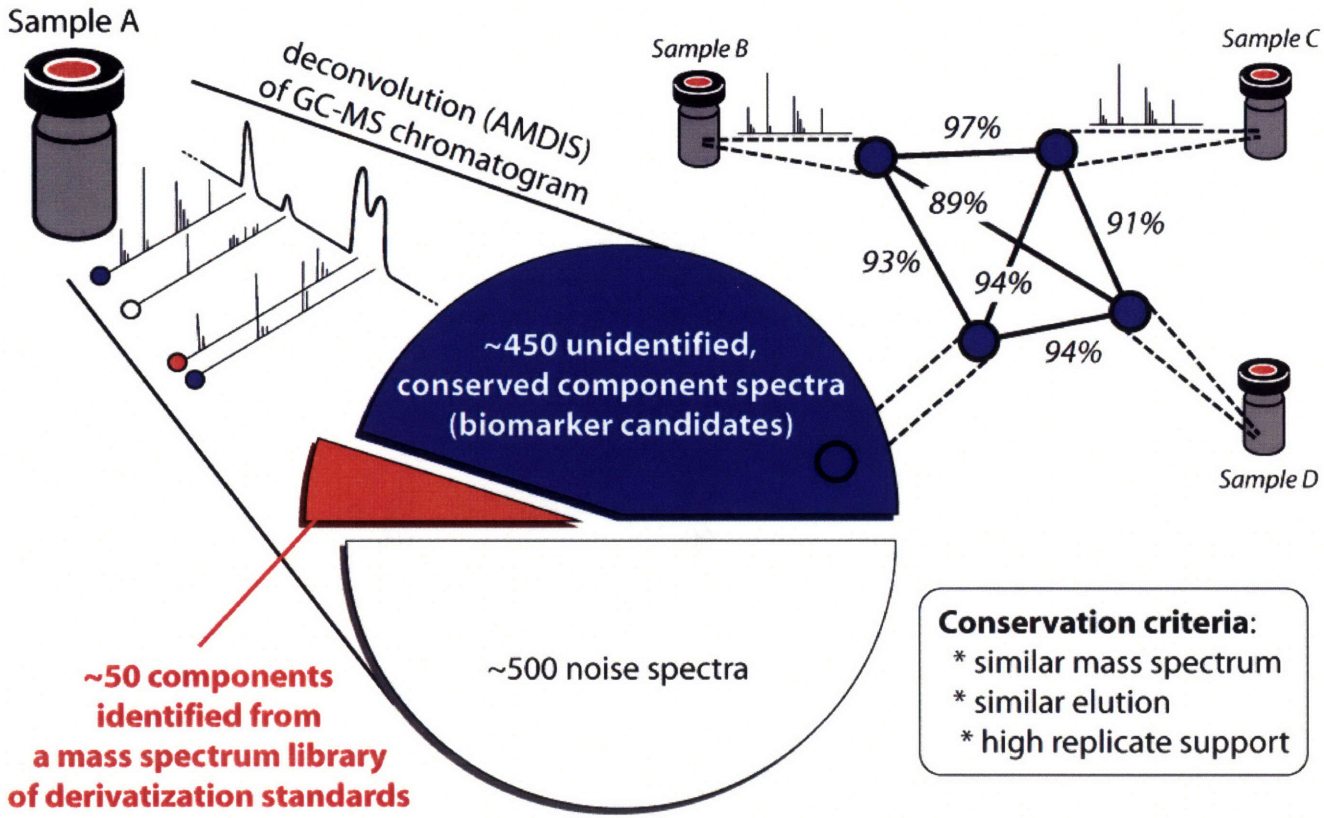


Figure 5-2: The composition of a typical GC–MS metabolomic sample motivates the need to systematically track the potential biomarker components not found in a library of standards. For the GC–MS vial labeled "Sample A", we display the deconvolution and enumeration of components by AMDIS from the GC–MS output. The large number of "noise" components (usually small peaks near the GC baseline) emphasizes that constructing a conglomerate library of all putative components would be unwise. Components labeled with blue circles did not match any spectra in a standard library but, as displayed in the upper right, had highly similar spectra occur in replicate samples. This is not surprising, as construction of standard spectra libraries are confounded by a variety of factors. Given the conservation of this blue component, we conclude that it is not noise, and thus may catalog its spectrum in a SpectConnect library. In future analysis, we can track this specific component among conditions and investigate its potential biomarker activity. Note that though the relative amount of "white" noise spectra may decrease with better deconvolution and peak enumeration, there will still remain a significant number of blue "unidentifiable" spectra without an automated, systematic way to handle them.

not necessarily ideal for biomarker discovery. As previously mentioned, Metalign and other software can perform pairwise spectrum comparison in place of PCA, and other methods also exist [92, 123, 93, 150] to these ends, though most either focus on a small subset of data or are not generally applicable. The method most relevant to this work [81, 82] avoids dependence on libraries of standards, but still depends on complex multivariate analysis; it focuses solely on biomarkers rather than identification and tracking of all possible metabolites in the samples. As such, biomarker discovery remains an open problem, with numerous potential applications in diagnosis and prognosis that reach beyond the scope of a simple classification [121].

5.2.3 The SpectConnect approach

Here, we present a method and freely available implementation, SpectConnect, to automatically cataloger and track otherwise unidentifiable conserved metabolite peaks across sample replicates and different sample condition groups without use of reference spectra. SpectConnect compares every spectrum in each sample to the spectra in every other sample. By doing so, it is capable of determining which components are conserved (according to some criteria) across replicate samples. SpectConnect also determines which of these components differentiate one sample condition (e.g., time or treatment) from another, whether by changes in concentrations or merely by their presence/absence. The only requirement of the experimental measurements is that each sample condition must have replicates. In a sense, SpectConnect relies on an increase in signal relative to noise that is created by this requirement of replicates. While injection ("technical") replicates are the easiest way to provide the required replicates, it is also desirable to include biological replicates in a group of samples. This is due to systematic error in peak detection and deconvolution software that may consistently find a noise peak in technical replicates. Though this approach adds more biological variability to a group in terms of metabolite concentrations, it should have significantly less impact on the presence/absence of a metabolite. Ultimately, we hypothesize that these "true", important spectra will be conserved across most or all replicates of a sample, while spectra that are artifacts of noise will not.

For the core of the necessary computations, SpectConnect uses Gemoda, our freely available generic motif discovery algorithm for sequential data [84]. Gemoda efficiently compares candidate spectra and identifies conserved spectra across samples using various clustering methods. If a chromatographic peak is true signal and not noise, we expect that the mass spectrum of each of its occurrences in the different samples will be pairwise similar to each other. Projected onto a graph consisting of nodes for each spectrum and edges between each pair of spectra that are similar, this pairwise similarity defines a cluster known as a "clique". By limiting our clusters to cliques, we exclude potential artifacts of noise and focus on what we believe are the true metabolites in the sample. We exploit Gemoda's ability to efficiently find cliques in order to locate these clusters of spectra that represent the true metabolites. An illustration of this process can be seen in 5-2.

With SpectConnect, we can find three distinct types of information. First, we identify all of the components that are conserved across a single group of samples or replicates. Second, for each metabolite peak conserved in at least one group, we can determine all other groups in which it occurs. Finally, for any given pair of groups, we can find the likelihood, to some degree of statistical significance, that each metabolite peak is present in unequal amounts in the groups.

5.3 Methods

5.3.1 Experimental methods

Standard mixtures were prepared using commercially–available stock chemicals. *E. coli* strains obtained from previous work [5] were fermented and sampled over 30 hours. The samples were quenched and frozen; after thawing, the metabolites were separated from the rest of the mixture and concentrated. MCF derivatization [171] was performed prior to GC/MS analysis using an HP 5890 GC coupled to an HP 5971 quadrupole mass selective detector (EI). Additional details on all experimental and statistical protocols used can be found in AppendixA.

5.3.2 Peak identification and deconvolution

AMDIS [154] was used to perform component peak identification and spectral deconvolution. Our purpose in using AMDIS is to distill the raw GC–MS data into a more easily digestible set of deconvolved peaks for analysis. This is represented in Figure 5-3. AMDIS performs two important tasks as an upstream part of SpectConnect–based analysis. First, AMDIS enumerates all of the peaks that it believes to be "true" and not just noise or baseline. Second, it uses a deconvolution algorithm to determine whether there are multiple pure components coeluting to create one broad peak; if there are, it enumerates both peaks and determines their appropriate spectra. Deconvolution is a well–studied problem in other fields like signal processing and image processing; it essentially means attempting to tease apart the effects of two overlapping signals, or functions, in the presence of noise. The application here is obvious: with two peaks eluting off of the column, their respective signals are defined by the mass fragments they create. Two sets of mass fragments occurring at the same time can look like one set of fragments, and thus look like only one signal, so deconvolution approaches attempt to identify when there is only one set of ion fragments and when there are two.

The following parameters were used: medium shape requirement, low sensitivity, and medium resolution. Each GC–MS sample's results were processed, and the .ELU files created as output of AMDIS (and containing data for all of the enumerated peaks in each sample) were saved for use with SpectConnect. Though in principle any program could be used for component peak enumeration, we designed this implementation of SpectConnect to work with AMDIS output since AMDIS is freely available and accepts a wide variety of manufacturers' raw data formats. However, AMDIS's tendency towards false positives in an effort to be sensitive [154] makes subsequent clustering steps much more difficult and is one reason that SpectConnect performs "pre–processing" steps, as mentioned below. However, even if peak enumeration and deconvolution were ideal, the resulting data would still greatly benefit from a systematic search for conserved metabolite peaks and biomarker candidates.

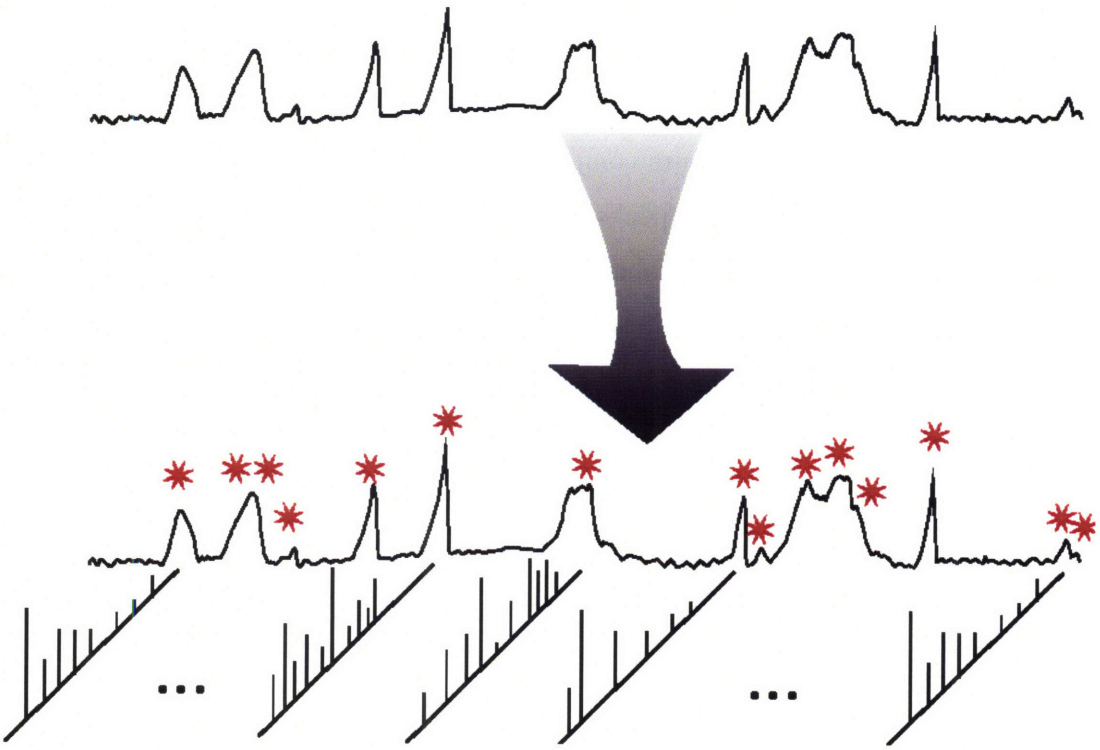


Figure 5-3: The important function of AMDIS as an upstream processing step for SpectConnect analysis. AMDIS takes the raw GC–MS data (top of figure) and distills it into only a few well–defined peaks with known mass spectra (bottom of figure). Pure spectra even in the case of coeluting metabolites (overlapping chromatograms) is obtained by deconvolution on the GC–MS signal. The resulting peaks and their ion spectra are then used as the starting point for replicate analysis in SpectConnect.

5.3.3 SpectConnect

SpectConnect is implemented in Python (http://www.python.org) and uses Gemoda [84] for the majority of its computations. A user may choose how well two spectra must match to be deemed "similar", how much error should be allowed in retention time for occurrences of the same metabolite peak, and how many times a peak must occur to be considered "conserved" (also known as its support). This study used the default values for these parameters, which are as follows: spectra must be 80% similar based on a weighted dot product [154] to be considered similar, they must be within one minute in retention time to be considered similar, and they must occur in at least 75% of sample replicates to be considered conserved. The weighted dot product for our spectrum–to–spectrum comparison is defined as:

$$\frac{\left(\sum_m m^2 \sqrt{I_{1,m} I_{2,m}}\right)^2}{\left(\sum_m m^2 I_{1,m}\right)\left(\sum_m m^2 I_{2,m}\right)}$$

where m takes on all valid m/z values in either mass spectrum and $I_{1,m}$ and $I_{2,m}$ are the ion intensities at $m/z = m$ for the first and second spectra being compared, respectively. Additional (optional) restrictions can also be placed on the data analysis, including minimum relative abundances for peaks. Further analysis of parameter selection and resulting limitations is given in the Discussion section.

Given a set of .ELU files representing replicates from the same sample condition, SpectConnect first parses each file to extract all pertinent information, including the retention time and mass spectrum of each peak. It then uses Gemoda to identify and eliminate all internal matches from each sample: any spectra within a single sample that are within the elution threshold and meet the weighted dot product similarity criterion are combined into a single group, and one single spectrum is chosen as the representative peak for further comparisons and clustering. This is due to the aforementioned oversensitivity of AMDIS; it may return multiple peaks with highly similar retention times and spectra that are really the same metabolite. As such, we look to find all of the peaks in a small window that are extremely similar and reduce them to just one representative peak (see Figure 5-4).

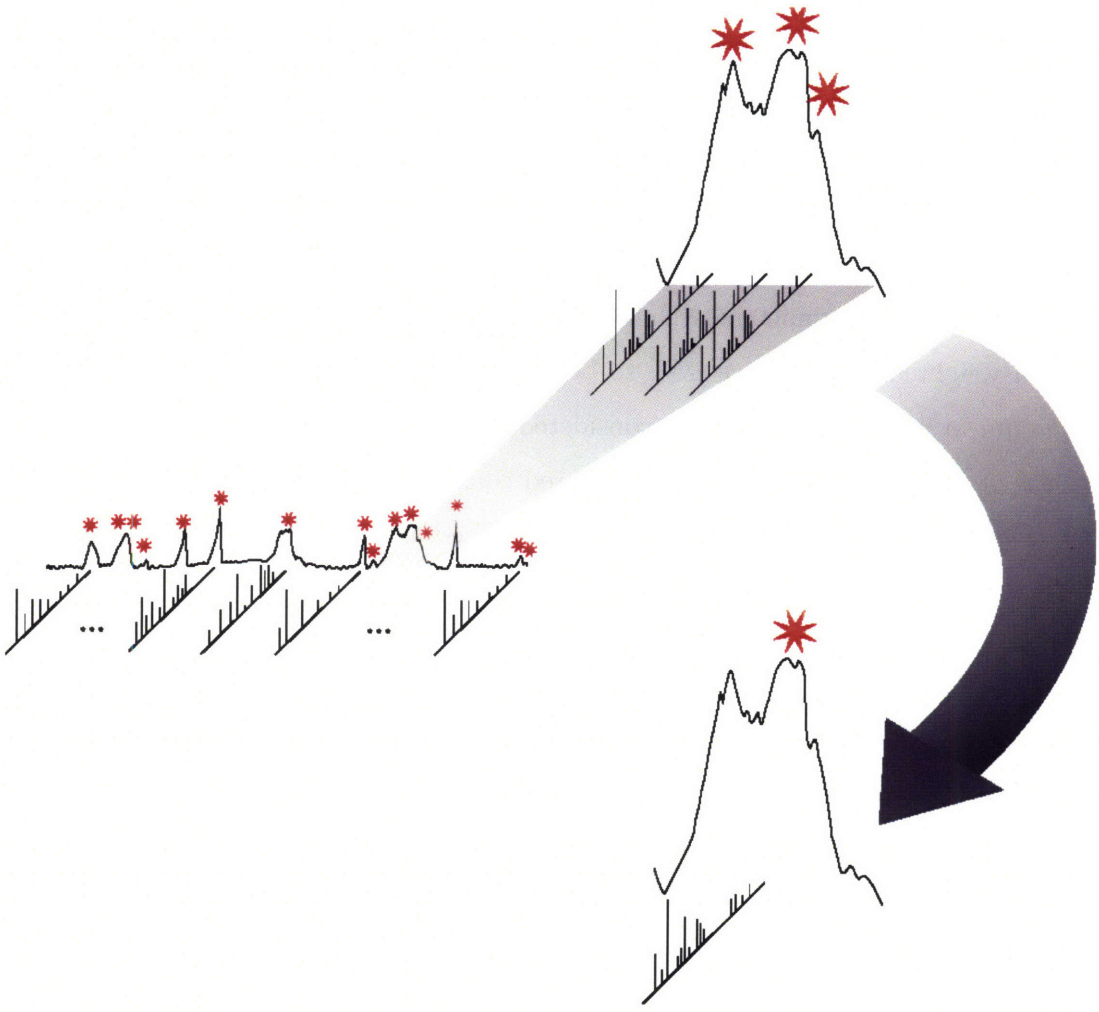


Figure 5-4: The pruning preprocessing step implemented in SpectConnect. As noted in the main text, AMDIS frequently is oversensitive (even when its own parameter values are adjusted appropriately) in identifying "unique" peaks. AMDIS often identifies multiple peaks with extremely similar retention times and mass spectra as being unique even though we suspect from analyzing the data that they only come from one metabolite. As such, we implemented a preprocessing step that looks to distill all of these "false positive" extremely similar peaks into one representative peak by identifying all of the false peaks within some window of time on the chromatogram. All peaks that are sufficiently pairwise similar to each other are then removed, except for the one with the largest relative abundance. Due to its high peak area in the chromatograph, we expect this to be the main peak and expect that its spectrum is the least sensitive to noise, so we use it as the representative peak.

After all replicate input files have been parsed and pre–processed, the resulting information is used to find conserved metabolite peaks. Each sample's reduced set of spectra (which now includes unique spectra that are not internally similar) is supplied to Gemoda for clustering (see Figure 5-5). Here, the appropriate elution, similarity, and support thresholds are enforced. As stated earlier, we require that true metabolite peaks have spectra that are well–conserved across replicate samples. We use Gemoda's maximal clique–finding algorithm to find sets of pairwise similar spectra. However, our requirements can lead to complications because clique–based clustering does not require that each item participate in just one cluster. Since a given metabolite peak may be involved in multiple clusters, it is important to minimize overlapping similarities and similarities that are not believed to be as significant. This desire to ensure a library of non–similar metabolite peaks, combined with AMDIS's high–sensitivity and low–specificity approach to peak identification, makes the previously explained pre–processing steps necessary.

For each clique returned by Gemoda, the most representative spectrum is chosen, as judged by a weighted dot product of spectra. Since the cliques may be overlapping, representative spectra may be self–similar, and so this set of spectra is further processed to eliminate internal matches in the same way as described above. After this step, SpectConnect creates a final "library" of metabolite peaks conserved for this sample condition; this library is returned to the user as output and is used in additional calculations.

The identification of conserved peaks is repeated for each "sample condition" or set of injection or biological replicates, resulting in a set of peak libraries. These sample libraries can then be further condensed into one cumulative library by using Gemoda to determine in how many conditions each metabolite peak occurs. The resulting library is essentially the union of all previous libraries, and it is also returned to the user as output.

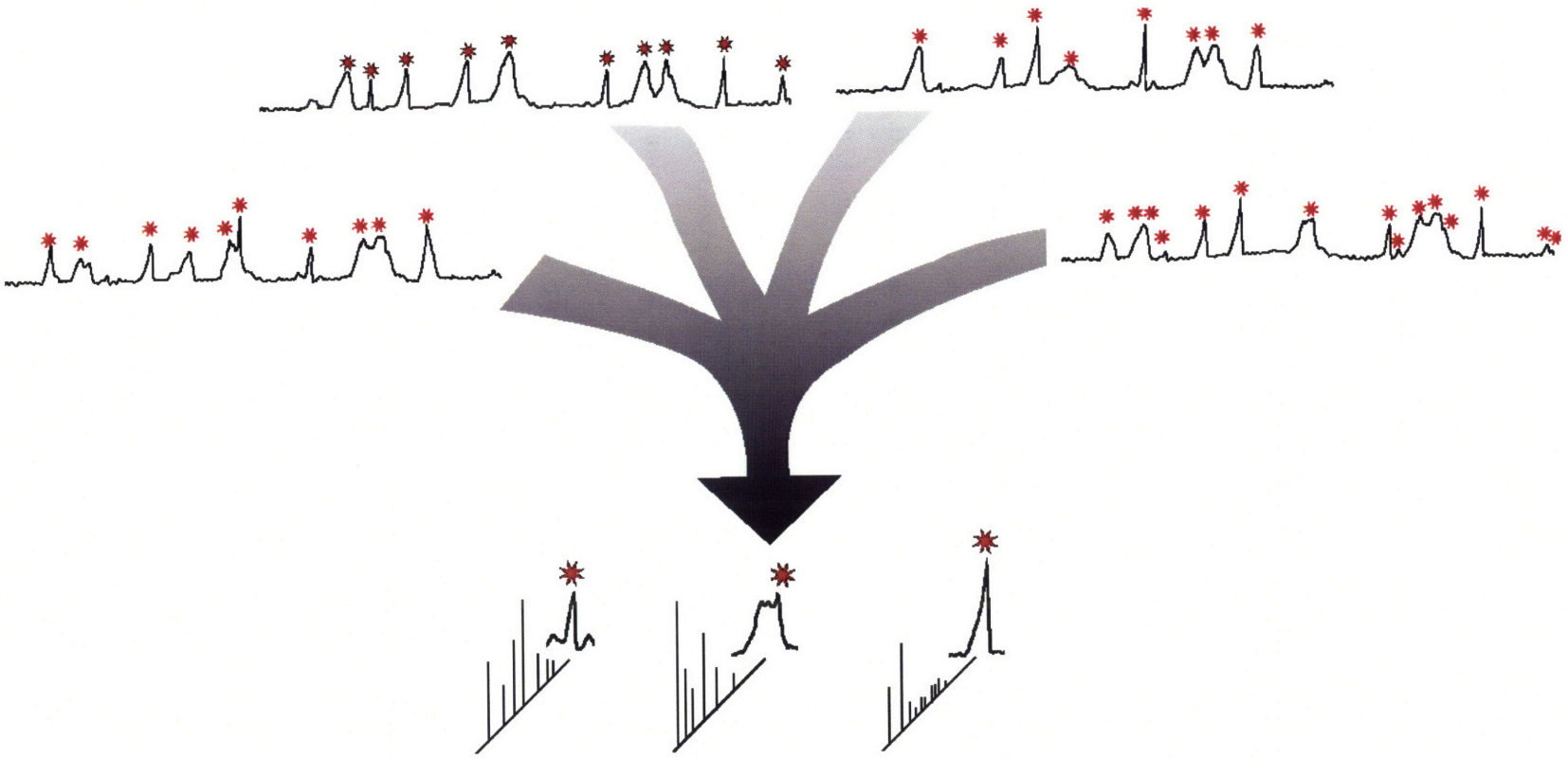


Figure 5-5: The core step in the SpectConnect approach. After identifying the different unique metabolite peaks and their respective mass spectra, SpectConnect uses Gemoda to compare all of the peaks and identify well–conserved clusters (cliques) of metabolite peaks across samples. These conserved metabolite peaks are then most likely to be "real" metabolite peaks, even if it is not clear what chemical structures produced their mass spectra.

5.3.4 Statistical Methods

Principal Components Analysis

Principal components analysis was performed to assess the ability of proposed metabolites to capture the variance in metabolic data across multiple time points in two strains of *E. coli*. First, a cumulative library of components was identified from the results of all samples of all strains at all time points. Biological replicates at the same time points were included in the same condition, yielding 15 conditions (three strains at five sample times). The set of all components that were conserved in at least one strain for at least one time point was obtained using SpectConnect. Then, a wild–type strain and strain 2 were analyzed using two different libraries: a previously available library of MCF–derivatized standards (the "known library"), and the library of conserved metabolites provided by SpectConnect. The metabolite–normalized (or column–normalized) relative abundances for each sample were then used to compute principal components for each dataset in Matlab (Mathworks, Natick, MA, USA).

Statistical Analysis

Necessary statistical tests are performed for each identified metabolite to determine if it is differentially present to a significant degree. For each metabolite, each set of conditions is compared pairwise using a two–tailed t–test of the relative abundance (a percentage of total sample signal) as reported by AMDIS. If the metabolite is identified in both conditions in a given t–test, then the t–test is straightforward; if not, then we make a conservative adjustment. We note that metabolites near the threshold of detection may not be consistently identified using AMDIS. Consequently, we approximate a metabolite's relative abundance for a sample condition in which it was not identified by a normally distributed sample of numbers whose mean and standard deviation are equal to the relative abundance of the smallest peak in the input .ELU files. This allows some conservative quantification of the uncertainty involved in "not detecting" a metabolite in one sample by approximating it with a distribution whose resulting abundances would not be expected to be detectable

using AMDIS and SpectConnect. The result of a t–test is considered significant if it meets an overall p = 0.01 confidence level. In order to avoid overestimating the importance of our tests, we applied a Bonferroni correction for the total number of t–tests performed. Metabolite condition–pairs that meet the t–test of statistical significance are noted in the cumulative library output. It is important to note that relative abundance is not equivalent to abundance, and our p–values reflect confidence in differential relative abundance. If desired, a more rigorous approach to determining absolute abundance can be taken using internal standards. For the purposes of this work, we believe such an approach is unnecessary when screening for differential metabolites.

5.4 Results

5.4.1 Mixtures of known components

To verify SpectConnect's ability to enumerate individual components of a mixture, we analyzed a known standard mixture of amino acids with MCF derivatization in replicate GC–MS runs; the results of this experiment can be found in Table 5.2. Of the standard mixture components, 16 should have been detectable using MCF derivatization; using SpectConnect, we identified 15 of them. Isoleucine and leucine could not be simultaneously identified due to the resolution of our pre–processing technique (see Methods): though they are properly deconvolved by AMDIS, their spectra are too similar to be classified as distinct by SpectConnect. 32 additional peaks were also detected as conserved across replicates, of which some were identified as byproducts of the derivatization reagents. We believe that the remainder of these peaks, which are tenfold to hundredfold smaller in size than the median of the known components' peaks, reflect impurities in the stock mixture.

We next analyzed the same mixture spiked with additional compounds. Using the same amino acid standard as above, we spiked eight stock chemical compounds into the standard (see Table 5.2) and used MCF derivatization to analyze the re-

Table 5.2: A control experiment confirms that SpectConnect can identify metabolites in a known mixture and differential metabolites in a spiked mixture. Analysis of an amino acid standard mixture later spiked with additional compounds demonstrates SpectConnect's ability to identify conserved and differential components in a mixture. *Note that due to the resolution of our preprocessing technique (see Methods), leucine and isoleucine can not be detected as distinct.

Class:	Compounds	Conserved in Standard Mixture	Conserved in Spiked Standard
Amino acids:			
Polar:	Gly, Ser, Thr, Cys	+	+
Aromatic:	Phe, Tyr	+	+
Charged:	Asp, Glu, Lys, His	+	+
Nonpolar:	Ala, Ile/Leu*, Met, Pro, Val	+	+
Other metabolites:			
Keto acids:	ketoisocaproic acid, alpha–ketoglutarate		+
Hydroxyacids:	lactic acid, citric acid		+
Fatty Acids:	stearic acid		+
Amines:	3–aminobutyric acid		+
Other:	methylmalonic acid, chlorophenylalanine		+

sulting sample in replicate GC–MS runs. Each of these compounds was identified by SpectConnect as occurring exclusively in the supplemented mixture. In addition, approximately 100 other peaks were identified as conserved in the supplemented mixture and not present in the control mixture. Based on analysis of single–compound GC–MS runs and the fact that these peaks are also tenfold to hundredfold smaller than the median peak of the known components, we are confident that these extra peaks largely represent impurities introduced with the addition of multiple doping compounds. Some of these peaks were even identified from our library of standards: for instance, in the spiked sample we found conserved peaks for *cis*–aconitate, a dehydration product of citric acid and thus a reasonable "contaminant". It should also be noted that aspartic acid (from the standard mixture) and citric acid (added to the standard mixture) coelute, yet we still identified both as unique mixture components and only citric acid as discriminatory between sample groups.

By perturbing each SpectConnect parameter from its default, we evaluated the impact of parameter selections on our results (see Table 5.3). This impact was fairly small for this example; the primary impact was on the number of other, unexpected conserved peaks that were found. All parameter choices but one were able to detect the conserved and differential metabolites in the two samples.

5.4.2 Biological samples

To demonstrate the capabilities of SpectConnect on biological samples, we analyzed GC–MS time–course data from fermentation runs conducted with three different strains of *E. coli* (see Appendix A). The strains came from previous work in engineering the overproduction of lycopene [5]. We found that across five time points over 30 hours, there were a total of 544 metabolite peaks (chromatogram peaks) that occurred in at least one of the strains in at least one time point (but not in a blank derivatization control), while 184 of those occurred in all of the strains in at least one time point. (Parameter sensitivity for the number of conserved peaks found is addressed in Discussion.) Qualitatively, this indicates that the genetic differences of these strains have caused significant differences in their respective metabolisms. This

Table 5.3: An analysis of the impact of SpectConnect's parameters on the number and type of conserved peaks that it finds. To determine the impact of our parameter selections, we individually perturbed each parameter from its default and analyzed the results we obtained. *Note that due to the resolution of our pre–processing technique, isoleucine and leucine could not consistently be detected as distinct (though in spiked mixtures, the two were frequently detected as distinct).

Thresholds	Conserved Amino Acids Found (16 possible*)	Conserved Additives Found (9 possible)	Total Conserved Peaks Found	Conserved Peaks Not Expected
Default parameters (1 min elution time, 80% similarity, 75% support)	15	9	139	110
High elution threshold (0.5 min)	15	9	153	123
Low elution threshold (2 min)	15	9	127	98
High similarity threshold (90%)	12	8	130	108
Low similarity threshold (70%)	15	9	125	87
High support threshold (100%)	15	9	92	63
Low support threshold (50%)	15	9	217	188

result is to be expected for mutants with deletions of metabolic enzymes: some subsets of metabolites are rendered inaccessible, so a significant metabolic adjustment is necessary to compensate for such changes.

Using this cumulative library of 544 metabolite peaks, we then analyzed the metabolomic profile of one mutant strain relative to that of the reference strain in the course of the fed–batch cultivation. Figure 5-6 shows, for the time sample with the greatest metabolic deviation from the reference strain as measured by lycopene production (strain 1 at 24 hours [5]), the increase in the number of metabolite peaks that are detected as biomarkers when using the SpectConnect library relative to those detected when using a pre–existing library of reference spectra for MCF–derivatized metabolites. The p–values plotted are the results of t–tests comparing relative abundances (abundances normalized by total ion count) of each metabolite peak between the two sample conditions. (These tests are performed automatically by SpectConnect to identify biomarker candidates.) Only molecules that can be detected in at least one replicate of each sample are included in Figure 5-6. While a few compounds are identified using the known library, significantly more spectral signatures are detected with SpectConnect.

Figure 5-7 demonstrates that principal components analysis using the SpectConnect library for the data qualitatively captures the differences between strains and time points better than the previously known MCF reference library. The SpectConnect library allows better resolution between the two strains and even suggests more distinct "trajectories" along sample times than is possible using the MCF reference library, including a divergence of trajectories before a difference in lycopene production is detectable. Since PCA is unsupervised, these results support the notion that the metabolite peaks returned by SpectConnect capture biological aspects of the system rather than just noise. As noted above, PCA is not the ideal method for identifying potential biomarkers; rather, Figure 5-6 indicates our putative biomarkers as defined by pairwise t–tests. Figure 5-7 serves to support the notion that the additional metabolite peaks detected by SpectConnect help to better capture the variance in the data.

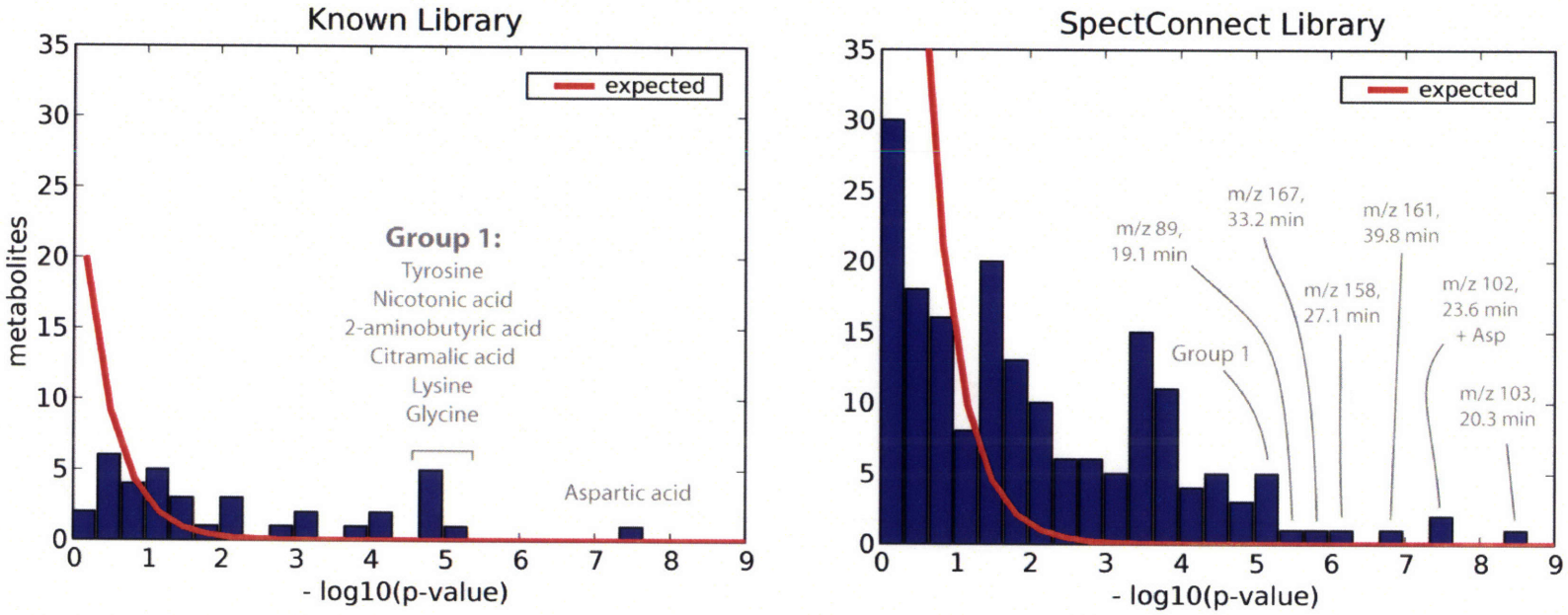


Figure 5-6: Comparing conditions using a global SpectConnect library helps identify more biomarker candidates. In each graph, we compare wild type and mutant (strain 1) cell populations sampled from their respective fed batch reactors at the point of largest deviation of phenotype of interest (lycopene production at fermentation time 24 hours). In each graph we use the identical set of experimental measurements to calculate confidences in differential levels (p–value) among the samples for those metabolite peaks which were detected in both conditions. (We do not display data for those peaks which may be considered biomarkers merely by their presence or absence.) Lines illustrating the expected number of metabolite peaks in each bin are based on a uniform random distribution of p–values. A) Using a reference library for MCF–derivatized metabolites, previous methods would only be able to identify the biomarker candidates in this p–value histogram. B) After using SpectConnect to create a larger library of conserved components and their spectra, we can identify a greater number of statistically significant biomarkers, including all of the most significant biomarkers found using the known library. Interesting biomarker candidates may then be further explored and biologically identified using known ad hoc techniques involving the spectra and retention information.

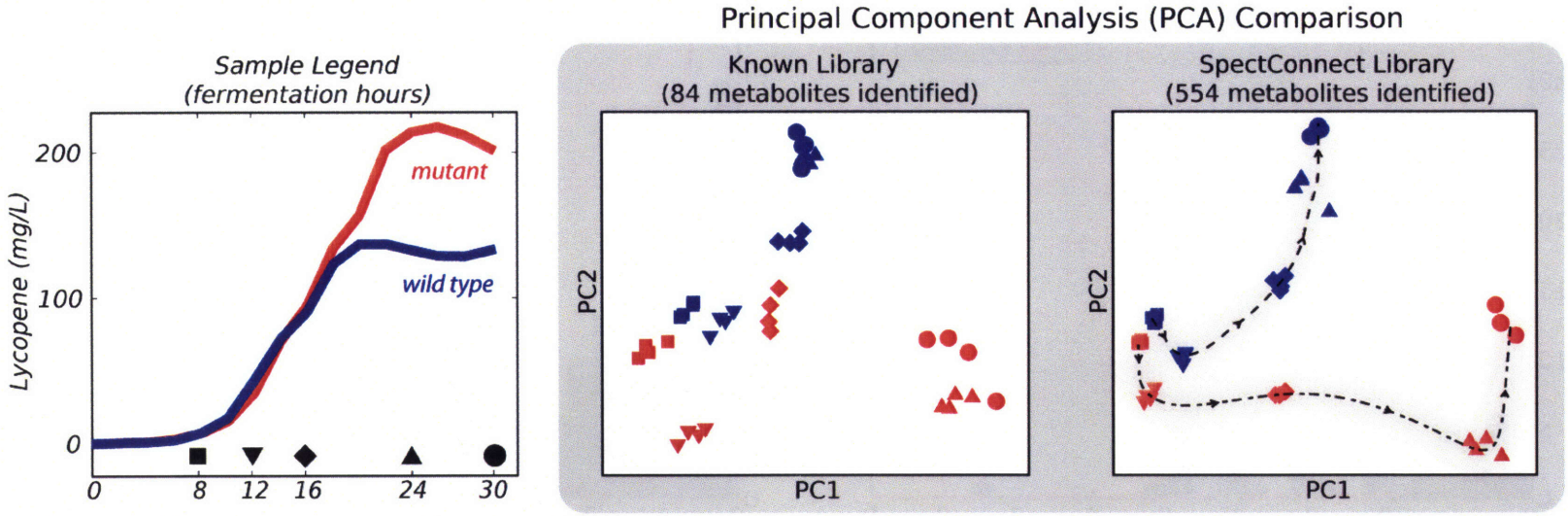


Figure 5-7: Including all conserved components in a PCA analysis better captures biological variation in metabolite profiles of cell physiological states. The first panel shows lycopene production as a function of time for the reference strain and the mutant strain (strain 1) in separate bioreactors. The 40 data points in the second and third panels represent identical experimental metabolic measurements from 40 samples. Four samples are taken from each strain at five different times, as denoted by the shapes on the x–axis in the first panel. A) With the library of reference spectra, some separation is seen between strains, though some time points are difficult to resolve. We particularly note the lack of separation at time points 24 and 30 hours. B) After using SpectConnect to create a global library of conserved components, we obtain a better characterization of metabolic states and profiles. The SpectConnect library allows better resolution between the two strains and even suggests more distinct "trajectories" along sample times than is possible using the MCF reference library, including a divergence of trajectories before a difference in lycopene production is detectable.

5.5 Discussion

The method and program that we present here allows for systematic, automated analysis of GC–MS metabolite profiling data sets, including metabolites that may not be structurally identified by a reference library. In the *E. coli* metabolome dataset, we see almost an order of magnitude increase in the number of metabolite peaks that can be tracked with GC–MS measurements without manual curation of unidentified peaks. Accordingly, the enriched metabolite peak set allowed for a more fruitful downstream data analysis: differential relative abundances identified more biomarker candidates than would be possible using strictly library–based approaches without unidentified peaks, and PCA projections offered better characterization and separation of different sample classes. In addition, we note that all SpectConnect computations performed in this work took reasonable computation times, ranging from minutes to a few hours.

Since our approach relies on adjustable thresholds at which two components are considered similar and thus conserved, some caveats related to our assumptions should be explicitly addressed. Using set thresholds for spectral similarity or similarity in retention time necessarily implies distinguishing sharply between similar cases on either side of a threshold. This distinction is obviously not ideal, and for these reasons an automated system can never fully replace an experienced and knowledgeable researcher. Overall, the ability of our algorithm to systematically track conserved components relies upon intelligently–chosen assumptions, the choices of which inherently create finite resolution limitations to the exhaustiveness of the conserved component search. For instance, using the *E. coli* data set with 554 total metabolite peaks, changing the elution threshold will yield 549 to 569 metabolite peaks (at 2 and 0.5 minutes), changing the similarity threshold will yield 440 to 602 metabolite peaks (at 90% and 70% similarity), and changing the support threshold will yield 308 to 1659 metabolite peaks (at 100% and 50% required conservation within replicates). These data are presented in Figure 5-8. Overall, these variations seem reasonable or expected, especially considering the large magnitude of change in threshold parameters and the fact that some of the parameters (i.e., 50% required conservation within replicates)

will intuitively yield a significant number of false positives. The more detailed parameter perturbation experiments detailed in Table 5.3 support the fact that, at least in vitro, SpectConnect is robust with respect to identifying the known conserved and differential metabolites in a sample. In general, though, it is clear that the exact values chosen for thresholds have some effect on the method's results. Nonetheless, for the purposes of trying to track as many metabolite peaks as possible in as simple a fashion as possible, we believe that such a cost is marginal compared to the benefit of a broadened scope of analysis.

The selection of these thresholds is based upon experimental protocols and simple heuristics. The default similarity threshold of 80% is chosen to conform with the commonly–implemented assumption that 80% similarity for comparison of a spectrum to a library represents a likely match. The default support threshold was chosen as 75% of samples, allowing for some experimental noise from the theoretical value of 100% but attempting to avoid inclusion of artifactual spectra that may occur at random. Since we used no internal retention index standards, we allowed for elution similarity thresholds of one minute to account for column drift and other noise in retention time data. The use of internal retention index standards would allow the reduction of elution time similarity thresholds, likely resulting in even finer resolution and less noise in SpectConnect's results. Finally, one may note that our preprocessing method for AMDIS data decreases the resolution of our approach. For instance, SpectConnect cannot distinguish between isomers with very similar elution times. Improved peak enumeration and deconvolution methods will help remove the need for this preprocessing and thus restore finer resolution to our method. As such, this shortcoming is representative of the current implementation and is not inherent to the general approach.

Despite these limitations, SpectConnect shows a great deal of promise for future applications. Removing the need for a library of reference spectra not only allows superior data analysis but also gives more flexibility in metabolomic sample preparations. Because SpectConnect works without adjustment on GC–MS data produced using any derivatization chemistry (including the most popular TMS–based tech-

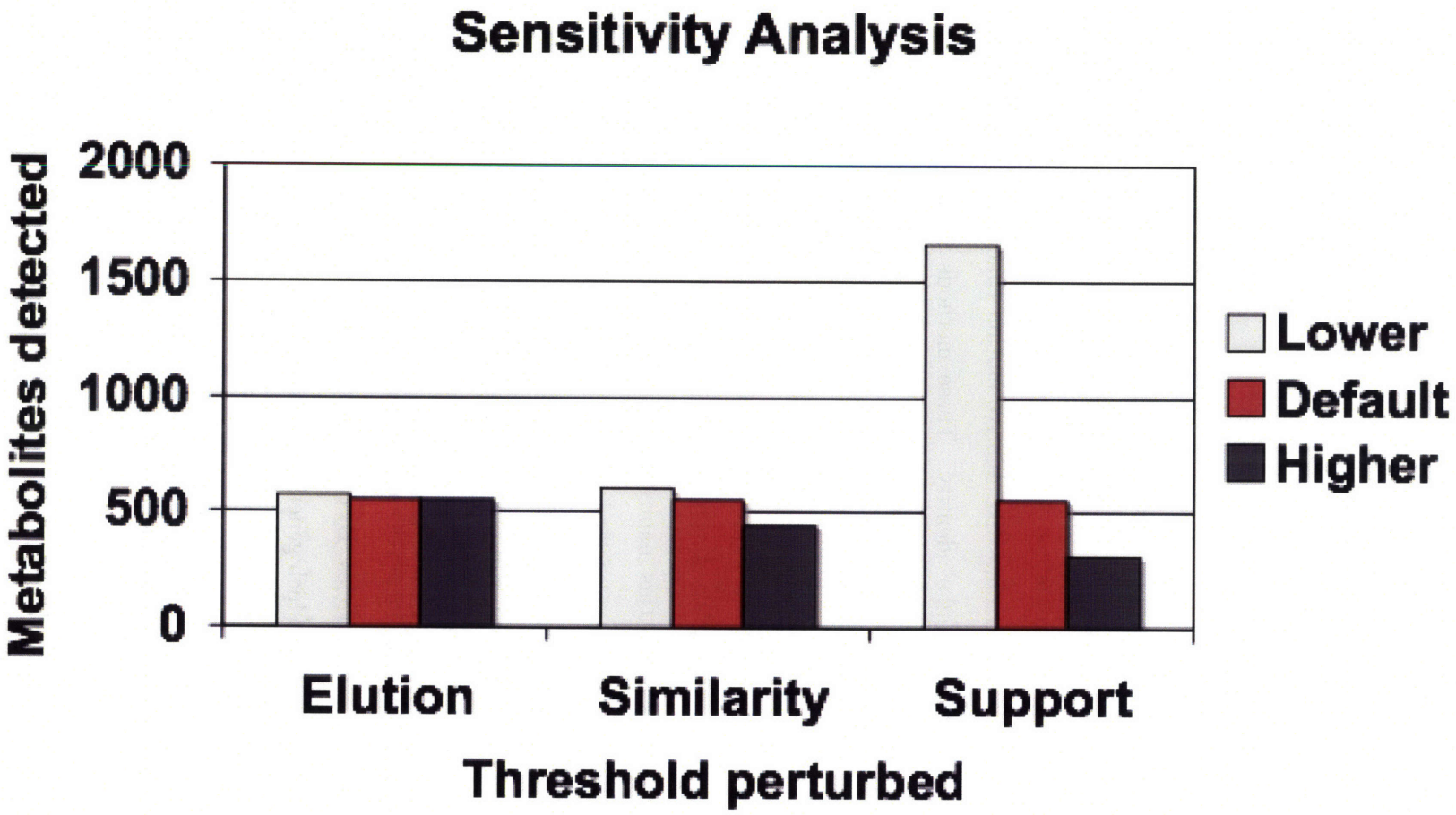


Figure 5-8: Sensitivity of the *E. coli* dataset's SpectConnect results as measured by number of metabolite peaks found conserved. The changes in parameter values in some cases are rather significant, marking different goals for the data analysis. For instance, using "high" support for SpectConnect analysis means that each metabolite must be present in every sample, meaning that many low–lying peaks that could easily be missed in upstream pre–processing techniques or due to biological error (sample variance) will be lost. In this case, it is not surprising that the number of conserved peaks found decreases so drastically. Thus, while the high sensitivity to the "support" parameter is not desirable, it is also likely not easily avoidable.

niques), the adoption of newer, potentially simpler derivatization methods [173] is made more practical, as the need for a comprehensive reference spectrum library is not as pressing. Just as chromatographic and mass spectrometry instrument parameters are explored and optimized [125], variants of a derivatization chemistry may be tested to increase the range of measurable compounds.

When interesting metabolite peaks are identified using SpectConnect, established methods can be used to ascertain the molecular identities of components that cannot be matched to a library reference spectrum. Specifically, exact mass measurements from high resolution mass spectrometers can be used to find molecular composition and structure; additional context clues like relative elution time and isotope ratios can further help pinpoint molecular identity. These methods have been adopted since the early stages of metabolomics research [56]; revision and refinement of these methods is still an active research area, as seen by recent efforts to enhance our ability to structurally identify high molecular weight metabolites [96]. Sharing these newly characterized spectra in recently–created public mass spectrum databases [147] would enable even faster exploration of previously unidentified cellular metabolites. While we do not address such issues in this manuscript, pursuit of structural identification for unknown metabolites that we classified as highly discriminatory is certainly the next step in better characterizing the metabolic perturbations in our *E. coli* strains. Such identifications and characterizations may play a significant role in advancing the capabilities of the field of metabolomics.

SpectConnect also allows for the potential to thoroughly explore the differences between metabolomes of different species. By comparing many diverse species we may be able to determine the extent to which genetic similarity correlates with metabolomic similarity. That is, we can perform a more accurate assessment of the phylogenetic uniformity of the metabolome than is otherwise currently possible, as we will not be limited to those metabolites that are currently well–known and well–characterized.

In total, SpectConnect's direct contributions are significant: it helps broaden the scope of the systematic search for biomarkers, and it provides a unique, powerful, automated, and simple tool for interpreting complex, high–dimensional metabolomic

data.

Chapter 6

Future directions in metabolomics: Exploring an organism's metabolome

6.1 Overview

This chapter will address future directions in metabolomics as relevant to this work. This discussion will be dedicated to a project that was inspired by the potential applications of SpectConnect (as presented in Chapter 5) to large–scale metabolomic experimentation. The results of preliminary explorations in these directions will be presented in this chapter. I will discuss potential future directions for this specific project, including deficiencies in the current experimental approach that have been identified and must be fixed in order to continue with the project.

6.2 Introduction

So much of the explanation regarding SpectConnect referred to "metabolomics" as the ultimate goal of the work. One purpose of metabolomics is, obviously, to simultaneously analyze as many metabolites as possible, in the hopes that one can capture all of them in one assay. SpectConnect looks to move towards this by allowing the

tracking of metabolites by mass spectrum rather than by name, thus eliminating a previous limitation on metabolomic data analysis. In biological systems, though, some metabolite intermediates are present in so small quantities that they may be well below the detection limits of one's instrumentation. In this case, the abilities of our data analysis techniques are not the limiting aspect of the process. The fact that some of these metabolites are just not always present in detectable quantities will be the analysis–limiting aspect of this process.

One can easily imagine a situation where a pool of a certain metabolite usually has a concentration just below the detection limits of the analytical instrumentation. Due to biological variation, one may expect that occasionally this metabolite may be present in high enough concentrations that the instrument can detect it. However, if we do not have a reference standard for this metabolite (as would seem to be likely for such intermediates), then we would have no way to identify those times when the metabolite was detectable. SpectConnect would also be ineffective, as one isolated metabolite peak amongst multiple biological replicates does not provide sufficient confidence in the metabolite's authenticity to include it in SpectConnect analysis.

It is in addressing these issues that we were inspired to see if we could characterize an organism's entire metabolome. On one level, it is an intrinsically interesting scientific problem that no one has truly been able to solve to date (with the possible exception of the Human Metabolome Project [179], though many have doubts about the completeness of that database). On another level, and more related to the above discussion, if we can characterize the organism's entire metabolome, we can utilize both data analysis approaches — SpectConnect and library–based matching — for arbitrary samples. This would allow us to track both conserved and sporadic metabolites, perhaps giving even more potential for biomarker discovery and sample characterization than SpectConnect alone can give. Of course, one can imagine that if one had a truly complete reference library that the need for SpectConnect would no longer exist; this is only partly true. While knowing the reference spectra for all metabolites under one set of derivatization conditions would be sufficient for tracking of metabolites under those conditions, there are numerous different sets of

derivatization conditions that people may be interested in using. Also, the addition of foreign nutrients or chemicals to a medium or biological system (for example, a person taking a prescription drug) will cause non–native metabolites to appear that would not otherwise be included in our characterization of the metabolome. Thus, it seems there would always be some function for SpectConnect, even if the creation of a truly complete reference library would serve to make it unnecessary in many situations.

6.3 Truly "metabolomic"?

Of course, this idea begs the question, "Can we really perform a truly metabolomic analysis at all?" More specifically, one may wonder if, given our hyphenated separation–characterization approach (GC–MS, LC–MS, etc.), it is possible to detect all of the metabolites in a sample in just one assay. It seems that the answer to this question is almost certainly "no". For GC–MS, the reason is obvious: in order for the metabolites to elute on the gas chromatograph, they must be sufficiently volatile. As most metabolites are not very volatile, the first step in metabolomic GC–MS analysis is to derivatize the sample, which changes certain functional groups on molecules to other functional groups that drastically increase the vapor pressure of the individual compounds. This "volatilization" is limited only to the classes of compounds containing specific functional groups that one's given derivatization agent of choice can transform, whether they are hydroxyl groups, amine groups, or some other groups. Thus, in each assay, we are limited to only analyzing the subset of the metabolome that has at least one of the functional groups that we can derivatize. In LC–MS, analysis is restricted based on the phase of the sample: one will take either the polar or the nonpolar portion of the sample and use the appropriate column for an assay (the two do not typically use the same column). Thus, in this case, we are limited to only polar or nonpolar metabolites in any given assay. Similar arguments can be made for most hyphenated metabolomics approaches.

To this end, we acknowledge that given current technologies, a truly metabolomic

single assay is just not feasible. We then turn our focus towards at least trying to capture as much as we can with one technique — when discussing GC–MS, one could go so far as to refer to it as a "derivatizome", though that will likely be the word's only usage in this document. Specifically, we will try to characterize all of the polar phase metabolites that could possibly be detected using TMS derivatization and a GC–MS.

6.4 Our model system and approach

We chose to use *Saccharomyces cerevisiae*, or baker's yeast, for our investigations. We chose yeast for a number of reasons. First of all, it is one of the most well–studied systems, so experimental protocols are well–defined and its physiology and metabolism are also both relatively well–defined. Second, as a unicellular organism, it is a relatively simple system to study (compared to multicellular organisms with different cell types that may have different typical metabolomic profiles). Third, as a eukaryote, knowledge gleaned from our work is more likely able to be applied to more complex and interesting systems than one might expect from a prokaryotic cell. Fourth, *S. cerevisiae* is an industrially relevant organism, such that as engineers we may able to leverage our findings into a practical application with substantive impact. One last reason for using yeast is the relative robustness of *S. cerevisiae* to culture conditions compared to some other common systems (e.g., *Escherichia coli*). This last reason is just a bit of practicality: given so many different possibilities for our model system, we may as well choose one that is easier to grow.

Our approach, then, is to perturb these yeast cells in as many ways as we can think of (and as is practical) and use SpectConnect to analyze the results. By doing this, we believe we will be able to characterize all of the metabolites that can be detected by GC–MS given our derivatization chemistry.

First of all, the use of SpectConnect is a necessity in this project. We will be looking at hundreds of relatively small peaks in tens to hundreds of different experimental conditions (and thus tens to hundreds of different samples' chromatograms). As noted

earlier, we expect that many low–concentration metabolites will not have easily accessible reference spectra. In Chapter 5, it was pointed out that of the estimated 600 metabolites in *S. cerevisiae*, only about 200 of them can actually be purchased [46]. Tracking only metabolites with known standards, then, is unacceptable. In order to distinguish the legitimate metabolite peaks from the noise peaks across these numerous metabolomic analyses, it will be necessary to use SpectConnect to track peaks by retention time and mass spectrum rather than by name. Thus, we will need to have multiple sample or injection replicates of our cultures. Finally, it is worth noting that even if we did have all of the desired metabolite standards for known metabolites, it would still be worthwhile to use SpectConnect in case there are some relatively unknown metabolites in obscure pathways. Given the host of hypothetical and unannotated genes in the *S. cerevisiae* genome, it is reasonable to expect that not every possible metabolic reaction has been discovered; our approach is one reasonable way of finding those potentially unknown metabolites and reactions.

Perturbing the yeast cells serves to change metabolism such that we expect to detect metabolites that would otherwise slip under the detection limits of the instrumentation. We will need to use both genetic and environmental perturbations for these experiments. As we perform more perturbations, SpectConnect will have detected more unique metabolites across the different samples; this will enable better tracking of low–concentration metabolites in samples where they may only be detectable in a small number of sample replicates.

Environmental perturbations help to test metabolic response to a variety of feedstocks, additional chemicals, or external conditions (like heat or salt). Some metabolic pathways may only be active in certain conditions; others may have some basal level of activity so low that most of the metabolites in the pathway are not usually detectable. By forcing these pathways to become active, we are more likely to detect these otherwise low–concentration metabolites.

Genetic perturbations will provide a broader spectrum of information than can be obtained by environmental perturbations alone. Appropriate genetic perturbations include gene knockouts, gene overexpressions, and potentially even gene knockdowns

(for critical metabolic enzymes). If, for example, a gene towards the end of a pathway is knocked out, we expect some degree of buildup of the immediately upstream substrate metabolites that can now no longer be converted to their ultimate target. Such cases will provide the ability to characterize and track those metabolites even if they are normally at extremely low concentrations in wild–type strains. Other knockouts may prompt a sort of metabolic "rewiring" to circumnavigate the eliminated gene and highlight pathways that otherwise might not be easily seen. In total, then, genetic perturbations of our cultures represent a powerful tool to explore as much of the metabolomic space as possible.

Of course, the power of genetic perturbations does not obsolete the utility of environmental perturbations. We expect that some metabolic changes may only (or best) be effected by environmental perturbations. Environmental perturbations can easily affect multiple enzymes or transcription factors simultaneously, so in this sense they offer an almost uniquely multifaceted approach to perturbations. In addition, environmental perturbations are extremely easy to create; genetic perturbations are much more time–consuming. In this way, it makes sense to pursue as many environmentally perturbations as possible since they are the simpler, cheaper, less time–consuming alternative.

Perturbations will be selected based on knowledge of yeast metabolic pathways. Initial perturbations will be relatively simple in an effort to start accumulating most of the "obvious" metabolites. We will then survey the metabolites that SpectConnect has characterized to see how many of them have known reference standards. These now completely "known" metabolites can be used as a reference point on the metabolic map: if we have already detected all of the metabolites in a given pathway, then a knockout in that pathway may not be useful since we don't expect the accumulation of metabolites that we have not yet detected. Conversely, pathways that seem to have little coverage in the current dataset are more promising targets for gene knockout experiments and other genetic perturbations.

We expect that eventually the number of unique metabolites will asymptotically level off. In theory, each new perturbation is meant to detect new metabolites. Even-

tually, we will have found most of the metabolites that are in the cell, and it will be increasingly harder to detect new ones. Since SpectConnect is not prone to accumulating many noise peaks (since each peak must be well–conserved within sample replicates), we then expect that eventually we will perform multiple perturbations that do not identify any previously unseen or uncharacterized metabolites. This would then be the practical stopping point for our experiments. Since we would never know if the library of metabolite spectra was totally complete, we would never be certain that another experiment would not allow us to detect a previously unseen metabolite. However, if our total number of metabolites as a function of the number of perturbations seems to reach an asymptote, we can reasonably expect that we have found most of the metabolites.

6.5 Preliminary analysis

Before beginning experiments, we performed some preliminary analysis with SpectConnect to identify the minimum number of sample replicates desired and to get a sense for the sensitivity of the results with respect to this number. To do this, we analyzed a dataset previously provided by Dr. Henri Brunengraber [32]. This dataset was derived from 10 replicates of ethanol–perfused rat liver. We applied SpectConnect to each possible subset of replicates with cardinality of at least 2; by doing this, we were able to identify essentially a "distribution" of possible conserved metabolite profiles that could be obtained by any random subset of any given size. These more or less amount to bootstrap distributions for each possible number of samples analyzed, thus characterizing the inherent uncertainty in our data based on the finite size of our sample set.

The results of this analysis are seen in Figure 6-1. From this figure, it seems that using only three or four replicates provides an overestimate of the number of well–conserved metabolites (which we assume to be defined by the metabolites found in the complete dataset, with 10 replicates). The non–monotonous nature of the curve is not surprising when one considers the threshold values that SpectConnect uses. These

samples were analyzed with all default values, meaning that the "support" threshold (minimum number of samples in which a metabolite peak must be present in order to be tracked) was 75%. Fractional values are rounded to the nearest whole number, such that sometimes when one replicate is added to the analysis the required support does not increase. For example, if we have six sample replicates and added a seventh, the required support at the 75% threshold would increase from 4.5 to 5.25. Rounded to the nearest whole number, both of those values is 5. Thus, due to the discretization in the small number of sample replicates, the amount of permissible noise in the data increases. We would then expect to find more conserved metabolites: some of these are legitimate metabolite peaks that were right at the threshold of detection and thus were sometimes just below that threshold, while other peaks may be artifacts of noise. Generally speaking, one would expect relatively few noise peaks at such high numbers of replicates — it ought to be difficult for a "noise" peak to occur in five replicates when there are only seven total — so we presume that most of the metabolite peaks added are just low–lying peaks.

Additionally, we see that SpectConnect is only somewhat sensitive to the number of replicate samples that are analyzed. As we increase from five to ten replicates, we see that the total number of metabolites found does not change drastically; the total variation is less than 15%. When we factor in the confidence intervals, we see that there is almost no statistically significant change as the number of samples is increased. Those confidence intervals are significantly (and monotonically) decreased, though, by adding additional sample replicates. This means that for any possible subset of replicates, the largest and smallest possible number of metabolite peaks that would be found is much closer to the true expected value when more sample replicates are used; we can then have more confidence in those values. Additionally, we see that there is a relatively consistent decrease in the number of metabolites that are identified as sample replicates are added (except at the previously mentioned discontinuity). This means that we can have more confidence in the metabolite peaks found when there are more sample replicates as the number of noise peaks is decreased; however, this also likely means that we are slowly losing the ability to track low–concentration

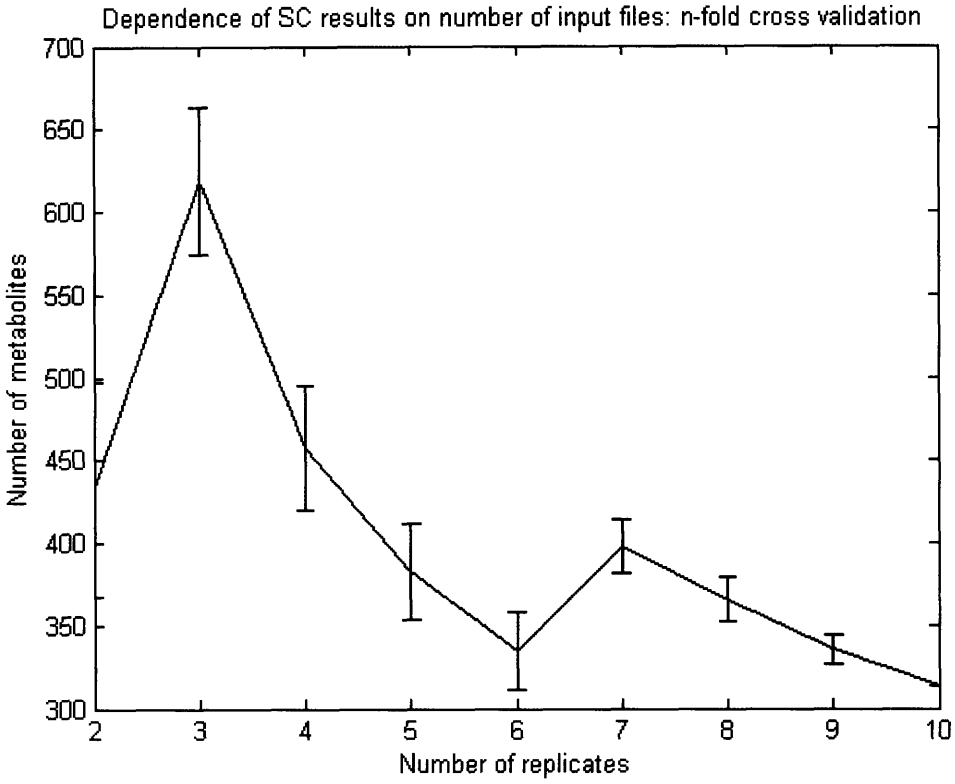


Figure 6-1: A bootstrapping attempt to characterize the inherent error in SpectConnect due to a finite dataset sample size. For the total dataset obtained from ten sample replicate GC–MS injections of ethanol–perfused rat liver obtained from Dr. Brunengraber[32], we performed SpectConnect analysis for each possible subset of replicates with cardinality of at least 2. The mean number of metabolites found for each possible subset size is represented by the blue line, with 95% confidence intervals represented by the error bars. The discontinuities in the otherwise monotonic curve are described in the main text. Overall, we see that as more replicates are used, the variation due to the finite sample set size decreases. We also see that the total number of metabolites typically also decreases, until it converges on the well–conserved set of metabolite peaks found by analysis of the whole dataset. We expect that some of the metabolites lost between having five and ten replicates are low–concentration metabolites that may sometimes be just below the threshold of detectability for our analytical methods, making them untrackable given our requirements for tracking metabolites.

metabolites which may be obscured by instrumental or analytical noise. It seems that given these phenomena, we will likely have to make some tradeoff between specificity and sensitivity.

From the information gleaned from these preliminary calculations, we can also analyze the chances that SpectConnect is finding "false" peaks. This can be seen by visualizing the data in a slightly different way, as seen in Figure 6-2. Here, we compare the number of peaks found as we add replicates one at a time. For instance, with only two replicates (the red x at 2 on the x–axis), there are about 425 peaks found, whereas with three replicates there are about 575 peaks (as seen by the blue circle at 2 on the x–axis). The solid black line represents the total number of peaks found between the two sets of replicates; that is, when SpectConnect is run with one "condition" having two replicates and the other "condition" having those two replicates plus another, the black line represents the total number of metabolite peaks in the non–redundant library created from analyzing both conditions. If adding additional sample replicates were causing noise to enter our analyses, we would expect either that the black line would be above both the red x and the blue circle, or that we would see erratic behavior in the black line. The former possibility would mean that certain disjoint subsets of metabolites were being tracked, while the latter would indicate overall, non–deterministic sensitivity of the method to the number of samples analyzed. Instead, we see that the black line tracks almost exactly with the greater of the red x or blue circle, meaning that the addition or subtraction of samples is only creating subsets and supersets of metabolite peaks; that is, if unique metabolite peaks are found in one set of samples, no unique peaks are found in the other, making its peaks a subset of the other samples' peaks.

From all of this information, we can make a few generalizations about the results that we expect SpectConnect to provide in our experiments. First of all, we expect that the number of sample replicates will have a relatively weak effect on the overall results of our analysis once we have at least four or five sample replicates. We can set whether we are more or less likely to find low–lying metabolite peaks by choosing an appropriate number of sample replicates; for instance, seven replicates ought to lead

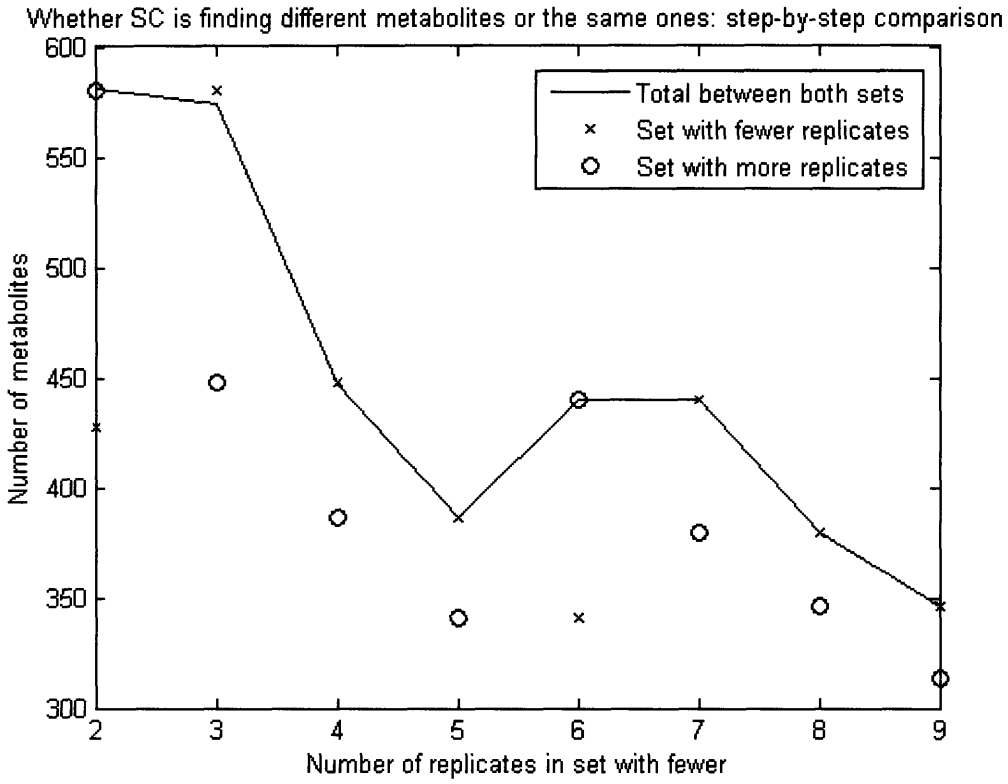


Figure 6-2: A stepwise attempt to determine the robustness of the actual metabolite peaks identified by SpectConnect. Here, we are not as interested in the total number of metabolites identified as much as the overlap between different datasets analyzed by SpectConnect. For each point on the x–axis, SpectConnect was run with x sample replicates. The number of metabolites found in that analysis is denoted by the red x. The same x metabolites, plus 1 more, are also run as a separate "condition" in the same SpectConnect run. The number of metabolites found in that analysis are denoted by the blue circles. The library comprising all of the peaks in both conditions is denoted by the black line. Since there is no erratic noise in the black line, and the black line almost always tracks the greater of the blue circle and red x, we infer that the actual metabolites being tracked as the number of replicates increases are robust, even if the total number of metabolites tracked tends to vary.

to the discovery of more low–lying peaks than six replicates would reveal. (Of course, using seven replicates would likely entail either some sort of asymmetric or unusual experimental protocol, or explicitly leaving out portions of datasets, which would raise even more questions about the best way to analyze the data.) We also have quite a bit of confidence in the robustness of the actual peaks discovered — as opposed to the number of peaks discovered — as a function of the number of samples that are used. We do not expect the sets of peaks found to change drastically, though the total number of peaks may vary a little bit. Altogether, it seems that SpectConnect is a promising tool to perform our goal of characterizing and tracking all of the chemically accessible metabolite peaks in an organism.

6.6 Experimental plans and difficulties

Initial experimental plans and timelines called for a combined experimental effort with Jason Walther. While I had no previous experimental experience in our group's work, I was slated to help out as an extra pair of hands for complicated techniques and otherwise to relieve some of the experimental burden from Jason so that this project would not interfere with his primary thesis objectives. He spent some time training me, and together we began developing an optimized protocol and workflow. We estimated that the experiments would be sufficiently low–maintenance that the entire workflow, from cultures to GC–MS injection, could be done over a long period in our spare time.

As we developed our protocol and workflow and began preliminary tests, we found a few unexpected circumstances. First of all, the entire process was much more time–consuming than we had expected. Among other confounding issues, different environmental perturbations caused different growth rates, meaning that it was difficult to get experiments synchronized and thus more time–consuming to quench and process the samples. An alternative would be to freeze quenched samples and accumulate them for later batch processing, but we found that there were numerous bottlenecks in our experimental workflow that precluded the processing of more than about eight

samples (three biological replicates and a control). While this approach wound up being more efficient in terms of time, it still amounted to at least a week of working time per experimental perturbation. Finally, some difficulties with our instrumentation sidetracked us for weeks, ultimately leading us to wait for a new instrument to arrive. By the time this new instrument had arrived and was running, a few months had been lost. At this time, I was anxious to get some of this work started, while Jason had accumulated other important priorities that needed to be addressed, so I began the experimental work on my own. The results presented here are the beginnings of this work.

6.7 Experimental results

Only environmental perturbations were addressed in this preliminary work. Both glucose and galactose were used as carbon sources for the cultures. Glycerol was another possible simple carbon source, but I was unable to get the cultures to grow to any significant density using glycerol as a sole carbon source, so I eventually abandoned those attempts. Two different environmental perturbations were analyzed: heat shock and hyperosmotic shock. Two different time courses were performed for each perturbation. For heat shock, the temperature of the fermentations was raised to 37°C. In one time course, this was done by merely transferring the cultures to a warmer orbital shaking incubator. In the other time course, samples of the cultures were added to medium already at the target temperature. For hyperosmotic shock, the total salt (NaCl) concentration was increased to 1 M. In one time course, this was done by adding concentrated salt to the cultures; in the other, samples from a preliminary culture were added to prepared medium such that the total concentration of salt would be 1 M. The time points analyzed were at 0, 15, 30, and 75 minutes after the perturbation. Unfortunately, at least one set of samples was lost due to accidentally being thawed. (Detailed protocols for experimental sampling and preparation are included in Appendix B.) The total number of metabolites found in each sample condition is seen in Figure 6-3.

The control conditions in this figure have a surprisingly large number of both total and unique metabolite peaks. Deriv–1 and Deriv–2 are controls that contained only derivatization reagents. Gal–C and Glu–C are controls that contained only medium and derivatization reagents. These may have a high number of tracked metabolites for a number of reasons. For the derivatization controls, I ran fewer sample and injection replicates than for the other samples, so there may be more noise peaks. Alternatively, since there are fewer compounds in these samples, it is possible that the baseline and other noise may be slightly lower, and thus the low–lying peaks reflective of those derivatization reagents may be seen. For the glucose and galactose media controls, similar reasons may explain the large number of both total and unique metabolite peaks detected by SpectConnect.

The remaining eight conditions have trends more in line with expectations, though with some deviations. The number of metabolites detected, typically on the order of 100, seems reasonable. The number of unique metabolites detected is typically rather small; this would be expected, as the sample conditions analyzed are not sufficiently diverse that we would expect a significant number of metabolite peaks found in only one condition. On the other hand, the number of metabolite peaks found in the cultures 75 minutes after perturbation was surprisingly small. This may be due to the perturbations causing the cells in the cultures to die, though that ought not to have been the case, as the perturbations were not so severe that they should have evoked such a response. It is probably more likely that this is due to experimental error, whether in the culture conditions or in the sample preparation. This experiment has not been repeated to confirm the source of this anomaly. The only other somewhat anomalous result is the fact that the time 0 sample for both heat shock and salt shock is surprisingly dissimilar from the glucose sample (which should, in principle, not be any different). Again, this may be due to experimental error. Another likely source of this variation may be where the two cultures were in their growth curve when they were sampled: one may expect that the metabolic profiles of a culture still in exponential growth phase and one in stationary phase would be quite different.

Another way of viewing this data is seen in Figure 6-4. This figure shows the

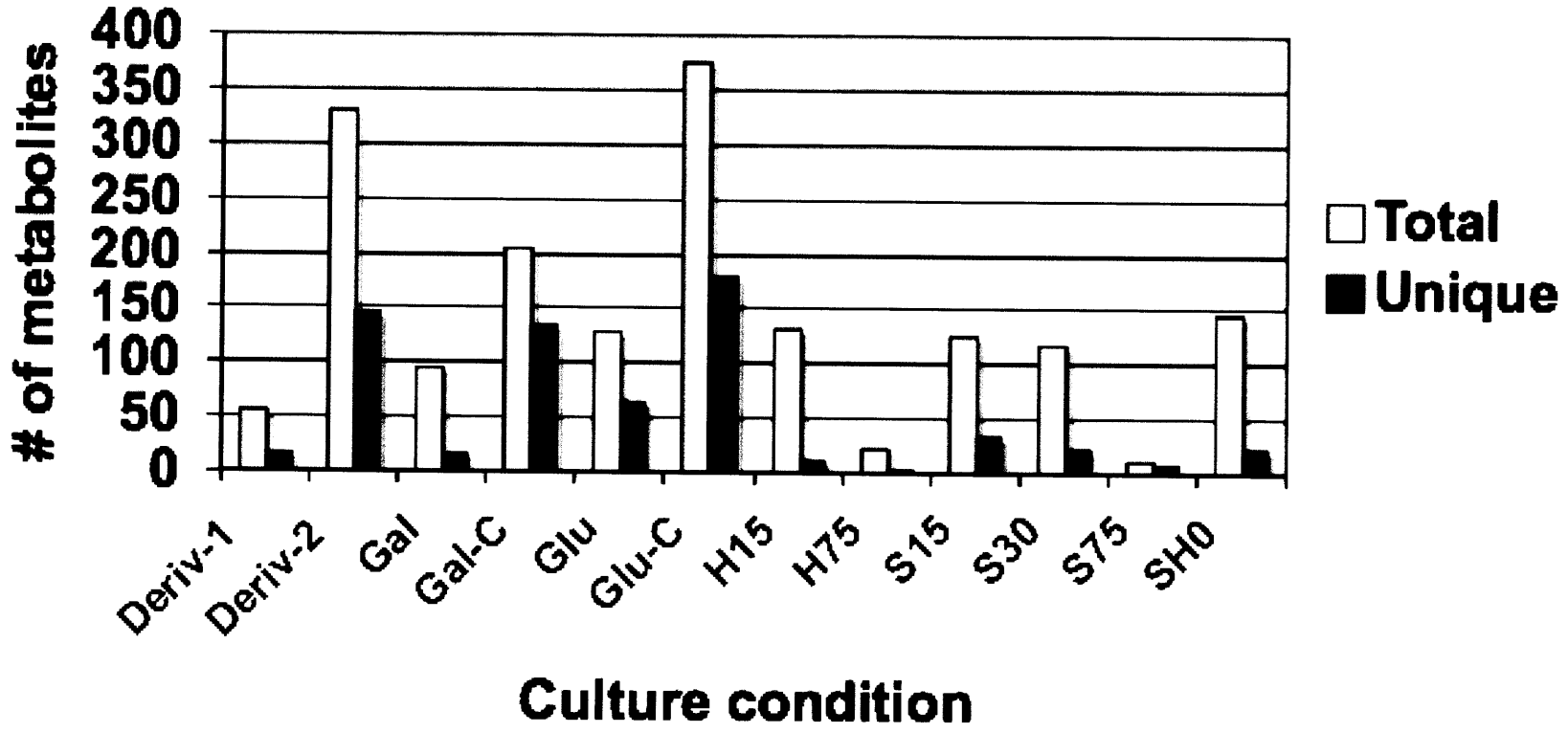


Figure 6-3: Metabolites tracked by SpectConnect. Deriv 1 and Deriv–2 are controls containing only derivatization reagents. Gal and Glu are cultures grown on galactose and glucose medium, respectively. Gal–C and Glu–C are controls containing culture medium and derivatization reagents. H15 and H75 are samples taken 15 and 75 minutes (respectively) after heat shock at 37°C. S15, S30, and S75 are samples taken 15, 30, and 75 minutes (respectively) after hyperosmotic salt shock. SH0 was the culture from which the heat and salt shock time courses were taken, before perturbations. The white bar represents the total number of metabolites found in the condition, while the dark bar represents the number of metabolites found only in that condition.

number of metabolites that are found in a given number of sample perturbation conditions. The fact that no metabolites were found in seven or eight conditions is very surprising, as we would expect a core set of metabolites to be present at detectable levels in all perturbations; this may be accounted for by the perceived error in the 75 minute time point samples. That leaves very few metabolites present in five conditions and none present in all six of the remaining conditions. Again, this is disheartening, but may be explained by the other perceived errors discussed above. This distribution is surprisingly weighted towards metabolites present in only one condition, but this may be partly explained by the issues previously discussed.

This overlap can be seen more clearly in Figure **6-5**. Here the correlations are quite evident: cultures from time course samples 0, 15, and 30 are extremely similar. The potentially "dead" cultures at the 75 minute time point have virtually nothing in common with any of the other cultures. What is most surprising is that HS0 has very little in common with Gal and Glu, which each in turn have very little in common with each other. Other than potential growth phase differences, one would expect HS0 and Glu to be very similar, as they are essentially identical cultures. Analysis of HW0 was performed at the same time as the 15, 30, and 75 minute time points, while the Gal and Glu samples were analyzed some time prior to that. In the intervening time, quite a few problems with the GC–MS instrumentation had been discovered and quite a bit of routine maintenance had been performed; if some of the subsequently identified GC-MS problems were present during the analysis of the Glu and Gal samples, this may explain the lack of correlation with the other samples. As noted already, though, the consistency between the different time points (other than 75 minutes) is encouraging for our potential reproducibility. The culture for the 30 minute time sample was performed a few weeks prior to the other cultures, meaning that even across some biological variation we are still able to capture correlations and similarities in metabolomic profiles.

The character of all metabolites found can be qualitatively seen in Figure **6-6**. Clearly, quite a few peaks were found in both the derivatization and culture medium controls. These peaks had relatively little overlap with the peaks identified from yeast

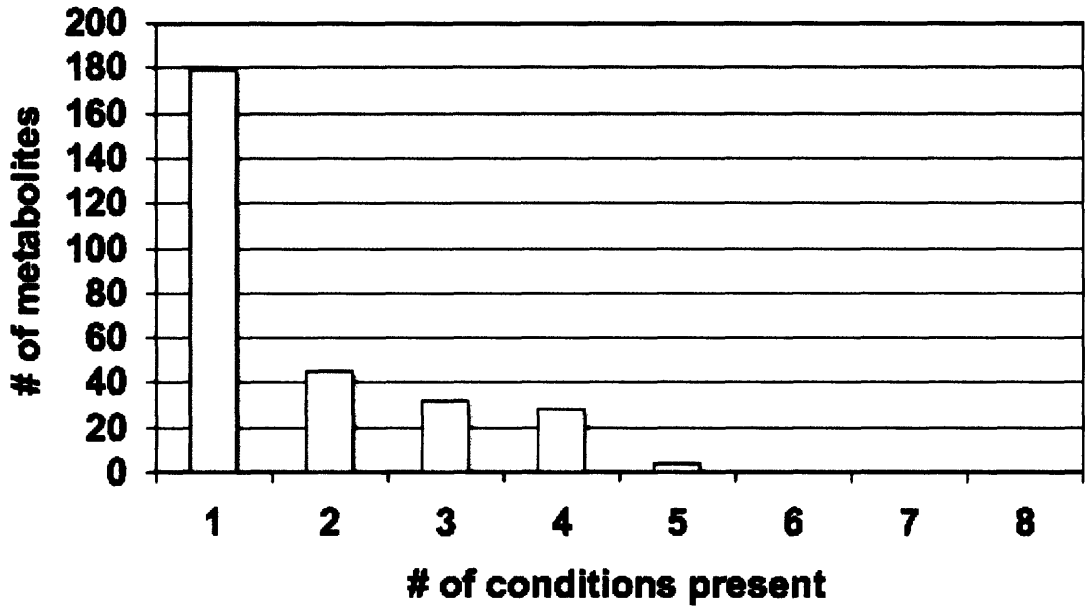


Figure 6-4: Histogram of metabolite frequency across the eight non–control conditions in Figure 6-3. The number of metabolites present in x conditions is plotted on the x–axis. This distribution is surprisingly weighted towards unique metabolites, a phenomenon discussed in more detail in the main text. The number of metabolites found in 2, 3, and 4 different conditions is encouraging given the perceived accuracy of some of the experiments performed.

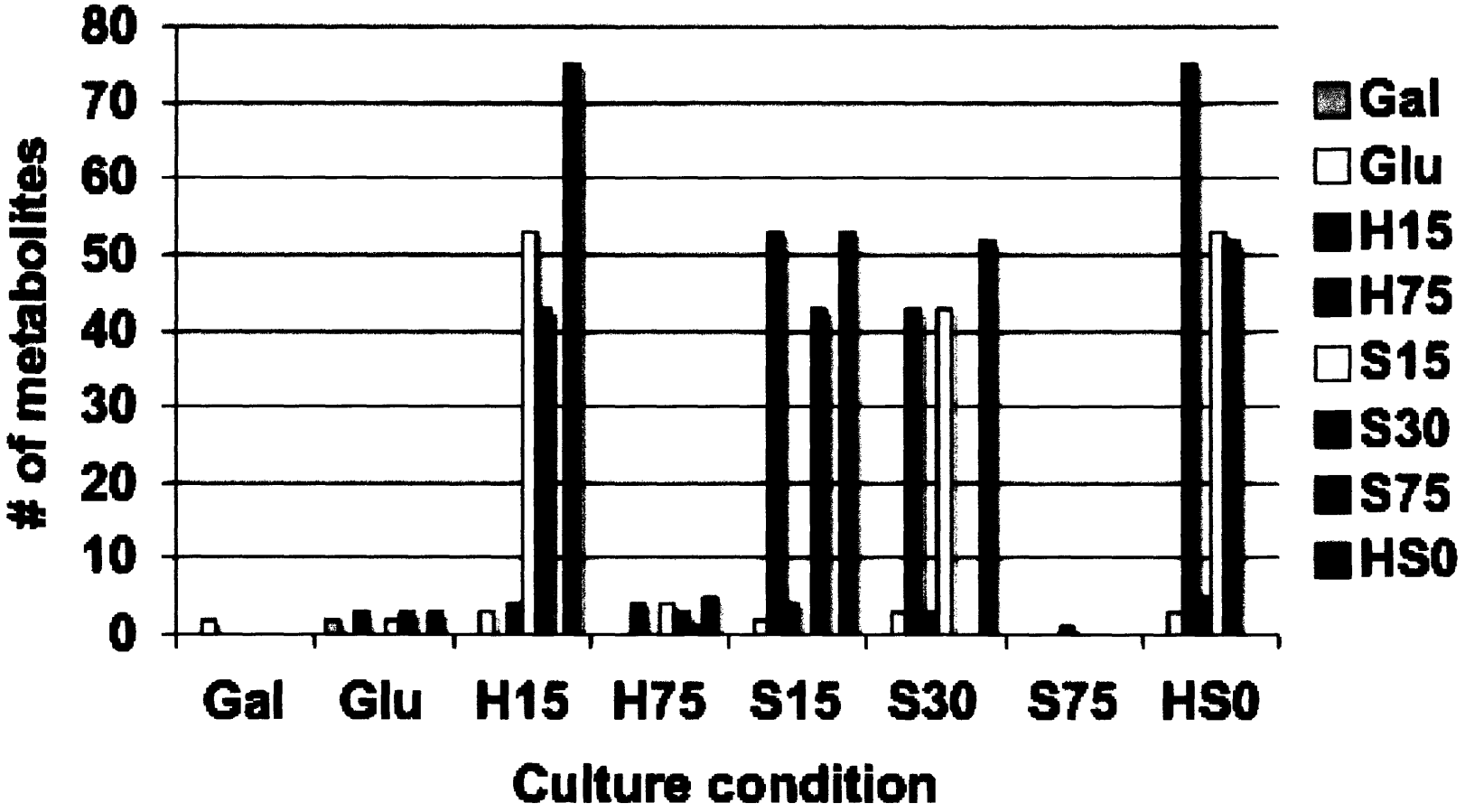


Figure 6-5: Histogram indicating the correlations in metabolite profiles across multiple environmental perturbation conditions. For each culture condition across the x–axis, a bar is drawn indicating the number of metabolites overlapping between that condition and another condition. (Self–overlap has been removed from this figure.) Condition labels are as in Figure **6-3**. One can see a clear correlation between H15, S15, S30, and HS0, as should be expected. S75, H75, Gal, and Glu seem to be outliers. Looking at Figure **6-3**, it is clear that the lack of overlap in H75 and S75 may simply be due to a decreased number of metabolites detected and tracked, while the reason for Gal and Glu being so disjoint from the remainder of the conditions remains somewhat unclear.

cultures, with only about 25% of the metabolite peaks from cultures falling in that category. A core set of metabolite peaks was identified in both controls and culture samples, while additional metabolite peaks were identified that were in both controls but not in any culture samples. Were we more confident in this experimental data, we would then hypothesize that we had enumerated 293 different metabolite peaks that can be characterized using TMS derivatization for GC–MS analysis of the yeast metabolome. In reality, though, I would imagine this number to be closer to about 100 judging from the time–course results in Figure 6-3.

6.8 Conclusions

The results of this preliminary study indicate that our approach is promising but that some experimental difficulties must be worked out. As noted above, the workflow for this process has yet to be optimized for a high–throughput approach. Without such optimization, performing the desired study would take an intimidatingly large amount of work. In addition, the experiments must be refined so that they are much more repeatable. Obviously we expect significant biological variation from experiment to experiment, perhaps even up to 20% in terms of metabolite concentrations, but the variation between SH0 and Glu (discussed above) is unacceptable. Culture and preparatory protocols must be refined such that the results produced are much more consistent. Despite all of those caveats, this does seem to be a promising approach to explore the metabolome of a species. SpectConnect is well–suited to solving this problem, as demonstrated by our preliminary computational experiments. Our simple metrics of analyzing the resulting data (number of unique metabolites, number of overlapping metabolites between conditions, etc.) are particularly useful for getting an immediate and simple characterization of the trends in the data.

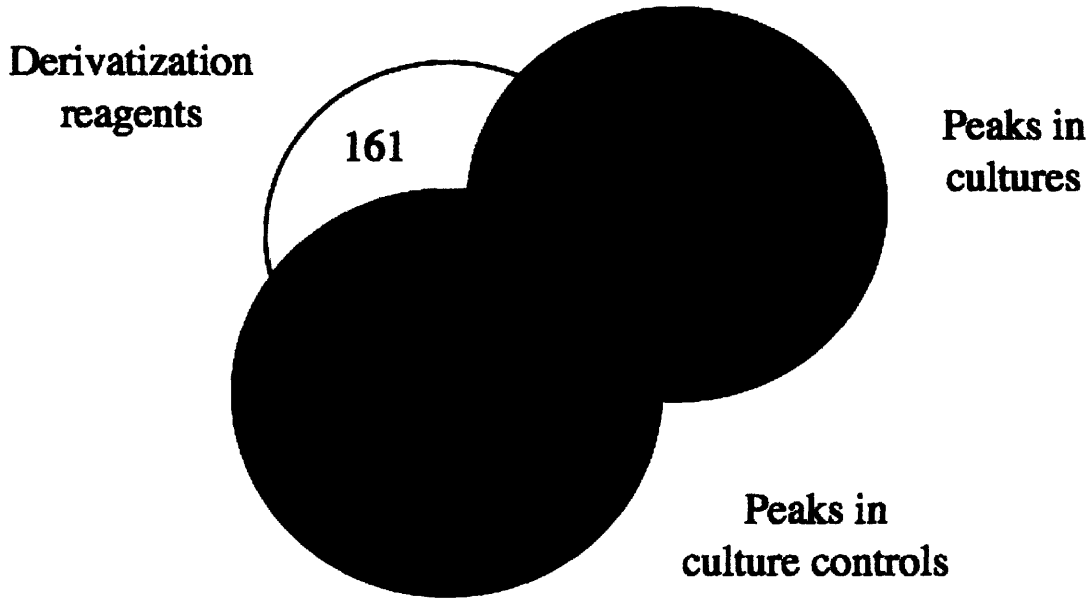


Figure 6-6: Venn diagram for the metabolite peaks found in yeast cultures and both types of controls. A number of peaks unique to cultures but neither control condition may be likely metabolites. Peaks found only in controls are likely only artifacts of sample prep and analytical noise.

Chapter 7

Additional projects in sequential data analysis

In the course of pursuing the main goals of this thesis, some additional interesting projects and opportunities have arisen. These "side" projects will be explained in more detail in this chapter.

Both side projects presented here spring from the investigation of the BLOSUM family of amino acid substitution matrices. The first project describes the discovery of an error in the code used to create the BLOSUM matrices and the subsequent impact that the error has had. The second project describes the imagined "evolution" of the BLOSUM matrices had they been updated to reflect the growth of the database used as the input for their initial creation. This work has been done in collaboration with Kyle Jensen and will soon be under consideration for publication.

Again, the word "we" may frequently used in this chapter to refer to the respective authors for each project and the work that they have contributed to the project.

7.1 A surprising error in the BLOSUM matrices

7.1.1 Overview

The BLOSUM family of substitution matrices, and particularly BLOSUM62, has become a *de facto* standard in protein database searches and sequence alignments. The key insight in their creation was the algorithm to calculate the matrix entries, which used conserved blocks of aligned proteins and limited the impact of highly similar proteins. However, the implementation used to create these matrices had some key inconsistencies with the published algorithm; these inconsistencies have in turn had a profound impact on the BLOSUM matrices used today.

To evaluate this impact, we have used exhaustive pairwise sequence comparison with both BLAST and Smith-Waterman searches against a set of hand–curated structure–based protein homologs to determine a matrix's effectiveness at locating distant homologs. We find that, to a statistically significant degree, the BLOSUM62 matrix used today outperforms the matrix that should have been created. We also find that this performance is dependent upon the order of sequences in the database used to create the BLOSUM matrices. Since the database sequences were ordered based on arbitrary metrics, including alphabetization, a slightly different database or different arbitrary decisions would have led everyone to be searching with an inferior matrix instead of an almost aberrantly superior matrix. We conclude that the surprising performance difference is an unexpected byproduct of scaling and rounding matrix entries. While some inconsistencies have been resolved in subsequent source code releases, the updated matrices have never been adopted. This means that we have been using "incorrect", but better, BLOSUM matrices.

Again, the word "we" will frequently be used throughout this section to refer to the collective authors of this work and their contributions.

7.1.2 Introduction

While many different amino acid substitution scoring matrices have been proposed in the literature, the BLOSUM series [66] of matrices is one of the most widely used. (Overviews of some of the available matrices can be found elsewhere [69, 67, 174].) In fact, one single matrix, BLOSUM62, has become an almost *de facto* standard for default, out–of–the–box sequence comparison and database searches. It is the default matrix for such commonly–used tools as FastA [131], BLAST [10], t–coffee [124], and Clustal–W [163]. Ssearch [129, 130], one of the most commonly–used implementations of the Smith–Waterman [152] local alignment algorithm, currently uses another BLOSUM matrix, BLOSUM50, as its default.

One of the intellectually appealing aspects of the BLOSUM family is that it was created in a straightforward, intuitive way that reflected contemporary knowledge of protein structure and similarity. It was constructed in 1992 from Blocks 5, a database of highly conserved protein regions, or "blocks", derived from families in the PROSITE database [74]. A thorough explanation of the derivation is given elsewhere [66], and a "primer" version of the explanation has also recently been given [49]. In short, the blocks were used as a training set to derive a set of implied target frequencies with which an amino acid of one type can ably substitute for another amino acid. To minimize increased impact of highly homologous sequences, all block members that met some threshold for percentage identity were clustered together. These clustering thresholds gave rise to the names of the matrices — BLOSUM62, BLOSUM50, etc. — and helped distinguished the relative level of similarity between sequences that each matrix was tuned to detect.

In the course of looking into the potential impact of updated Blocks releases on the BLOSUM matrices that could be made from them [159], we noticed some anomalies in the code used to create the initial BLOSUM family of matrices. While some of these anomalies have been corrected in subsequent versions of related programs by the same authors [70], it is interesting to note that the matrices in common use were never updated accordingly, so few people are aware of the anomalies or the impact

that they may have had on the substitution matrices that we use on a daily basis. In the following sections, we will detail the anomalies and analyze the impact they had on the BLOSUM matrices by analyzing the effectiveness of a variety of matrices at detecting distant homologs.

7.1.3 Methods

Matrix creation

Matrices were created using the methods described in the original BLOSUM manuscript [66]. The programs and Blocks training data used to create the original BLOSUM series of matrices were obtained from `ftp://ftp.ncbi.nih.gov/repository/blocks/unix/blosum/blosum.tar.Z`. Additional, updated programs were obtained from `ftp://ftp.ncbi.nih.gov/repository/blocks/unix/blosum/programs`, while Blimps programs were obtained from `ftp://ftp.ncbi.nih.gov/repository/blocks/unix/blimps/`. All matrices were created using the same Blocks 5 release available when the BLOSUM matrices were initially published.

Sequence datasets

We used the ASTRAL database [30] as the database for our searches. Our method was designed to emulate the work by Price *et al.* [136]. ASTRAL is created based on the SCOP database [117], which classifies proteins based on their function, structure, and sequence into a hierarchical structure of classes, folds, superfamilies, and families. Sequences in the same superfamily can have low sequence similarity, but are likely to have a common evolutionary origin based on their structural and functional features. Because these classifications are made by human inspection, not via automated sequence alignment procedures, it makes an ideal "gold standard" for remote homolog detection tests.

From the full set of ASTRAL genetic domain sequences, we chose the sequence set from which 40% identical sequences had been eliminated. By using this subset, our search focuses on the detection of remote homologs that are more challenging

for substitution matrices to discover and thus will differentiate the abilities of the respective matrices to find distant relatives. The sequences were further filtered by pseg [181] for the removal of low–complexity regions. The unfiltered sequence set is available on–line from the ASTRAL database at `http://astral.berkeley.edu/scopseq-1.69/astral-scopdom-seqres-gd-sel-gs-bib-40-1.69.fa`. This non–redundant set numbers 7290 sequences. Each sequence was extracted from the database one at a time and used as a query for the entire database. Search results in the same superfamily as the query were considered to be true positives.

Search methods

We used both the Smith–Waterman [152] local alignment algorithm and BLAST [10] for searches against the databases. In particular, we used the ssearch implementation of the Smith–Waterman algorithm [129, 130] and the NCBI version of BLAST in the C Toolbox, obtained from `ftp://ftp.ncbi.nlm.nih.gov/toolbox/ncbi_tools/`, which gives the same results as the commonly–used web interface at NCBI. While Smith–Waterman is an exhaustive, sensitive search that may give a more accurate assessment of a given matrix's performance capability, BLAST is a faster search that is almost an "industry standard" and so better represents the impact of different matrices on the "average user". In addition, BLAST can be run in an ungapped fashion to eliminate the effects of gap penalty parameters.

For our Smith–Waterman database searches, we mostly used the default parameters of the ssearch program, with one notable exception: we varied the gap initiation (existence) penalty from -6 to -14 and the gap extension penalty from -1 to -2 for the initial and revised BLOSUM matrices we investigated. (In general, we expect relatively broad maxima of approximately optimal penalty values, as seen previously [60].) For matrices with different scales, we changed the gap parameters accordingly.

For our BLAST searches, we varied the gap penalty parameters as in the ssearch sections. Appropriate λ, K, α, and β parameters for significance estimation were calculated using the island method described elsewhere [126, 9]. Calculations were

performed using routines provided by Stephen Altschul [7]. Other than gap penalties, all other default parameters for the toolbox version of BLAST version 2.2.13 (12/6/2005) "blastall" were used. Notably, this does not include composition–based adjustments of E–values, which are now used by default on the NCBI web interface of BLAST. Previously [146], these adjustments have been shown to actually decrease performance in individual protein database BLAST searches.

Evaluation of results

We used the Bayesian bootstrap method to evaluate the statistical significance of the mean difference in coverage between any two substitution matrices in our ASTRAL–based tests [136]. This method uses coverage vs. errors per query as a means to evaluate the effectiveness of different substitution matrices, where coverage is defined simply as the fraction of true positives found at a given errors per query threshold. Concerted Bayesian bootstrapping is used to determine the statistical significance of the difference in matrices' effectiveness by evaluating whether slightly different reference databases would have yielded different coverage vs. errors per query curves. True positives were identified as described above. The best–performing gap parameters for each respective matrix were used to compare performance.

It is worth noting that the most appropriate way to compare two families of matrices is via entropy analogues. A matrix's relative entropy is a reflection of the required minimum length of homology necessary for a potential "match" or "hit" to be distinguishable from noise [6]. Comparing matrices with different entropies would be an inappropriate comparison as the matrices could be tuned to find homologous regions of different lengths. By comparing matrices with the same relative entropy, we can better assess the "value" or "correctness" of the information encoded in the matrices. As such, all matrices in this work that were used to search any database were (unless otherwise noted) chosen to have approximately the same relative entropy as BLOSUM62, which was 0.6979. Previous work [67] has further shown that matrices with entropy values of about 0.7 typically have the best average performance within a substitution matrix family.

7.1.4 Results

The program used to create the original BLOSUM matrices, named `blosum.c`, had a number of different anomalies that caused its calculations to deviate from what they theoretically should have been. The anomaly with the most impact involved normalization of sequence weights in forming "clusters". The premise of the re–clustering step in computing the BLOSUM matrices is that sequences that are highly similar should be downweighted (counted as essentially only "one protein") so that they do not contribute too much to the substitution frequencies. This insightful step is the linchpin behind creating a family of matrices reflecting various evolutionary distances. As it turns out, this step was not implemented in the program exactly as it was described in the published algorithm.

When two sequences are compared to find their substitution frequencies, weighting consistent with the described algorithm would require that the contribution of such a pair be divided (normalized) by the size of the two clusters to which the sequences have been assigned, so that the overall contribution of each cluster is equal to just "one protein". However, as implemented in the program, the weight is normalized by only one cluster size, not both. As an example, we analyze this vastly simplified block consisting of six sequences of length 3:

```
KLW
KLA
RLW

KKL
VKL

PIL
```

If we were to re–cluster these sequences at 62% identity, then they would be in the three groups as indicated by the spacing above: `KLW` is 67% similar to both `KLA` and `RLW`, so those three form one cluster even though `KLA` and `RLW` are only 33% similar. For the purposes of determining the number of "substitutions" at any given position, each cluster is essentially considered as "one protein". In this example, then, there are only three "proteins" after clustering, meaning there should only be three total substitution pairs per position ("proteins" formed by clusters 1 & 2, 1 & 3, and 2 & 3). However, we should not throw away the information encoded by all of the possible substitutions from the members of the clusters, so we will count substitutions between all sequences that are not in the same cluster and weight them appropriately such that we only have a total of three substitution pairs. This serves to decrease the influence of highly similar sequences that have not diverged much. To perform this weighting, we should normalize each substitution, or pair, by the product of the two cluster sizes.

We will use the first column as an example. Without weighting, there are three K–K pairs, three each of K–R, K–V, and K–P pairs, and one each of R–V, R–P, and V–P pairs. Accounting for clustering, though, each pair would need to be normalized by the size of the clusters from which the sequences used to form the pairs were drawn. For the K–K pair within the first cluster, there would be no contribution (since all of the sequences in the cluster are essentially "one" protein). For the K–K pairs formed between the first and second clusters, there would be 2 pairs of weight 1/6, resulting in a total of 1/3 K–K pairs. Proceeding through the rest of the pairs, the total weights would be 1/6 for R–K, 1/3 for K–V, 7/6 for K–P, 1/6 for R–V, 1/3 for R–P, and 1/2 for V–P, which adds up to a total of three pairs, as expected.

However, in the original `blosum.c` program, only the size of the "second" cluster in a pair was used for normalization. The loop in the program proceeded from the first sequence encountered, so the only normalization would be the size of the cluster of the later sequence. For the case above, the K–K pairs would each be normalized by 1/2 instead of $1/(2 \times 3)$, leaving 1 K–K pair. The rest of the weights are 1/2 for R–K, 1 for K–V, 3 for K–P, 1/2 for R–V, 1 for R–P, and 1 for V–P, giving a total

of 8 pairs. Not only does the absolute number of pairs change from what should be computed, but the relative amounts of pairs change as well. This inconsistency has since been fixed in updated versions of `blosum.c` [70], though we were unable to find any published analysis of its impact on the BLOSUM matrices or any other programs that use the revised matrices. That is, all of the commonly–used search tools that we surveyed use the original, and not revised and "correct", version of BLOSUM62.

Other anomalies in `blosum.c` may also be found, two of which are integer overflow errors. These anomalies are an artifact of specific implementation choices and the programming language used, akin to an odometer "rolling over" after it exceeds the maximum mileage it can track. These two overflow errors and one transposition typographical error were found in the program, though each of these had significantly less impact than the normalization issue. Of these issues, one overflow and the normalization issue were fixed in later releases of the `blosum.c` [70] code in another package. What is most noteworthy, though, is that even though some of these issues were fixed, the BLOSUM matrices in wide use were never changed to reflect these corrections and we were unable to find published analysis of the corrections' impact on the BLOSUM matrices.

One implication of the normalization anomaly in the original `blosum.c` program is that the BLOSUM matrices currently used are not the only matrices that could be created from the sequences in the Blocks 5 database using the same program. Due to the implementation in `blosum.c`, the normalization error causes the resulting BLOSUM matrix to be dependent on the order of each sequence within each block (though not dependent on the order of the blocks overall). We verified this by shuffling the order of the sequences within each block, each time obtaining different matrices than BLOSUM62. (Our revised `blosum.c` code was insensitive to ordering within blocks in identical tests.)

We analyzed two shuffled versions with both ssearch and BLAST, and analyzed another five only with BLAST. Each of these shuffled versions of the Blocks database needed to be re–clustered at 63% in order to make a matrix isentropic with BLOSUM62 (see Methods). Performance was measured by "coverage", or the fraction

of true positive homologies uncovered at a given errors per query (EPQ) rate for a hand–curated gold standard dataset (see Methods). These shuffled BLOSUM matrices frequently showed performances different (to a statistically significant degree as assessed via Bayesian bootstrapping; see Methods) from the original BLOSUM62 matrix. Over the expected optima of gapped BLAST searches, no matrices were consistently better than the original BLOSUM62, and about half of them were consistently worse (Figure 7-1). In exhaustive Smith–Waterman searches, we found somewhat similar results. Both matrices analyzed were inferior to BLOSUM62 over a wide range of EPQ values (see Figure 7-2).

The difference in relative entropies of the matrices re–clustered at 62% is also particularly interesting. We created 500 versions of the BLOSUM62 matrix from 500 shuffled versions of the Blocks 5 database by holding re–clustering percentage constant instead of relative entropy. In none of these shuffled versions was the relative entropy of the 62% re–clustering matrix as high as it was in BLOSUM62. In fact, using the distribution of those 500 matrices as a basis and assuming the distribution to be approximately normal, the p–value for getting a relative entropy as high as that from BLOSUM62 is approximately 1.1^{-5}. This p–value prediction was subsequently verified by the generation of 425400 different matrices in the same fashion, of which only 7 had the same or greater entropy (a fraction equal to 1.6×10^{-5}). This unusually high relative entropy for BLOSUM62 is most likely a (random) function of the ordering of the sequences within the blocks. However, we have been unable to determine just what aspect of the ordering has such a great influence on the matrix's entropy. While changing which cluster size was used for the normalization brought the matrix's entropy down as expected, the resulting entropy was still not as aberrantly low as may be expected. Similarly, sorting the clusters within blocks by size of cluster also has an impact on the matrix's entropy, but not as much as expected: neither sorting could produce a matrix with an entropy that was as much of an outlier as the real BLOSUM62's entropy.

Perhaps the most interesting result of our work is that if the matrices had been initially created as they were intended, they would have been less effective than they

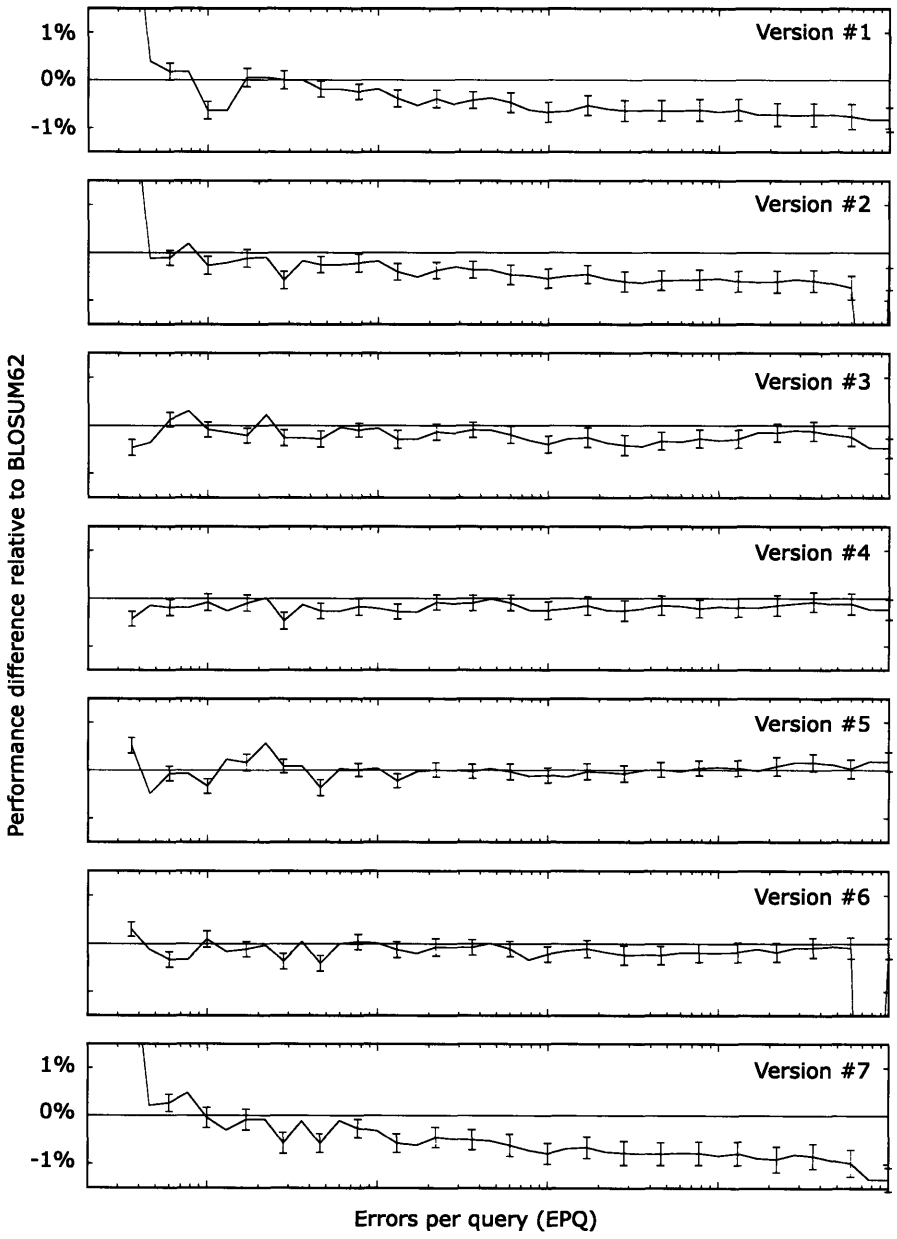


Figure 7-1: BLAST coverage performance difference between BLOSUM62 and matrices formed from shuffled versions of the Blocks database. "Coverage" is the fraction of true positive homologies found at a given errors per query (EPQ) threshold for a hand–curated gold standard dataset (see Methods). The results from each matrix's best–performing gap penalty values were used. Negative values indicate that the original BLOSUM62 is better. Error bars indicate 95% confidence intervals using Bayesian bootstrapping methods. Error bars that do not cross the origin indicate statistically significant differences between BLOSUM62 and the matrix being plotted. Gap extension penalties for all matrices were 1, while existence penalties were 11 for all matrices except for shuffled versions 1, 3, and 7, which used existence penalties of 10.

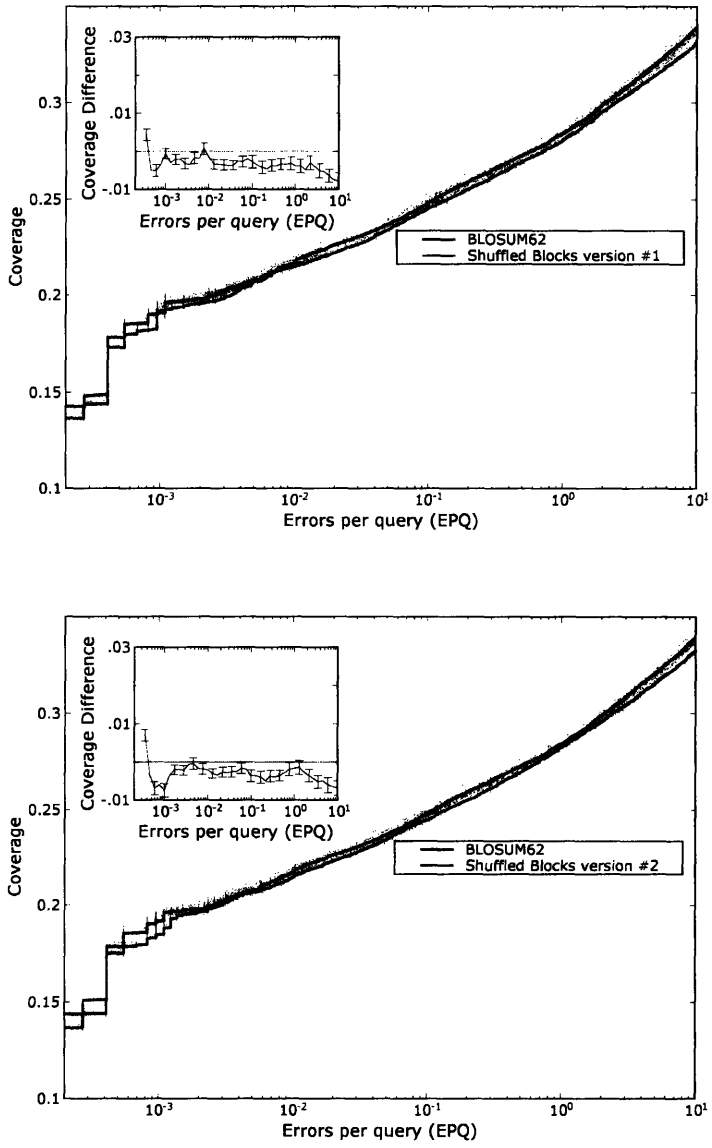


Figure 7-2: A coverage vs. errors per query (CVE) plot of ssearch performance for BLOSUM62 and two isentropic analogs. The two isentropic analogs were formed from Blocks databases where sequences within each block have been randomly shuffled. Thick lines represent the original data, while thinner lines represent individual bootstrap replicates. Note that the sets of lines overlap quite a bit, indicating similar but distinct performance. This is quantitatively displayed in the insets, which plot the difference in performance between BLOSUM62 and each shuffled version. Again, negative values indicate that BLOSUM62 performs better, error bars are 95% confidence intervals calculated using Bayesian bootstrap replicates, and error bars that do not cross the origin indicate statistically significant differences between BLOSUM62 and the matrix being plotted. Results for the best–performing gap penalties are displayed; for all three matrices, the optimal (existence, extension) values are (9,1).

currently are, as illustrated in Figure 7-3. That is, using a matrix created from the revised BLOSUM code (which we will refer to as an RBLOSUM matrix) with the same relative entropy as the original BLOSUM62 matrix, the isentropic RBLOSUM64 matrix performs consistently worse than BLOSUM62. The difference in performance between the two matrices is statistically significant across a wide range of EPQ values and in a wide range of search settings. For Smith–Waterman searches, this difference is seen at almost all combinations of gap penalty values (data not shown). Only for penalty values with extremely poor performances is this trend not observed, and even in these few cases RBLOSUM64 never performs statistically significantly better than BLOSUM62. In gapped BLAST searches, the same trend is observed (data not shown). BLOSUM62 performs better than RBLOSUM64 in ungapped BLAST searches as well (data not shown).

As another way to assess whether the difference between the performance of BLOSUM62 and the RBLOSUM64 matrix is significant, we used the isentropic BLOSUM matrices formed from shuffled versions of the Blocks database to generate a kind of "sample distribution" that represents the distribution of matrices that could have been formed using the same program with only different ordering within blocks. Since the RBLOSUM64 matrix is not order-dependent, we used 10–fold cross–validation to provide a sense of the inherent uncertainty or error in the RBLOSUM64 matrix due to the finite amount of training data available. Even under this relatively harsh method of sampling (compared to merely changing the order, not content, of the database), we see that there are some striking differences between the two distributions (see Figure 7-4). First, it seems that the distribution of performances for matrices based on shuffled Blocks databases is quite a bit larger than the distribution of performances for the cross–validation RBLOSUM matrices. In addition, it appears that BLOSUM62 is not necessarily representative of the "mean" matrix that could have been obtained using the original `blosum.c` code. That is, it performs as good as any of a variety of variants from shuffled Blocks databases, and better than most. Clearly, there is something (at least qualitatively) different between the two implementations, and the deviation of BLOSUM62 from RBLOSUM64 is more than one would expect from the

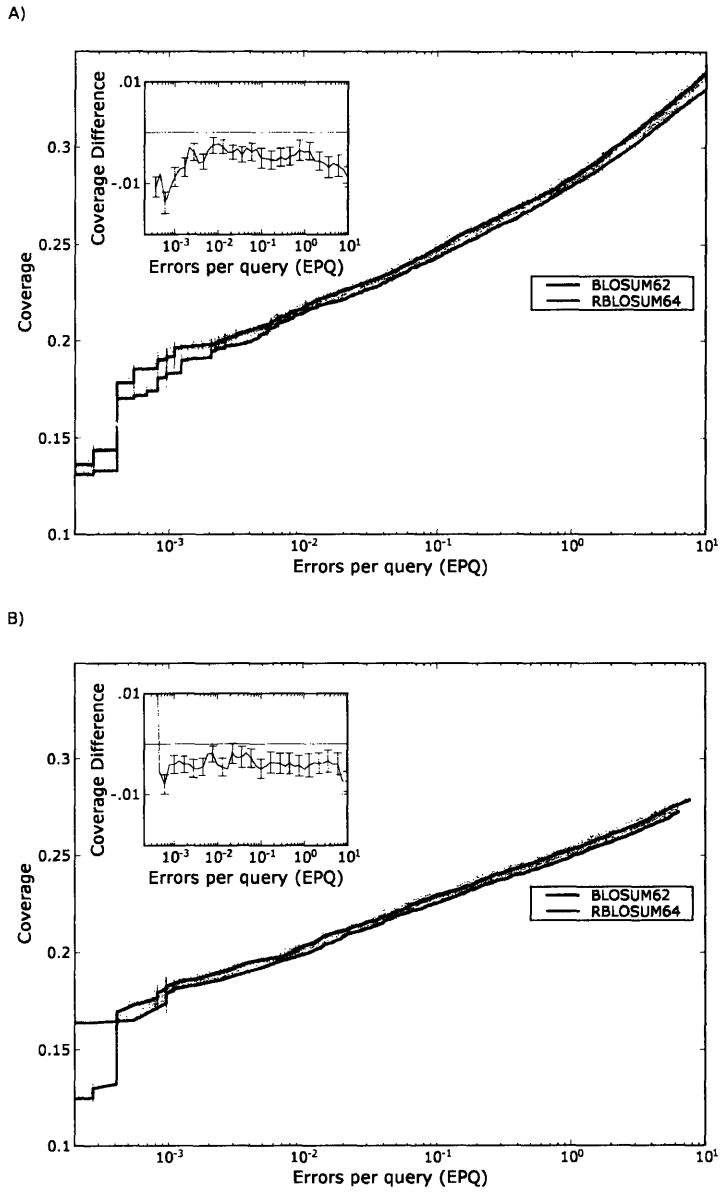


Figure 7-3: CVE plots indicating the (A) ssearch and (B) BLAST performance of BLOSUM62 and RBLOSUM64. Thick lines represent the original data, while thinner lines represent individual bootstrap replicates. The insets plot the difference in performance between BLOSUM62 and RBLOSUM64. Again, negative values indicate that BLOSUM62 performs better, error bars are 95% confidence intervals calculated using Bayesian bootstrap replicates, and error bars that do not cross the origin indicate statistically significant differences between BLOSUM62 and RBLOSUM64. Results from the best-performing gap penalties are displayed; in (A), for both matrices optimal (existence, extension) values were (9,1), while in (B) the values were (11,1) for BLOSUM62 and (10,1) for RBLOSUM64.

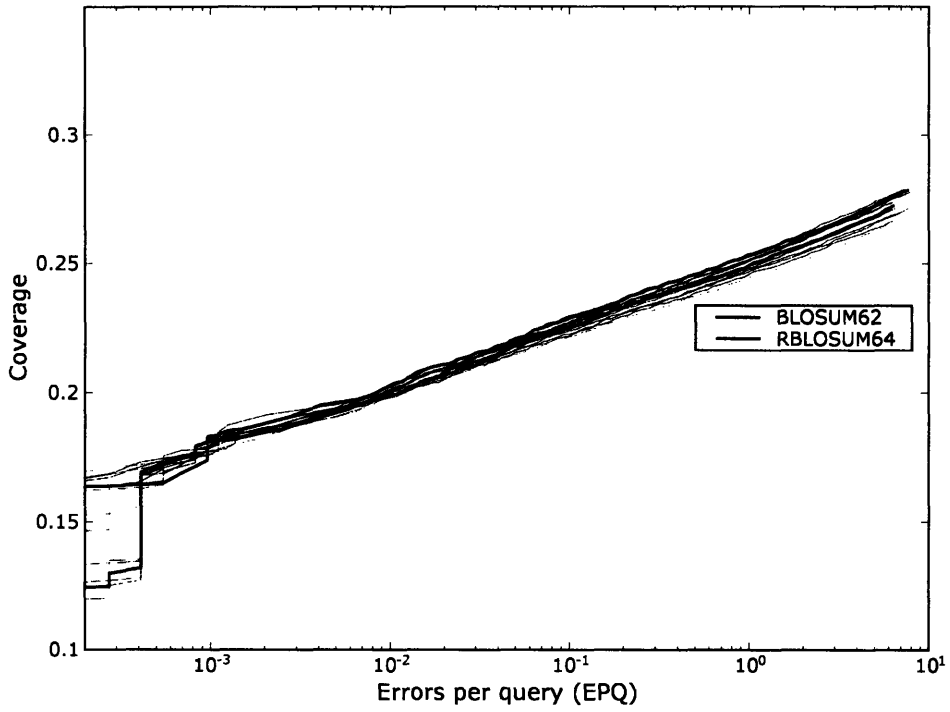


Figure 7-4: A CVE plot for BLOSUM62 and RBLOSUM64 performance distributions using gapped BLAST. Rather than Bayesian bootstraps, we display versions of the BLOSUM62 matrix derived from Blocks databases with shuffled blocks and versions of the RBLOSUM64 matrix derived from Blocks databases each with 25% of the blocks removed. These RBLOSUM64 matrices help to indicate the training set–dependent error that is inherent to a matrix rigorously derived from Blocks. We note that the variation of the RBLOSUM64 matrices is less than that of the BLOSUM62 matrices, and that the original BLOSUM62 matrix (the darker line) is not representative of the "mean" performance one could have expected. BLOSUM62 (and shuffled derivatives) gap penalties are as described in Figure 7-1, while all RBLOSUM64 reduced–training set (existence, extension) gap penalties were (10,1) except for one matrix, which used (11,1).

error and uncertainty inherent to RBLOSUM64, the matrix that "should" have been created.

In another set of experiments, we increased the scale of the matrices used in ssearch queries. BLOSUM62 values are scaled in half–bits, meaning that the base–2 log–likelihood raw scores computed from the training set substitution frequencies are

multiplied by 2 before being rounded off to integers for ease of use. However, this approximation discards some useful information contained in the raw matrix values. If, instead, we multiply those raw scores by 3 before rounding them (for a third–bit scale matrix), we would expect that the performance of the matrix would at least stay the same, if not improve from truncating less information content in the matrix values. While this was true for the RBLOSUM64 matrix, which had small (though not statistically significant) improvements, BLOSUM62 actually had statistically significant decreases in performance (using ssearch; data not shown). These two third–bit matrices performed very similarly, indicating that perhaps the intrinsic difference between them is not very great. In fact, the RMSD for the raw values in the matrices is only 0.104, supporting the idea that they are not too intrinsically different before scaling and rounding.

7.1.5 Discussion

We have shown that due to some anomalies in the program used to create the initial BLOSUM family of matrices, the widely–used BLOSUM62 matrix is, in fact, more effective than it "ought to" have been. In addition, we demonstrated that due to one of these anomalies, the original family of BLOSUM matrices was dependent upon the order of the sequences in the Blocks 5 database, meaning that the BLOSUM matrices currently used are not the unique versions of these matrices that could have initially been derived. While some of the issues in the original `blosum.c` code have since been identified and corrected, we find it interesting that fourteen years later, the community at large is not aware that the BLOSUM62 matrix used by so many people is not exactly what we thought it was.

As discussed above, the order of the sequences in the Blocks database had a profound effect on the values in the BLOSUM62 matrix and the effectiveness of that matrix. It seems that, fortuitously, one of the best possible arrangements was used. Interestingly enough, this arrangement is partly a simple matter of alphabetization: first all clusters with only one sequence are listed in alphabetical order, and then all clusters with multiple sequences are listed in an order such that the first sequences

across all of these clusters are in alphabetical order. With any different implementation, the effectiveness of BLOSUM62 may have been acceptable, but quite different.

We believe that the slight performance boost in BLOSUM62 relative to RBLOSUM64 is largely due to artifacts of truncation error. While BLOSUM62 has an abnormally high entropy for a re–clustering value of 62% with respect to other matrices that could have been formed with the original BLOSUM implementation, this higher entropy does not correlate with performance. It has been noted that the entropy of the BLOSUM62 matrix based on its scaled and rounded values deviates noticeably more from the raw matrix's entropy than one typically sees for other scaled and rounded matrices [8]. While using these scaled and rounded entropies to attempt "isentropic" comparisons partially resolves the ssearch performance difference between BLOSUM62 and RBLOSUM64, it only exaggerates the problem in BLAST searches (data not shown). While this entropy–related anomaly does not completely explain the performance difference of BLOSUM62, it does point out that there may be something different about the scaling and rounding of BLOSUM62 relative to other scoring matrices. Combining this observation with the fact that a larger scale for BLOSUM62 actually leads to a decrease in performance, we can then reasonably infer that the improved performance of BLOSUM62 is due to nothing more than a rare set of circumstances caused by miscalculated normalizations, low–resolution scaling, and some fortuitous rounding. This conclusion is actually quite fascinating, as it indicates that there is still more to learn about amino acid substitution frequencies: if essentially random processes (rounding and normalization errors) can cause a statistically significant increase in performance for our best rational estimates of these frequencies, then there is likely some fundamental aspect of modeling amino acid substitution that we have yet to grasp or capture.

The entire situation speaks to the circumstances of all scientific research: when we get results that are novel and reasonable, it is often difficult to thoroughly review the minutiae of one's own work and determine if there were any small anomalies along the way that may have affected the results. Perhaps most noteworthy is the fact that when some of these bugs were later fixed, their impacts were not updated: a

recomputed "correct" BLOSUM62 matrix was never adopted. Ultimately, this may have worked out for the best, as we have all been a little bit more productive in our BLAST searches and other work without even knowing it.

7.2 The evolution of the Blocks database and BLOSUM matrices

7.2.1 Overview

The fidelity of amino acid sequence alignment methods depends strongly on the target frequencies implied by the underlying substitution matrices. The BLOSUM series of matrices, constructed from the Blocks 5 database, is by far the most commonly used family of scoring matrices. Since the derivation of these matrices, there have been many advances in sequence alignment methods and significant growth in protein sequence databases. However, the BLOSUM matrices have never been recalculated to reflect these changes. Intuition suggests that if the Blocks database has changed — by the growth or addition of blocks — that matrices computed after these changes may be different than the original BLOSUM matrices.

Here we show that updated BLOSUM matrices computed from successive releases of the Blocks database deviate from the original BLOSUM matrices. At constant re–clustering percentage, later releases of the Blocks database give rise to matrices with decreasing relative entropy, or information content. We show that this decrease in entropy is due to the addition of large, diverse families to the Blocks database. Using two separate tests, we demonstrate that isentropic matrices derived from later Blocks releases are less effective for the detection of remote homologs, and that these differences are statistically significant. Finally, we show that by removing the top 1% large, diverse blocks, the performance of the matrices can largely be recovered.

7.2.2 Introduction

Many different scoring matrices have been proposed in the literature, but the BLOSUM series [66] and PAM series [39] of matrices are by far the most widely used. For reviews of the many different substitution matrices, the reader is referred to a variety of literature articles [69, 67, 174]. Despite the vast array of matrices available, a single matrix, BLOSUM62, has become a *de facto* standard — it is the default matrix for popular pairwise sequence alignment tools such as BLAST [10] and FastA [131] and multiple sequence alignment tools such as Clustal–W [163] and t–coffee [124].

The method used to create the Blocks database has its ancestor in the PAM matrices by Dayhoff [39]. The PAM matrices are made by first collecting a database of proteins known to be closely related. From these, a common ancestral protein sequence is inferred and the mutations required to produce the descendant sequences are tabulated. Because the set of descendant proteins is assumed to have diverged at the same time, the table of mutations can be interpreted as a rate of mutations. By extrapolating this mutation rate matrix (by multiplying each matrix by itself an arbitrary number of times), amino acid substitution matrices for increasingly divergent sets of proteins can be constructed, e.g. PAM1, PAM120, and PAM250.

The BLOSUM series of matrices was constructed in 1992 from Blocks 5 [66]: a database of protein blocks, or highly conserved protein regions, derived from families in the PROSITE database [74]. These blocks were used as a training set to derive a set of implied target frequencies that dictate the frequency with which an amino acid of one type should be aligned with an amino acid of another type. The various members of the BLOSUM matrix family — BLOSUM100, BLOSUM62, BLOSUM50, etc. — were made by clustering the sequences in each block at various thresholds, effectively down–weighting similar sequences to create matrices optimized for aligning more distant homologs.

The Blocks database is itself used for homology searching [68, 133] and other functions [144, 120]. As such, it is periodically updated, with ten major releases in the past ten years and some minor releases. Intuition suggests that these improvements

in the Blocks database may make it a better training set for creating scoring matrices. The goal of this manuscript is to show the effects of updates to Blocks on the matrices derived from the database.

When the BLOSUM matrices were initially created and published, it was hypothesized that the use of more protein groups (and thus more blocks) in the matrices' creation would have little effect on the matrix [66]. This was supported by the removal of specific blocks, or even half of the blocks, yielding approximately the same matrices. However, in retrospect it is obvious that the known protein motifs in 1992 are a small fraction of those cataloged in today's databases. Furthermore, it is plausible that motifs discovered "early" were inherently biased due to experimental methods and likely not representative of nature as a whole. It is unclear whether new, more recent blocks would yield identical, similar, or significantly different matrices.

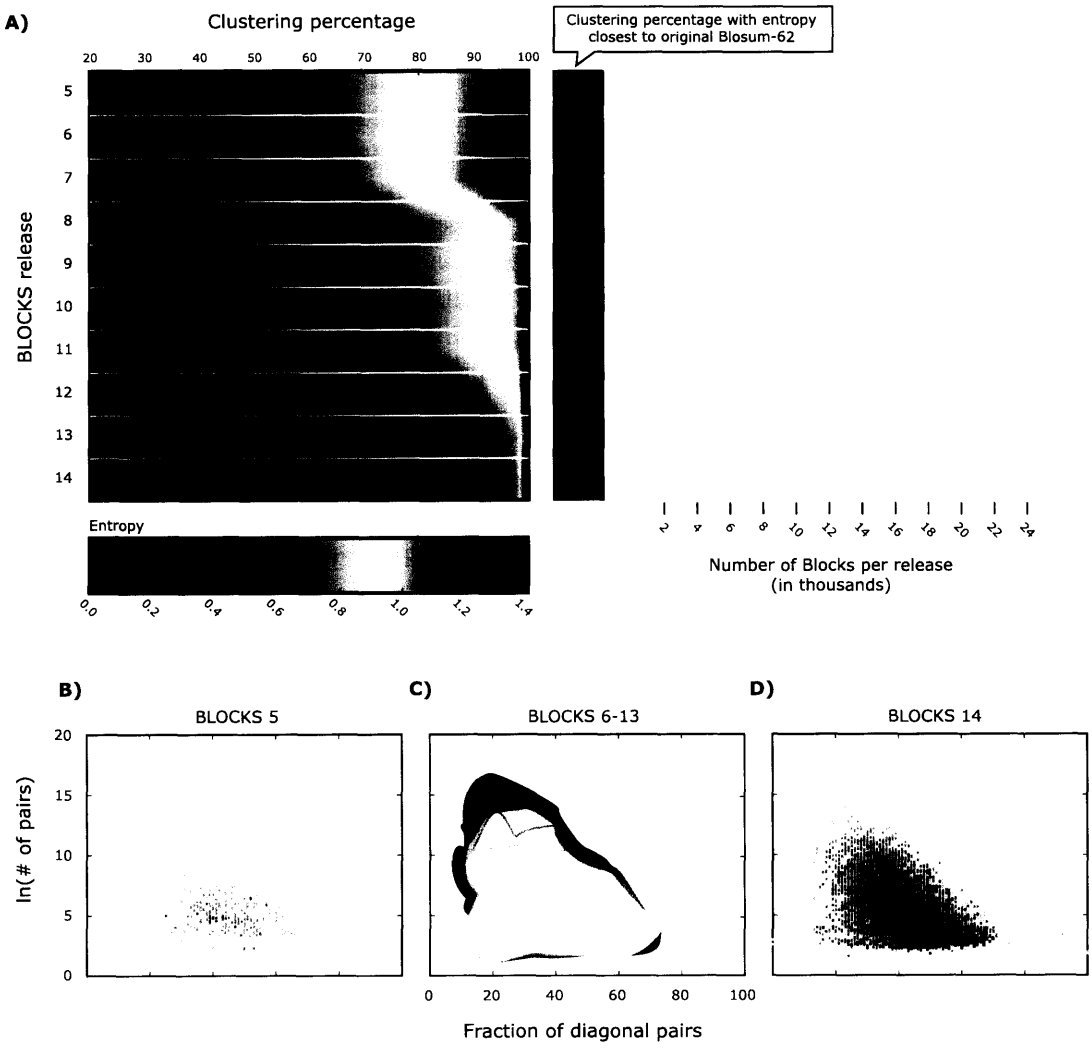
In the following sections, we detail the construction of updated BLOSUM scoring matrices from successive releases of the Blocks database and describe the results of two sequence alignment tests used to evaluate the performance of these matrices.

7.2.3 Methods

Matrix construction

All previous versions of Blocks databases were taken from the Blocks ftp server, `ftp://ftp.ncbi.nih.gov/repository/blocks/unix/`. BLOSUM matrices were constructed using a version of the BLOSUM source code (available from the above FTP server) originally used to prepare the BLOSUM family of matrices, but with some slight modifications and bugfixes discussed above. These changes included fixing integer overflows in multiple locations and fixing the weighting of substitutions between clusters of sequences. For each version of the Blocks database, a full scan of all integer–valued re–clustering percentages between 20 and 100 was performed (Figure 7-5). The matrix for each Blocks release with relative entropy closest to the originally reported BLOSUM62 matrix (0.6979) was selected as the representative matrix for that release.

Figure 7-5: Characteristics of the BLOSUM matrices calculated from successive releases of the Blocks database. Panel A) shows the entropy of the scoring matrices computed from various Blocks releases as a function of the clustering percentage used by the BLOSUM algorithm (see methods). Blue colors indicate low entropies and red colors indicate high entropies. A specific clustering percentage indicates that sequences within a block above that percentage are clustered together, such that their contributions to the counts of amino acid pairs are down–weighted. Intuitively, as the clustering percentage is increased, blocks that have greater degrees of similarity contribute more and more pairs, and the resulting matrices have a higher entropy, or equivalently, more information content. Oddly, at constant clustering percentage, matrix entropy decreases with successive Blocks releases (see part B below). The middle part of panel A) shows the clustering percentage which results in the matrix which has an entropy closest to the original BLOSUM62 matrix. The rightmost panel shows the number of blocks in each release of the Blocks database. As shown, the database grew tremendously in recent releases. Panel B) of the figure shows a scatter plot in which each block in the Blocks 5 database is represented as a dot. The location of the dot along the x-axis represents the percent of the amino acid pairs contributed by that block that lie along the matrix diagonal — i.e. identical pairs such as A–A, G–G, etc. The location of the dot along the y–axis indicates the total number of amino acid pairs contributed by that block. (Note that the y–axis is in log units and that the matrix was computed at 50% clustering.) In general, blocks located towards the upper right quadrant of the plot contribute to higher entropy matrices, whereas those towards the upper left quadrant contribute to lower entropy matrices. In the middle panel, the outlines of the point clouds for each successive release of the Blocks database are shown (light to dark colors). Finally, panel D) shows the scatter plot for Blocks 14. Notably, successive releases of the Blocks database incorporated many large blocks comprising distantly related sequences, as shown by the migration of the point clouds towards the upper left quadrant. These new blocks contribute many off–diagonal pairs and are the root of the trend shown in panel A): decreasing entropy at constant clustering percentage with successive releases of the database.


A)
Clustering percentage
Clustering percentage with entropy closest to original Blosum-62
BLOCKS release
Entropy
Number of Blocks per release
(in thousands)
B)
BLOCKS 5
C)
BLOCKS 6-13
D)
BLOCKS 14
ln(# of pairs)
Fraction of diagonal pairs

Sequence datasets

Two different database searches were used to judge the ability of each matrix to detect homologs: a search of SWISS–PROT 22 [21] using a set of queries previously determined to reflect "difficult" searches that are able to distinguish the abilities of different matrices [67], and a search of the ASTRAL database [30] using each member as a query. These two different validation strategies have different benefits: the former is historically relevant, as it was a method used to initially demonstrate the superiority of BLOSUM62 to other matrices [66, 67]. The latter is more time–consuming, but it reflects current knowledge of protein homology and allows for the determination of the statistical significance of differences between matrices.

The first method we used for testing matrices was designed to emulate previous work [67]. In that work, the 257 PROSITE 9.0 [18] families that were most challenging to detect were used as queries against SWISS–PROT 22 (numbering 25,044 sequences). For each family, the list of all members was used as true positives.

The second method we used for testing matrices was also designed to emulate previous work [136]. We used the ASTRAL database [30] as the basis for our more exhaustive experiments for detection of remote homologs. ASTRAL is created based on the SCOP database [117], which classifies proteins based on their function, structure, and sequence into a hierarchical structure of classes, folds, superfamilies, and families. Sequences in the same superfamily can have low sequence similarity, but are likely to have a common evolutionary origin based on their structural and functional features. Because these classifications are made by human inspection, not via automated sequence alignment procedures, it makes a perfect "gold standard" for remote homolog detection tests.

From the full set of ASTRAL genetic domain sequences, we chose the sequence set from which 40% identical sequences had been eliminated. By using this subset, our search focuses on the detection of remote homologs that are more challenging for substitution matrices to discover and thus will differentiate the abilities of the respective matrices to find distant relatives. The sequences were further filtered by

pseg [181] for the removal of low–complexity regions. The unfiltered sequence set is available on–line from the ASTRAL database at `http://astral.berkeley.edu/scopseq-1.69/astral-scopdom-seqres-gd-sel-gs-bib-40-1.69.fa`. This non–redundant set numbers 7,290 sequences. Each sequence was extracted from the database one at a time and used as a query for the entire database. Search results in the same superfamily as the query were considered to be true positives.

7.2.4 Search methods

We used both the Smith–Waterman [152] local alignment algorithm and BLAST [10] for searches against the databases. In particular, we used the ssearch implementation of the Smith–Waterman algorithm [129, 130] and the NCBI version of BLAST in the C Toolbox, obtained from `ftp://ftp.ncbi.nlm.nih.gov/toolbox/ncbi_tools/`, which gives the same results as the commonly–used web interface at NCBI. While Smith–Waterman is an exhaustive, sensitive search that may give a more accurate assessment of a given matrix's performance capability, BLAST is a faster search that is almost an "industry standard" and so better represents the impact of different matrices on the "average user". In addition, BLAST can be run in an ungapped fashion to eliminate the effects of gap penalty parameters.

For our Smith–Waterman searches, we used the ssearch default parameters for unknown matrices, which are a -10 penalty for gap initiation and a -2 penalty for gap extension. We believe that these parameters are reasonable settings; they represent an intermediate ground between the values used in the initial BLOSUM paper (-8/-4) and current commonly–used settings (for instance, the defaults for BLOSUM62 in ssearch are -7/-1, while in BLAST they are -11/-1). Moreover, previous work [60] has shown that while slight performance boosts can be found by optimization of gap penalties, there is frequently a broad maximum of penalty values with approximately equal efficacy. In addition, a sampling of the Kolmogorov–Smirnov statistic values returned by ssearch for searches using our penalty values were well within the acceptable range. This indicates that the distribution of alignment scores is the expected extreme–value distribution and that a significant alteration of the gap penalties is most likely

unnecessary. That is, our penalties are neither too forgiving nor too permissive.

Since the goal of this work is to analyze the BLOSUM matrices as affected by the changing entries in the Blocks database, the consistent use of some average, acceptable parameter values for all matrices provides a level, controlled environment for determining the relative raw ability of each matrix to detect remote homologs without requiring that we find the optimal parameter values for each matrix for our Smith–Waterman searches.

For our BLAST searches, we took advantage of the speed of the algorithm to scan the likely region of local optima of gap parameters. After varying the gap initiation (existence) penalty from -6 to -14 and the gap extension penalty from -1 to -2 for a test matrix, we determined that merely scanning from 9 to 12 in the existence penalty (with an extension penalty of 1) is likely to find the global optimum of gap penalty parameters. Under these conditions, we saw local optima of performance within those parameter ranges for most matrices.

Other than gap penalties, all other default parameters for the toolbox version of BLAST version 2.2.13 (12/6/2005) "blastall" were used. Notably, this does not include composition–based adjustments of E–values, which are now used by default on the NCBI web interface of BLAST. Previously [146], these adjustments have been shown to actually decrease performance in individual protein database BLAST searches.

7.2.5 Evaluation of results

For both sets of database searches, we used the same respective methods for evaluating search results as in previous literature. In the PROSITE–based testing, we used head–to–head comparison of effectiveness in finding family members. For all PROSITE families that were queried, the matrix that found the most true positives was noted. The relative effectiveness of any two matrices was then found by subtracting the number of times that one matrix was more effective from the number of times that the other was more effective. True positives were defined as described previously. The search criterion used was the same as for the previous work [67],

as previously described [130]: if a true positive appeared before 99.5% of the true negative sequences, it was considered "found".

For ASTRAL–based testing, we used the Bayesian bootstrap method to evaluate the statistical significance of the mean difference in coverage between any two substitution matrices [136]. This method uses coverage vs. errors per query as a means to evaluate the effectiveness of different substitution matrices. Coverage is defined simply as the fraction of true positives found at a given errors per query threshold. True positives were identified as described above.

7.2.6 Results

We began by first assembling the matrices that we would be using in our experiments. As stated in the Methods section, we used a modified version of the original BLOSUM program that incorporated multiple bugfixes. We created a matrix for each integer clustering value between 20 and 100; the results can be seen in panel A of Figure 7-5.

The center of panel A lists the re–clustering percentage needed for each Blocks release to produce a matrix with entropy closest to that of the original BLOSUM62 matrix. We used this set of isentropic matrices for our sequence alignment tests. A given matrix's relative entropy reflects the required minimum length of homology in order for it to be distinguished from noise [6]. Merely maintaining (in this case) a re–clustering percentage for a time–dependent family of matrices would have little meaning, as changes in entropy could occur that would obscure the effectiveness of the information encoded in the matrix. In this sense, it is only "fair" to compare matrices of the same entropy. Thus, we used matrices with the same relative entropy of BLOSUM62, 0.6979, which is approximately the value previously shown to be most effective for database searches [66]. (Note that, due to the bugfixes mentioned earlier, the BLOSUM matrix computed from Blocks 5 had its entropy analog at a re–clustering percentage of 64 rather than 62.) We refer to matrices computed from the "revised" BLOSUM code as RBLOSUM, making the baseline matrix for that family RBLOSUM64.

The right-hand side of panel A in Figure 7-5 shows that the number of blocks in

each release increases in an almost monotonic fashion, with the exception of release 9. The general trend is expected, as the PROSITE database that is used to create the blocks would likely have more families of known homology added in later releases. The decrease in blocks in release 9 remains an anomaly; we speculate that it may have been due to a one–time change in parameters in the creation of the blocks, though we have no way to verify this theory.

Inspecting the heat map in panel A of Figure 7-5 reveals that, as expected, relative entropy increases with increasing re–clustering percentage in any given Blocks release. However, at constant clustering percentage, matrices computed from successive releases of the Blocks database show markedly decreased relative entropy. We hypothesized that this trend was due to changes in the character of blocks in the database. Indeed, panels B–D of Figure 7-5 suggest that the presence of extremely large, diverse blocks may have been the cause of this phenomenon. The scatter plots in panels B–D show point clouds representing all the blocks in a given Blocks release (panel C shows the outlines of these clouds). Each block is represented as a single point at a location that indicates the degree to which the block contributes identical amino acid pairs (x–axis) and the total number of amino acid pairs contributed by the block. The three panels show a trend towards the incorporation of blocks that have many sequences that are only remotely homologous. This trend is manifested in the migration of the point clouds towards the upper left quadrant of each of the three scatter plots.

These panels explain why the re–clustering percentage needed to be increased so much in order to create isentropic matrices. As large blocks with more diverse sequences are added to the database, something must be done to offset that diversity in order to obtain an isentropic matrix. Since the highly diverse members of a family (block) will not cluster together, they will have a significant impact on the substitution counts that are used to derive the matrices. In order to offset this impact and steer the entropy of the matrix away from that of the background, it is necessary to increase the re–clustering percentage used to compute the matrices. In this way, blocks containing highly homologous sequences will have greater influence on the substitution counts

and steer the matrix closer to the desired counts and information content.

Having assembled a set of isentropic matrices, we then used our two tests — the historical, PROSITE–based test and the statistically rigorous, ASTRAL–based test — to evaluate the effectiveness of updated BLOSUM matrices. By using both of these tests rather than just one, the comparison of updated substitution matrices is grounded in the same metrics as would have been used when the matrices were first published, while providing quantitative statistical results.

We found that, with time, the character and quality of the entries in the Blocks database has changed significantly. Figure 7-6 shows a slightly complex trend that warrants some analysis. The figure shows boxes whose vertical position indicates their relative performance; the further a box is vertically from the Blocks 5 box, the greater the difference in performance between the isentropic matrices derived from those releases (see caption). In early updates of Blocks, the resulting RBLOSUM matrices tended to hover around a certain performance. This is consistent with previous hypotheses [66] that the BLOSUM matrix would not be altered by adding to or subtracting from the Blocks database. The variation could be explained in part by integer rounding; since the desired scores are rounded to the nearest whole number, it is possible that the intended scores for a given matrix are not completely accurately represented by a given BLOSUM matrix. Another possibility is that changing block quality causes these fluctuations; this possibility is further analyzed below. However, the particularly poor performance of Blocks releases from 12 on, and that of release 9, is inconsistent with the initial hypothesis that matrix performance would remain approximately constant.

These results are largely consistent with our results from the ASTRAL–based tests. Figure 7-7 is a representative result for a set of Bayesian bootstrapping runs for the ASTRAL–based test (in this case, for releases 5 and 14 of the Blocks database). The lighter, thinner lines track coverage as a function of the allowed errors per query (EPQ) for individual bootstrap runs, while the two thick lines represent the full–database result. Clearly, there is some overlap between the two distributions, but a pairwise comparison of runs (as demonstrated by the inset evaluated at 0.01 EPQ)

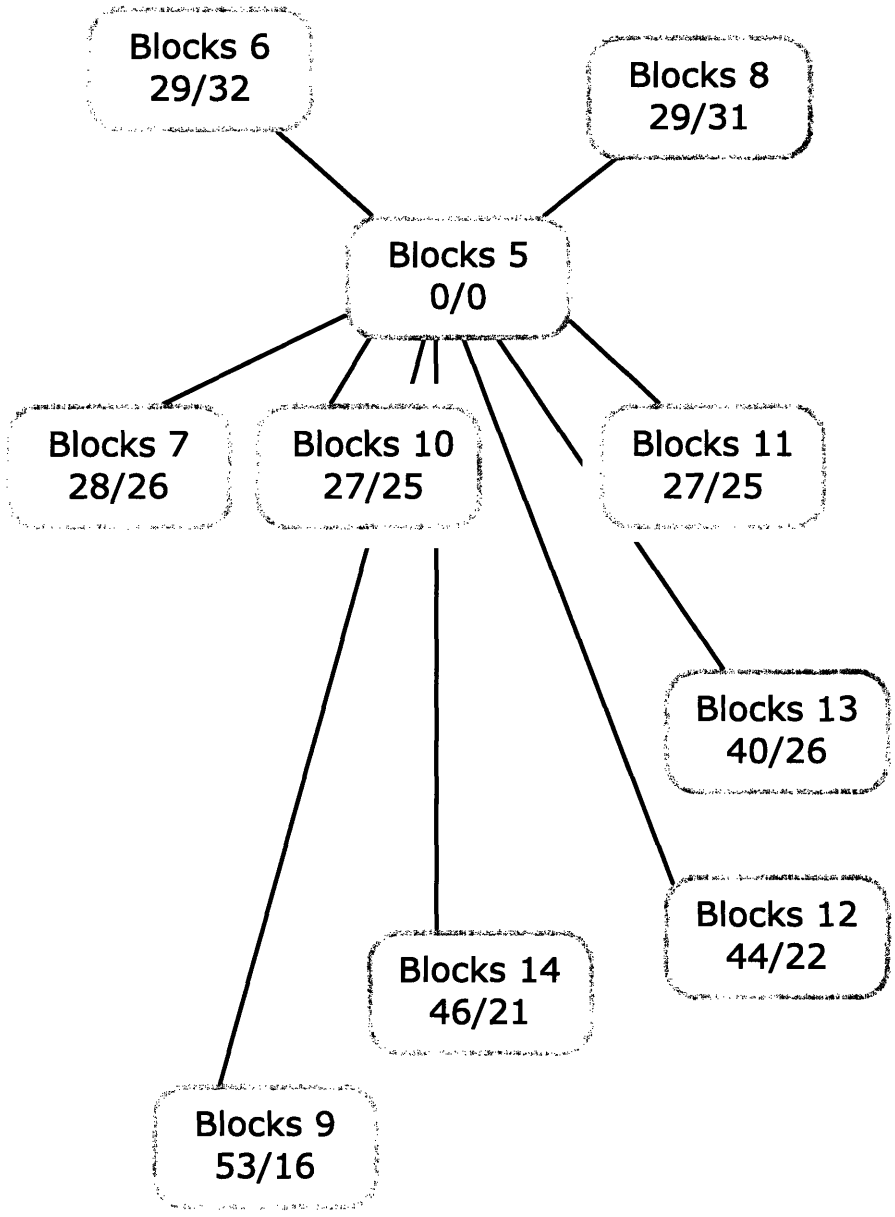


Figure 7-6: The relative performance of updated BLOSUM matrices. This figure is designed to emulate Figure 4 from the initial BLOSUM manuscript [66]. All matrix performances are compared to the revised BLOSUM62 isentropic analogue derived from Blocks 5, RBLOSUM64. Vertical distance from Blocks 5 indicates relative performance, with matrices above Blocks 5 performing better and those below it performing worse. Comparisons were based on the 257 "difficult" queries [67], derived from PROSITE 9.0 keyed to SWISS–PROT 22. Numbers in each box indicate the number of groups for which RBLOSUM64 from Blocks 5 performed better than and worse than isentropic matrices from other releases. Releases immediately following Blocks 5 seem to cluster around the same level of performance, while later releases (and release 9) have unusually bad performance.

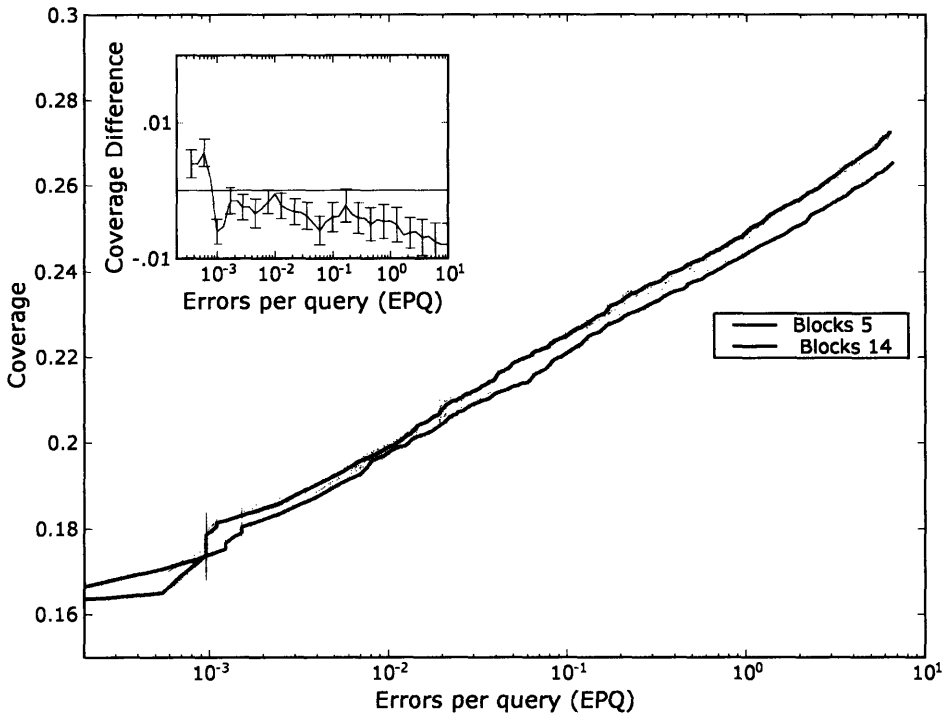


Figure 7-7: A complete set of Bayesian bootstrap replicates, with inset plot of performance difference statistics. These data were created using the PSCE software [136]. (See that manuscript [136] for a thorough explanation of Bayesian bootstrapping). Each thin, faintly colored line represents one Bayesian bootstrap run. The thick lines represent the total dataset results. In this case, the two distributions overlap somewhat, but statistical analysis of the data reveals that the difference in coverage is statistically significant across a wide range of EPQ values.

shows a distinctly non-zero difference between the two distributions. The difference in coverage at a variety of EPQ values can be used as a metric to judge how consistently different the performances of any two matrices are.

This metric is used in Figure 7-8 to show the performance of all updated matrices relative to the baseline RBLOSUM64 matrix computed from Blocks 5. These results correspond quite well to the results in Figure 7-7. That is, releases 7, 8, 10, and 11 perform comparably to 5 and release 6 is slightly better, while releases 9, 12, 13, and 14 perform substantively worse than release 5. These latter releases have statistically significant differences. This agreement suggests that the original test employed in

Figure 7-6 [66, 67] was rather effective and efficient in that the results of the test would not have changed much with access to today's larger databases.

7.2.7 Discussion

The reason for the poor performance of RBLOSUM matrices derived from later releases of Blocks remains to be explained. Figure 7-5 suggests that the number of blocks and shifting isentropic clustering percentage are not reasonable explanations. If these were so, one would expect to see either gradually degrading performance (for database size) or significant step changes in performance at releases 8, 12, and 13 (for isentropic clustering percentage). However, there is certainly not a gradual degradation in performance, and there is no significant change in performance at release 8. In addition, any decrease in performance at release 9 disappears for the next two releases.

We hypothesized that two phenomena — the decreased entropy at constant clustering in successive Blocks releases, and the poor performances of these releases — were both caused by the changing character of blocks added in later releases. Specifically, we thought that the trends shown in panels B through D in Figure 7-5 might be responsible for these phenomena.

To test this hypothesis, we sorted the blocks in the Blocks 14 database by the number of off–diagonal (i.e., non–identity) amino acid pairs contributed to the RBLOSUM matrix by each block. We then removed the blocks that were the top 1% of contributors to off–diagonal pairs (243 blocks) and created an isentropic RBLOSUM matrix from this "cleaned" database. Notably, the re–clustering percentage required to create an isentropic matrix decreased from 94 to 84 for the cleaned database. The performance of this matrix relative to RBLOSUM64 from Blocks 5 is shown in Figure 7-9. The cleaned version of the Blocks 14 database largely restores its utility to the level of Blocks 5, and provides a statistically significant improvement in performance over the original Blocks 14 (see Figure 7-10).

The performance of the RBLOSUM matrix created from the "cleaned" Blocks 14 database supports our hypothesis that the addition of large, diverse blocks has had

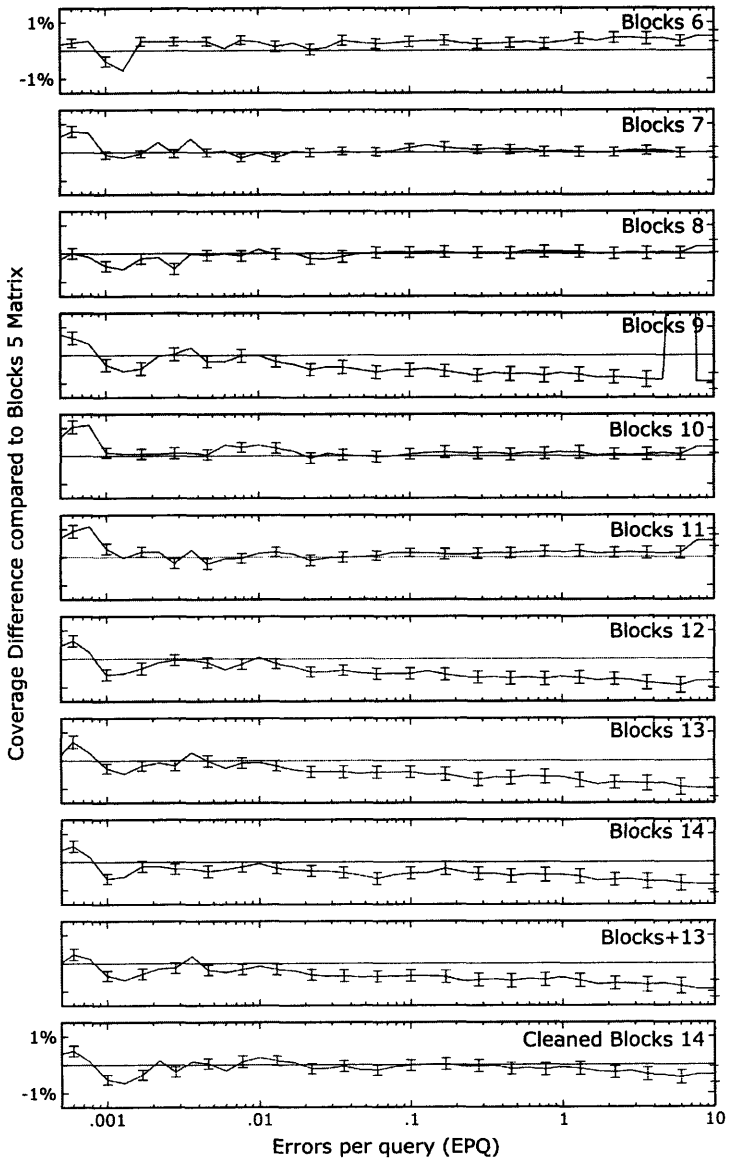


Figure 7-8: Plots of the differences in performance of updated RBLOSUM matrices. Each matrix is compared to the RBLOSUM64 matrix (derived from Blocks 5) in 200 Bayesian bootstrap replicates to find the mean difference in coverage, and the confidence interval for that coverage, at a specific EPQ rate. These differences are plotted as a function of EPQ rate, with positive values meaning that a given matrix performs better than RBLOSUM64 on the dataset. Error bars represent 95% confidence intervals. At data points where the error bars do not intersect with the origin, the performance difference between the matrices is statistically significant. These results correlate well with, and provide statistical analysis of, the results in Figure 7-6.

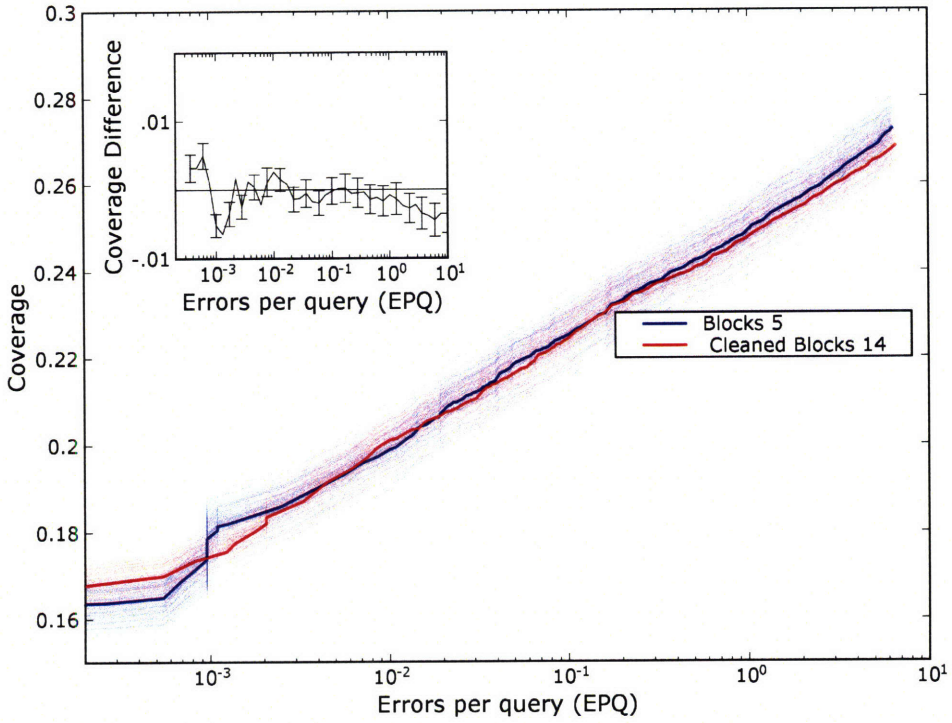


Figure 7-9: Coverage of a cleaned RBLOSUM matrix compared to the original RBLOSUM64 matrix. Again, thin, faint lines represent individual bootstrap runs, while the dark line represents the parent dataset. These two distributions are quite similar, with the cleaned RBLOSUM matrix being about as effective as the RBLOSUM64 matrix. The inset shows the coverage difference between the two matrices' coverage as a function of errors per query. Error bars represent 95% confidence intervals. Note that most error bars cross the origin, indicating statistically indistinguishable performance between the two matrices.

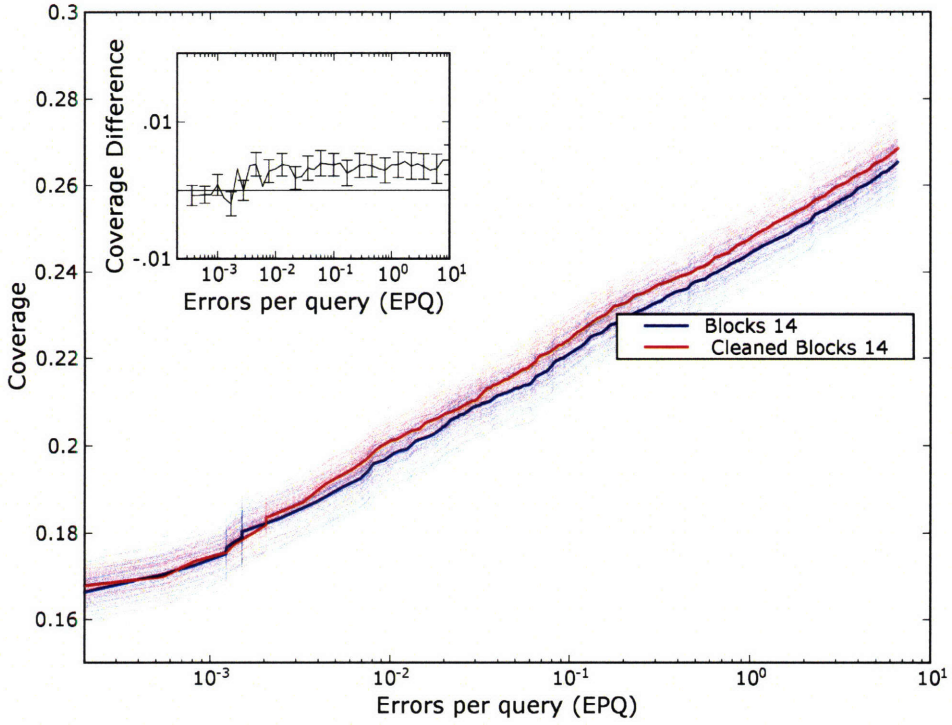


Figure 7-10: Coverage of a cleaned RBLOSUM matrix compared to the RBLOSUM matrix derived from Blocks 14. Again, thin, faint lines represent individual bootstrap runs, while the dark line represents the parent dataset. These two distributions are quite distinct, with the cleaned RBLOSUM matrix being significantly more effective than the RBLOSUM matrix derived from Blocks 14. The inset shows the coverage difference between the two matrices' coverage as a function of errors per query. Error bars represent 95% confidence intervals. Note that at most EPQ values the error bar does not cross the origin, indicating a statistically significant difference in performance.

an adverse effect on the performance of updated RBLOSUM matrices. We believe that the decrease in performance may be due to a change in the database that is used to create the Blocks database [65]. Initially, Blocks was based on the PROSITE database. As of release 12 of Blocks, blocks were formed from InterPro groups rather than PROSITE groups. In release 12, only InterPro groups with cross–references to PROSITE groups were used to create blocks. In release 13, this restriction was lifted, and it has remained lifted to the current release of Blocks. We believe that this explains almost all of the trends that we observe in the data. When the Blocks database partially shifted to being based on InterPro, performance first decreased slightly with the addition of sequences that had not previously been included. When the shift was completed, performance degraded significantly. The only unexplainable anomaly is the unusually poor performance of release 9 of Blocks; we believe that can be attributed to the unusually small number of blocks in that release. Again, we speculate this may have been due to some one–time change in parameters, but we have no way to prove or disprove such a speculation.

In conclusion, we see that in some sense, the hypothesis initially proposed [66] was true: for releases of the Blocks database based on PROSITE, despite some slight variation, the performance of isentropic RBLOSUM matrices is relatively constant over successive releases. However, since the quality of the blocks added in recent releases has decreased, such is not the case for the matrices derived from the current Blocks database. This suggests that, to the extent that there are "bad" blocks, there may also be "good" blocks, and sensible, judicious selection of these blocks may be a reasonable approach for the creation of amino acid substitution matrices.

Chapter 8

Conclusion

8.1 Summary of results

In Chapter 3, we developed a generic motif discovery algorithm (Gemoda) capable of handling diverse types of sequential data. Our approach decoupled what we identified as three key steps in the motif discovery process: comparison, clustering, and convolution. Each of these steps is considered completely independently of the others, allowing for a modular setup where any comparison metric can be used with any clustering routine and any convolution method. As a proof of concept, we applied this approach to the discovery of binding sites in the upstream region of a number of *E. coli* regulons, the discovery of protein sequence motifs in a well–known class of enzymes, and the discovery of secondary structure motifs in proteins known to be in the same family but with significant primary sequence diversity. Our approach proved robust with respect to noise in quite a few cases and showed the versatility of a truly generic approach to motif discovery.

In Chapter 4, we applied our approach to the (l,d)–motif binding site discovery problem. This problem is an abstraction and simplification of the general transcription factor binding site discovery problem. We were able to solve this problem in a computationally reasonable time and in a provably exhaustive fashion. Most previous approaches were unable to accurately identify the embedded motifs in such a problem. Our philosophy also lent itself well to the expansion of this abstract problem to a less

restricted, more realistic problem where motif lengths can vary and the presence of motifs in a given dataset is not as certain. The other existing approach [52] that is capable of completely solving the (l,d)–motif problem would find significant difficulty in solving this extended problem. Thus, Gemoda has a unique contribution in being able to handle more biologically realistic datasets and still find motifs contained therein in a provably exhaustive fashion.

In Chapter 5, we then applied our approach to real–valued GC–MS data. We created a program called SpectConnect which essentially calls Gemoda in an iterative fashion and appropriately processes the results of each Gemoda run. Previous methods for analyzing GC–MS data largely depended upon the use of libraries of mass spectrum reference standards to track the metabolites in a sample. SpectConnect avoids the use of these reference libraries (except for retrospective analysis of its results) in favor of an approach that utilizes an increase in the signal–to–noise ratio that is created by running multiple replicates on the GC–MS instead of just one sample. While this causes a linear increase in experimental time due to the long runs typically associated with GC–MS analysis, it greatly increases the capabilities of downstream metabolomic analysis because one is no longer restricted to tracking only the metabolites with know reference standard mass spectra. We showed that our approach works both for simple chemical mixtures as well as for complex fermentation mixtures.

In Chapter 6, we looked to apply SpectConnect to the characterization and analysis of the accessible metabolome of a species. Given our assay of choice (GC–MS) and our derivatization chemistry of choice (TMS), we sought to track and analyze all of the metabolites that could possibly be detected. This can be done by performing a sufficient number of environmental and genetic perturbations that will cause intracellular fluxes and concentrations to shift, leading to each metabolite hopefully being at a detectable concentration in at least one experiment. Despite initial experimental difficulties, we performed a number of perturbation conditions and began preliminary analysis for the feasibility and efficacy of this study. Once some experimental techniques and protocols are refined, it seems likely this project will have a lot of promise

for a thorough analysis of the metabolome of a single species under many different conditions.

In Chapter 7, we explored a few side projects that were completed during the course of this thesis work. One analyzes some errors in the creation of one of the most widely–used amino acid substitution matrices and the impact that these errors have had on subsequent sequence comparisons and database searches. The second project investigates the imagined "evolution" of these matrices if they had been updated according to the amount of training data available for their creation. While both of these projects are a bit further flung from the motif discovery problems addressed in earlier chapters, they represent a broader approach to motif discovery and bioinformatics; they also entail the use of a wide variety of relevant tools and algorithms in bioinformatics, which is an important aspect of any course of learning involving computational biology.

Altogether, then, I have shown the creation and evolution of a novel, generic approach to motif discovery in sequential biological data that has quite a few interesting applications. A number of interesting possibilities for future work have arisen from this thesis; these will be discussed in greater detail below.

8.2 Objectives achieved

The following objectives of this thesis have been achieved:

1. To develop a motif discovery algorithm that is both as exhaustive and generic (or "data–agnostic") as possible. Gemoda is a completely generic approach to motif discovery. By breaking its calculations into three distinct and independent phases, a completely modular approach was created that allows any type of sequential data to be analyzed with the same algorithm, requiring only an appropriate comparison function for the specific data type. Subsequent choices of clustering and convolution functions help to further increase the capabilities of this motif discovery approach while still providing reasonable and useful default functions that will be useful for most problems.

2. To apply this approach to existing problems in bioinformatics. The (l,d)–motif problem was an outstanding problem at the time we were able to solve it in a provably exhaustive manner. The analysis of the upstream regions of regulons in *E. coli* for the discovery of transcription factor binding sites is a relevant biological problem. The discovery of local secondary structure motifs in protein structures is an interesting existing problem in bioinformatics. All of these problems have been addressed by using Gemoda to discover interesting or novel motifs in the respective datasets. In fact, the (l,d)–motif problem was extended and made more biologically relevant by extending the principles of Gemoda to that existing abstraction of the transcription factor binding site discovery problem.

3. To apply this approach to novel problems in bioinformatics. The development of SpectConnect is a perfect example of applying our generic approach to novel problems in bioinformatics. In some senses, the problem of identifying conserved spectral motifs in GC–MS data had not really been identified because existing techniques were so entrenched in the library–based comparison approach. By using our clustering and data–handling capabilities, we were able to define and solve this problem, providing a useful tool for metabolomic data analysis.

8.3 Future directions

Looking at the work in this thesis, there are some specific areas that could benefit from additional investigation. There are also some significant promising research directions that stem from the results presented in this thesis. These future directions, which include convolution, deconvolution, disease studies, and tracer analysis, will be discussed below.

8.3.1 Convolution

Convolution is the most important future direction for developing Gemoda. As noted in Chapter 3, Gemoda currently uses an exhaustive convolution method that depends on completely overlapping cliques. While this is useful for finding dense and continuous motifs, there are numerous examples in biology of motifs that contain gaps. These gaps are regions that have little to no sequence conservation and may be of variable width even for instances of the same motif. Discovery of motifs with gaps is certainly one of the hardest outstanding problems in bioinformatics, primarily due to the combinatorial increase of potential motifs that is associated with variable–width gaps.

The convolution step in Gemoda can likely be extended to begin to address the problem of finding gapped motifs. This step is just as modular as the comparison and clustering steps in Gemoda, so there is no reason that a different convolution method could not be implemented. That method need not be exhaustive like the current method; it could easily be heuristic. Either way, it is conceivable that a convolution method could be designed that would allow for some finite number of gaps in motifs such that more complex motifs can be found. In fact, this approach may even allow for the detection of multiple otherwise low–signal short motifs whose co–occurrence may indicate a strong motif; examples of this have been seen in programs such as MITRA [52]. Such an approach may also require a slightly different comparison function, though.

Even if a simple convolution step cannot be created to find gaps, it is likely that a "meta–convolution" approach could perform similar functions. In this case, we would look at the output created by Gemoda and essentially perform motif discovery on the motifs, identifying which motifs occur within some constant or variable distance of each other. While this approach is perhaps less elegant, it would fulfill the goals of gapped motif discovery and may provide additional insight. The obvious detriment here is that we would still be limited to motifs that are easily found on their own, thus limiting how close to the threshold of statistically insignificant noise each motif

component could be.

8.3.2 Mass spectral deconvolution

Mass spectral deconvolution is a key step in the SpectConnect work flow. In order to reliably compare the mass spectra that are found by the GC–MS, we must first tease apart any coeluting spectra such that we know what the pure spectra are for each metabolite that may elute in a peak. This step also usually helps to eliminate some noise in the mass spectrum as well. This problem has actually been studied in a quite a few other forms, most notably in the signal processing field; numerous approaches to deconvolution exist, and we need only adopt one that is appropriate to the problem at hand.

AMDIS, the tool that we currently use for mass spectral deconvolution, is capable of separating coeluting peak spectra but suffers from some key flaws. The most useful aspect of AMDIS is that it can take as input the raw data from any of a variety of different GC–MS manufacturers' equipment. This key feature means that by using AMDIS as an upstream processing step, SpectConnect can be used with essentially any GC–MS data, regardless of the manufacturer of the instrument used to capture the data. This cannot be said for many of the existing GC–MS data analysis approaches, many of which are restricted to the analysis of a certain manufacturer's raw data. In addition, AMDIS is freely available (even though it is not open–source), meaning that there is little economic hindrance in people using an AMDIS/SpectConnect workflow for analyzing their GC–MS data. The key flaw in AMDIS, though, is its oversensitivity. As noted in Chapter 5, AMDIS was designed for specific governmental purposes. Due to the requirements of its initial goals, AMDIS necessarily is extremely aggressive in identifying potential peaks. A cursory visual analysis of a chromatogram analyzed by AMDIS may reveal that one third, or even more, of the peaks that SpectConnect identifies are likely not metabolite peaks. Not only is this true for peaks very close to the baseline, but it is even true for very large peaks. One will often find multiple peaks enumerated that have almost identical mass spectra, indicating that AMDIS is being too sensitive in its definition

of what constitutes a true peak.

The development of a better deconvolution tool is thus a great opportunity for future work. While this clearly falls outside the purveyance of SpectConnect analysis, one can easily imagine using some combination of Gemoda's modular functions to effect a deconvolution algorithm. The premise here would be to identify individual ion trace chromatograms (as opposed to the Total Ion Current, or TIC, chromatogram) that covary. When quite a few individual ion chromatograms covary, it is likely that they constitute the core ions of the spectrum of a specific metabolite. One would then expand from those ion chromatograms to identify the portions of other ion chromatograms that may be a part of that peak's spectrum. This approach is somewhat similar to the approach that is implemented by AMDIS; in this case, though, we would have the advantage of open–source code for the appropriate adjustments and optimizations of the deconvolution approach as well as the advantage of controlling exactly how sensitive the method is. Obvious difficulties to overcome would include handling noise, changing baseline, peak tailing, and the identification of the best model of a peak.

The last difficulty mentioned above, that of identifying the best peak model, is particularly noteworthy. For an average quadrupole mass spectrometer, one frequently finds that the time per scan (as defined by the scan rate) for all of the ions being analyzed is of the same order of magnitude as the expected peak elution time. In this case, one sees a confounding effect where the mass spectra seem to "tilt" as the peak comes off the column. If we assume that the mass scanner goes from least to greatest m/z value, we will see that scans at the beginning of a peak's elution will be weighted towards the high m/z values, while scans at the end of a peak's elution will be weighted towards the low m/z values. This is because when the scanner starts at the low m/z value, there is very little of the compound eluting off of the column and so not much ion intensity is found. Around half a second later (for example) when the high m/z values are being scanned by the mass spectrometer, there is quite a bit more compound eluting off of the column and thus there is more overall ion intensity. This, even if the low and high m/z values are present at the same relative

intensity in the true mass spectrum, it would appear that, at the beginning of the peak's elution, the high m/z value would have a greater intensity that the low m/z value. This means that the mass spectrum changes as the peak comes off the column, thus further complicating the task of identifying a true mass spectrum and of finding individual ion traces that strongly covary.

8.3.3 Diseases and human metabolomics

The study of metabolomic effects of diseases is another promising future application of the work presented in this thesis. This work has already been started in one form by other members of our laboratory. It entails applying SpectConnect to the analysis of clinical samples to discover correlations between metabolite profiles and disease states.

The premise of the approach is simple and is illustrated in Figure 8-1. Whereas simple physiological readouts like blood pressure and pulse can give some idea of a person's condition, it is becoming increasingly important to use biofluid samples (i.e., blood and urine) to analyze more specific physiological readouts. One such well–known readout is the set of markers used for confirming the occurrence of myocardial infarction (heart attack). These, and other blood–borne markers, give insight into a person's (past or present) state with respect to a specific condition. Biofluids are likely to contain significant indicators that pertain to the organs and systems that directly interface with them; endocrine function and metabolic activity can likely be characterized, as can the function of the liver, pancreas, or kidneys. By taking these biofluids, extracting the metabolites, assaying them with GC–MS, and finding trends or patterns in the data, we can identify diagnostic or predictive markers (like myocardial infarction markers) that may increase the range of treatment options or improve prognosis based on earlier potential treatment. Small–molecule markers can already be seen in clinical use; for example, creatinine and urea in blood plasma are used to assay for loss of renal function. The discovery of similar markers for specific diseases or physiological conditions has great potential.

SpectConnect is uniquely equipped to perform such analyses thanks to its untar-

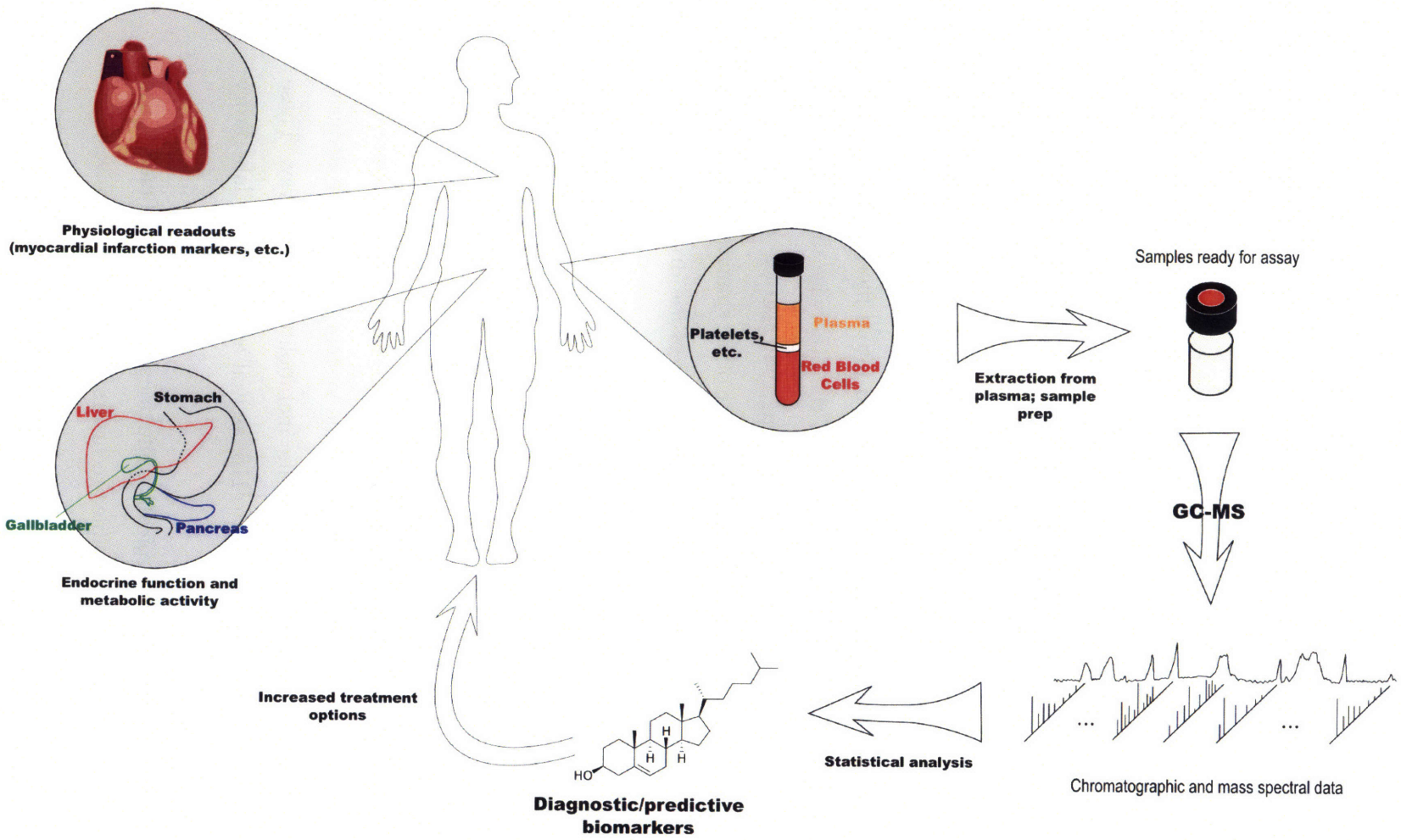


Figure 8-1: Workflow for metabolomic analysis of human disease. In the same vein as known physiological readouts like myocardial infarction markers, we can look in the accessible biofluids (like blood plasma, as in the right callout) to identify circulating extracellular metabolites that may give insight into the state of a variety of organs and systems (lower left callout). Metabolites can be extracted from these samples, appropriately prepared, and analyzed via GC–MS to give a metabolomic profile for blood plasma. Correlating these profiles with known diseased and control states will allow the identification of diagnostic and predictive biomarkers which will then in turn hopefully increase the range of treatment options and the efficacy of those options for affected patients.

geted approach to analyzing metabolomic data. Despite the progress of the Human Metabolome Project mentioned earlier, appropriate and reliable reference mass spectra are not prevalent enough to allow a library–based metabolomic analysis to be successful. By tracking all metabolite peaks, we can identify the peaks that are most strongly correlated with or most uniquely identify a specific state, whether or not the metabolite's chemical identity is known. The identification of these chemical structures is the logical next step in pursuing novel biomarkers for human diseases.

A similar approach can also be taken in the analysis of pharmacokinetic and metabolic effects of drugs. When a new drug is designed and tested, its degradation products must be identified. Just as SpectConnect can be used to analyze the changes in the metabolomic profile of biofluids in diseased states, it can also be used to analyze the changes of biofluids when a drug is injected into a model system. This approach will allow for the identification not only of the otherwise unknown and *a priori* unpredictable degradation products of the drug, but also the changes in all other biofluid metabolite concentrations. Knowing what other changes are occurring in the system is crucial to having a better understanding of the drug's effects and interactions before moving on with clinical trials.

8.3.4 Isotopic tracer analysis

Another possible future direction for applications of SpectConnect is in the analysis of stable isotopic tracers. These tracers work by increasing the amount of known stable isotopes (typically ^{13}C or ^{2}H) in a system by adding a labeled precursor. The flow or flux of that labeled precursor throughout the system can then be deduced by analyzing the labeling patterns at each specific metabolite in a pathway. These methods have seen wide use in the literature [118] as well as in our own laboratory and with collaborators [95, 13].

However, these approaches are only typically useful for small sections of metabolism that are well–known and have easily accessible precursors. It is desirable to investigate the broader labeling effects in a cell, but to do so for metabolites that have unknown mass spectrum is otherwise impossible. An extension of SpectConnect should allow

some sort of lumping that would facilitate the tracking and analysis of metabolite labeling in these flux experiments. Potential applications include a better understanding of metabolic fluxes as well as truly metabolomic (rather than just targeted metabolite profiling) approaches to flux analysis. This direction would require quite a bit of both experimental and computational work, but has quite a bit of promise for interesting and novel future applications.

Appendix A

Supplementary methods

A.1 Preparation and analysis of supplemented and control standard mixtures

A.1.1 Amino acid standard

A reference standard of 17 amino acids (Sigma #AAS18), of which 16 are derivatizable by the methyl chloroformate (MCF) method, was used. All amino acids except cystine are present at 2.5 μmol/mL; cystine (consisting of two disulfide-bonded cysteines) is present at 1.25 μmol/mL. 150 μL of the amino acid standard was combined in a 5-mL silanized glass tube with 8.5 μL of sodium hydroxide solution (2 M), 167 μL pure methanol, and 34 μL pyridine. The mixture was then derivatized with MCF as previously described [171].

A.1.2 Spiked amino acid mixture

Methylmalonic acid, α-ketoglutaric acid, 2-ketoisocaproic acid, lactic acid, 3-aminobutyric acid, 4- chloro-phenylalanine, citric acid, and stearic acid were added to methanol in equimolar amounts (total 2.3 μmol/mL). 167 μL of the spiked methanol solution were added to a silanized glass tube which also contained 150 μL of the amino acid mixture, 10 μL of sodium hydroxide solution (2 M), and 34 μL pyridine. The mixture

was then derivatized with MCF.

A.2 Preparation and analysis of *E. coli* strains

A.2.1 Bacterial strains and media

Three *Escherichia coli* strains were used. K12 PT5-dxs, PT5-idi, PT5-ispFD, provided by DuPont, served as the reference strain. The mutant strains K12 PT5-dxs, PT5-idi, PT5-ispFD, δgdhA, δaceE, δfdhF (mutant 1) and K12 PT5-dxs, PT5-idi, PT5-ispFD, δgdhA, δaceE, δPyjiD (mutant 2) [5], were also used. Reactor seed cultures were grown overnight in 50 ml cultures grown at 37°C with 225 RPM orbital shaking in either 2x M9-minimal medium [5] or R- medium [147] containing 5 g/L D-glucose and 68 μg/mL chloramphenicol. All experiments were performed in duplicate. Cell density was monitored spectrophotometrically at 600 nm. M9 Minimal salts were purchased from US Biological and all remaining chemicals were from Sigma-Aldrich.

A.2.2 Fermentation conditions

All fed-batch fermentations were conducted in 1.5-L Applikon vessels containing an initial volume of 500 mL (for M9-based cultures) or 600 mL (for R-medium-based cultures). A starting inoculum of 4% by volume (for M9-cultures) and 1% by volume (for R-medium) was used. pH was measured and controlled online using NH_4OH and HCl as appropriate for the desired control strategy. Temperature was regulated and controlled through water bath circulation. Glucose concentration was monitored and controlled online at a setpoint of 0.45 g/L using a YSI 2700 biochemistry analyzer connected to a pump with a feedstock of 200 g/L glucose with 0.1% antifoam (the set sampling interval was 20 minutes). Agitation speed was increased every two hours for the high cell density fermentations to obtain a stepwise increase from 400 RPM to 1100 RPM. Inlet air was supplied for aeration at a pressure of 15 psig.

A.2.3 Metabolite sampling, quenching, and derivatization

Pure methanol was precooled at −80°C and used to quench culture samples at 8, 12, 16, 24, and 30 hours into each fermentation. At each time point four 1-mL samples were taken. The samples were then frozen in liquid nitrogen, thawed at −40°C, and centrifuged at 3200 g for 10 min. The extract was collected and the cell debris was resuspended in 500 μL of precooled methanol ($-40C°$) and centrifuged again at 3200 g for 10 min. Having collected this extract, we pooled with the previous supernatant, and concentrated by vacuum centrifugation to approximately 150 μL. Each concentrated sample was transferred to a 5-mL silanized glass tube and combined with 80 μL of sodium hydroxide solution (2M), 167 μL of methanol, and 34 μL of pyridine. We derivatized the resulting mixture with MCF.

A.2.4 GC-MS analysis

We used a Hewlett-Packard system HP 5890 gas chromatograph coupled to an HP 5971 quadrupole mass selective detector (EI) operated at 70 eV. The column used for all analyses was a J&W DB-1701 (Folsom, CA, U.S.A.), 30 m × 250 μm (internal diameter) × 0.25 μm (film thickness). The MS was operated in scan mode (start after 10 min; mass range, 50-400 a.m.u. at 2.0 scans/s). We modified the analysis parameters from the original protocol described elsewhere [173]. The samples were injected in splitless mode. The oven temperature was originally held at 40°C for 2 min. Thereafter, the temperature was raised with a gradient of 5°C/min until 280°C was reached. This temperature was held for a final 10 minutes. The flow through the column was held constant at 0.9 ml He/min. The injection volume was 2 μL. The temperature of the inlet was 230°C, the interface temperature was 300°C, and the quadrupole temperature was 190°C. Injections were made in 4x replicate.

Appendix B

Supplementary methods

B.1 Preparation and analysis of *E. coli* strains

B.1.1 Bacterial strains and media

The *Saccharomyces cerevisiae* strain S288C was used for all experiments in the metabolome analysis project. Reactor seed cultures were grown overnight in 5 mL culture tubes, and then transferred to 50 mL cultures grown at 30°C with 225 RPM orbital shaking in YNB medium containing 20 g/L of glucose or galactose and 5 g/L ammonium sulfate. All experiments were performed in triplicate, most with culture medium controls. Cell density was monitored spectrophotometrically at 600 nm, and samples were typically taken (or perturbations performed) when OD_{600} was approximately 2. Heat shock perturbations were performed two ways: by transferring the culture flasks to an incubator at 37°C and by taking aliquots of fermentation cultures and transferring them to preheated fermentation medium. Salt perturbations were also performed two ways: by adding NaCl sufficient to make the culture's concentration 1 M, and by taking aliquots of fermentation cultures and transferring them to premixed fermentation media vessels such that the final concentration would be 1 M. YNB medium was purchased from US Biological and all remaining chemicals were from Sigma-Aldrich.

B.1.2 Metabolite sampling, quenching, and derivatization

A 75% methanol, 25% water mixture was precooled at −80°C and used to quench culture samples. For heat and salt shock perturbations, samples were taken either at 0 and 30 minutes after perturbation or at 0, 15, and 75 minutes after perturbation. For each sample, 8 mL of culture was added to 32 mL of quenching mixture. Samples were kept at −20°C or below in cold water baths or freezers. Samples were centrifuged at 3200 g for 10 minutes at −9°C, and the supernatant was discarded. The resulting yeast pellet was quickly washed with a 60% methanol, 40% water mixture; this wash mixture was removed and discarded, and the samples were returned to −80°C storage.

Extraction was performed by adding 2.5 mL methanol cooled at −80°C, 5 mL chloroform cooled at −80°C, and 2.0 mL of 12.5 mM tricine buffered at a pH of 7.4 and cooled on ice. These mixtures were than vortexed on high at −4°C. Extraction mixtures were then centrifuged again at −9°C and 3200 g for 10 minutes. The polar phase was pipetted off and transferred to a 5-mL glass tube. 2 mL tricine and 2 mL methanol were then added to the extraction mixture again; the mixture was vortexed for 15 seconds on high and then centrifuged again at −9°C and 3200 g for 10 minutes. The polar phase was pipetted off and pooled with previous polar phase. The resulting polar phase samples were then evaporated overnight.

Protecting groups were added before derivatization by adding 50 μL of methoxyamine HCl (20 mg/mL pyridine) to the dried metabolites. This mixture was incubated at 30°C for 90 minutes. We then added 80 μL of MSTFA + 1% TMCS and heated at 37°C for 30 minutes. We then syringe filtered the resulting mixture into microinserts in GC–MS vials.

B.1.3 GC-MS analysis

We used a Hewlett-Packard system HP 5890 gas chromatograph coupled to an HP 5971 quadrupole mass selective detector (EI) operated at 70 eV. The column used for all analyses was a J&W DB-35MS (Folsom, CA, U.S.A.), 30 m × 250 μm (internal diameter) × 0.25 μm (film thickness). The MS was operated in scan mode

(start after 10 min; mass range, 50-550 a.m.u. at 2.0 scans/s). The samples were injected in splitless mode. The oven temperature was originally held at 70°C for 5 min. Thereafter, the temperature was raised with a gradient of 5°C/min until 310°C was reached. This temperature was held for final minute. The flow through the column was held constant at 1.0 mL He/min. The injection volume was 1 μL. The temperature of the inlet was 230°C, the interface temperature was 250°C, and the quadrupole temperature was 200°C. Injections were made in 3x replicate.

Appendix C

Gemoda file documentation

C.1 Introduction

This chapter contains detailed documentation of the source code implementation of the Gemoda algorithm described in Chapter 3. Our implementation of Gemoda is written in the C programming language. As noted in Chapter 3, the different phases in the algorithm are completely modular. First of all, this means that any comparison metric can be used with any clustering algorithm and any convolution scheme. Second, this means that our program is extremely extensible. For example, if a user wants to use a new type of data or solve a new type of problem that requires a unique comparison function, all they need to do is write that comparison function in C; with only minor adjustments, Gemoda will then be able to use those comparison functions just as any other functions that come with the program.

This implementation makes uses the GNU Scientific Library [58] and the Basic Linear Algebra Subprograms (BLAS) [27, 43, 44] for optimal computational efficiency of complex mathematical operations like singular value decomposition. Such operations are necessary in, for instance, the analysis of protein secondary structure motifs. We have utilized some of the data structures provided in those libraries for certain analyses, while other data structures have been created and customized specifically

for the purposes of Gemoda.

Gemoda source code is available at `http://web.mit.edu/bamel/gemoda`. The software package includes a number of "helper" applications that increase interoperability with common bioinformatics tools.

This software is designed for UNIX–like systems and uses the GNU autotools framework for managing installation tasks and properly configuring itself for different computer architectures. Gemoda is distributed with a `configure` shell script that tries to guess system–dependent variables and to create a "makefile" that can be used as an input for GNU make.

Gemoda can be installed by using the following steps:

1. Change directories to the folder that contains the "src" directory as a subfolder. From this location, run the command `./configure`. To install Gemoda to a nonstandard location, use the optional flag `--prefix=PATH`, where `PATH` is the desired location, such as "/usr/local/software".

2. Type `make` to compile the software using your default C compiler, which is specified by the "CC" environment variable.

3. Type `make install` to install the software.

There are many other options for the `configure` script. To see a list of available options, use the optional flag `--help`.

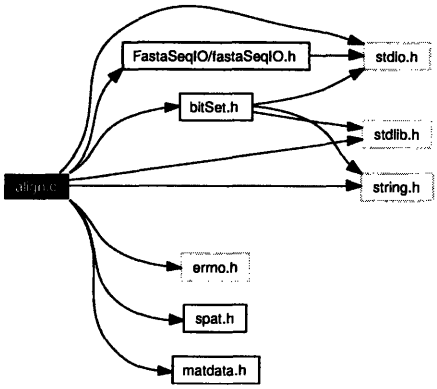
The remainder of this appendix contains detailed explanations of the organization and design of our software implementation of Gemoda. These sections are organized by file and are designed to show the dependencies and interactions of different functions.

C.2 align.c File Reference

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "FastaSeqIO/fastaSeqIO.h"
#include "spat.h"
#include "bitSet.h"
#include "matdata.h"
```

Include dependency graph for align.c:



Defines

- #define ALIGN_ALPHABET 256

Functions

- int alignMat (char *s1, char *s2, int L, int mat[][MATRIX_SIZE])
- bitGraph_t * alignWordsMat_bit (sPat_t *words, int wc, int mat[][MATRIX_SIZE], int threshold)

Variables

- const int aaOrder []

Detailed Description

This file defines functions that are used to create a similarity graph, or adjacency matrix via the comparison of small windows within a set of sequences. This file is

only used for string based sequences, and not real valued data. Usually, the adjacency matrix is created via a the alignment of the windows within the sequence set. Thus, the name of this file. However, other functions can certainly be defined for creating the adjacency matrix.

Definition in file align.c.

Define Documentation

C.2.0.1 #define ALIGN_ALPHABET 256

Definition at line 24 of file align.c.

Function Documentation

C.2.0.2 int alignMat (char * *s1*, char * *s2*, int *L*, int *mat*[][MATRIX_SIZE])

This function takes as its arguments two pointers to strings, a length, and a scoring matrix. The function computes the score, or degree of similarity, between the two strings by comparing each character the in the strings from zero two L minus one. Each character receives a score that is looked up in the scoring matrix. This is most commonly used for amino acid sequences or DNA sequences; however, it is applicable to any series of characters. This function returns a single integer, which is the score between the two words.

Definition at line 44 of file align.c.

References aaOrder, and mat.

Referenced by alignWordsMat_bit().

```
45 {
46   int i;
47   int points = 0;
48   int x, y;
49
50     // Go over each character in the L-length window
51     for (i = 0; i < L; i++)
52     {
53
54     // The integer corresponding to the character in
55     // the first string, so that we can look it up
56     // in one of our scoring matricies.
57     x = aaOrder[(int) s1[i]];
58
59     // And for the second character
60     y = aaOrder[(int) s2[i]];
61
62     // If the characters aren't going to be in the scoring
63     // matrix, they get a -1 value...which we'll give zero
64     // points to here.
65     if (x != -1 && y != -1)
```

```
66     {
67
68         // Otherwise, they get a score that is looked up
69         // in the scoring matrix
70         points += mat[x][y];
71     }
72     }
73   return points;
74 }
```

C.2.0.3 bitGraph_t* alignWordsMat_bit (sPat_t * *words*, int *wc*, int *mat*[][MATRIX_SIZE], int *threshold*)

This uses the function above. Here, we have an array of words (sPat_t objects) and we compare (align) them all. If their score is above 'threshold' then we will set a bit to 'true' in a bitGraph_t that we create. A bitGraph_t is essentially an adjacency matrix, where each member of the matrix contains only a single bit: are the words equal, true or false? The function traverses the words by doing and all by all comparison; however, we only do the upper diagonal. The function makes use of alignMat and needs to be passed a scoring matrix that the user has chosen which is appropriate for the context of whatever data sent the user is looking at.

Definition at line 88 of file align.c.

References alignMat(), bitGraphSetTrueSym(), mat, and newBitGraph().

Referenced by main().

```
90 {
91   bitGraph_t * sg = NULL;
92   int score;
93   int i, j;
94
95     // Assign a new bitGraph_t object, with (wc x wc) possible
96     // true/false values
97     sg = newBitGraph (wc);
98   for (i = 0; i < wc; i++)
99     {
100       for (j = i; j < wc; j++)
101     {
102
103         // Get the score for the alignment of word i and word j
104         score =
105         alignMat (words[i].string, words[j].string, words[i].length, mat);
106
107         // If that score is greater than threshold, set
108         // a bit to 'true' in our bitGraph_t object
109         if (score >= threshold)
110         {
111
112         // We use 'bitGraphSetTrueSym' because, if i=j,
113         // then j=i for most applications.  However, this
114         // can be relaxed for masochists.
115         bitGraphSetTrueSym (sg, i, j);
116         }
117     }
118     }
119
120     // Return a pointer to this new bitGraph_t object
121     return sg;
```

```
122 }
```

Variable Documentation

C.2.0.4 const int aaOrder[]

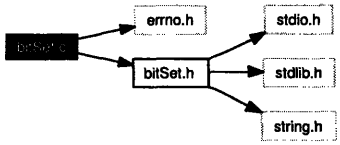
Definition at line 32 of file matrices.h.

Referenced by alignMat().

C.3 bitSet.c File Reference

```
#include "errno.h"
#include "bitSet.h"
```

Include dependency graph for bitSet.c:



Functions

- bit_t * newBitArray (int bytes)
- bitSet_t * newBitSet (int size)
- int setTrue (bitSet_t *s1, int x)
- int setFalse (bitSet_t *s1, int x)
- int flipBits (bitSet_t *s1)
- int fillSet (bitSet_t *s1)
- int emptySet (bitSet_t *s1)
- int checkBit (bitSet_t *s1, int x)
- int deleteBitSet (bitSet_t *s1)
- int bitSetUnion (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int copySet (bitSet_t *s1, bitSet_t *s2)
- int copyBitGraph (bitGraph_t *bg1, bitGraph_t *bg2)
- int bitSetDifference (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int bitSetSum (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int bitSetIntersection (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3)
- int bitSet3WayIntersection (bitSet_t *s1, bitSet_t *s2, bitSet_t *s3, bitSet_t *s4)
- int bitcount32 (unsigned int n)
- int bitcount32_precomp (unsigned int n)
- int bitcount64 (unsigned int n)
- int countSet (bitSet_t *s1)
- int nextBitBitSet (bitSet_t *s1, int start)
- int countBitGraphNonZero (bitGraph_t *bg)
- int printBitSet (bitSet_t *s1)
- int bitGraphRowUnion (bitGraph_t *bg, int row1, int row2, bitSet_t *s1)
- int bitGraphRowIntersection (bitGraph_t *bg, int row1, int row2, bitSet_t *s1)
- int printBinaryBitSet (bitSet_t *s1)

- int bitGraphCheckBit (bitGraph_t *bg, int x, int y)
- int bitGraphSetTrue (bitGraph_t *bg, int x, int y)
- int bitGraphSetFalse (bitGraph_t *bg, int x, int y)
- int bitGraphSetFalseSym (bitGraph_t *bg, int x, int y)
- int bitGraphSetTrueSym (bitGraph_t *bg, int x, int y)
- int bitGraphSetTrueDiagonal (bitGraph_t *bg)
- int bitGraphSetFalseDiagonal (bitGraph_t *bg)
- int printBitGraph (bitGraph_t *bg)
- int maskBitGraph (bitGraph_t *bg1, bitSet_t *bs)
- int fillBitGraph (bitGraph_t *bg1)
- int emptyBitGraph (bitGraph_t *bg1)
- bitGraph_t * newBitGraph (int size)
- int emptyBitGraphRow (bitGraph_t *bg, int row)
- int deleteBitGraph (bitGraph_t *bg)

Detailed Description

This file defines functions for handling bit sets and bit graphs.

Definition in file bitSet.c.

Function Documentation

C.3.0.5 int bitcount32 (unsigned int *n*)

Attempt at a fast way of counting how many true values are in a given bitSet_t. Currently deprecated, using precompiled version instead.

Definition at line 351 of file bitSet.c.

```
352 {
353   /*
354      works for 32-bit numbers only
355    */
356   /*
357      fix last line for 64-bit numbers
358    */
359
360   register unsigned int tmp;
361
362   tmp = n - ((n >> 1) & 033333333333) - ((n >> 2) & 011111111111);
363   return ((tmp + (tmp >> 3)) & 030707070707) % 63;
364 }
```

C.3.0.6 int bitcount32_precomp (unsigned int *n*)

Uses bits_in_char data structure to determine the number of true bits in a 32-bit int in an efficient manner. Input: 32-bit int (equal to one slot in the bitSet). Output: number of true bits in the input integer.

Definition at line 396 of file bitSet.c.

Referenced by countSet().

```
397 {
398   // works only for 32-bit ints
399
400   return bits_in_char[n & 0xffu]
401     + bits_in_char[(n >> 8) & 0xffu]
402     + bits_in_char[(n >> 16) & 0xffu] + bits_in_char[(n >> 24) & 0xffu];
403 }
```

C.3.0.7 int bitcount64 (unsigned int *n*)

Currently there is no support for 64-bit architectures.

Definition at line 420 of file bitSet.c.

```
421 {
422   n = PCCOUNT (n, 0);
423   n = PCCOUNT (n, 1);
424   n = PCCOUNT (n, 2);
425   n = PCCOUNT (n, 3);
426   n = PCCOUNT (n, 4);
427   n = PCCOUNT (n, 5);       // for 64-bit integers
428   return n;
429 }
```

C.3.0.8 int bitGraphCheckBit (bitGraph_t * *bg*, int *x*, int *y*)

Checks the value of a bit in a bitGraph_t object. Input: a bitGraph_t object, the index of the row of the bitGraph_t with the bit to be checked, the index of the bit in that row that is to be checked. Output: the value of the bit in the bitGraph being checked.

Definition at line 628 of file bitSet.c.

References checkBit(), and bitGraph_t::graph.

Referenced by main(), and measureDiagonal().

```
629 {
630   return checkBit (bg->graph[x], y);
631 }
```

C.3.0.9 int bitGraphRowIntersection (bitGraph_t * *bg*, int *row1*, int *row2*, bitSet_t * *s1*)

Finds the intersection of two rows (bitSets) within a bitGraph_t object. Input: a bitGraph_t object, first row to be compared, second row to be compared, and a bitSet_t to store the intersection results. Output: integer success value of 0 (and an altered destination bitSet_t object with a true value wherever both source bitSets had a true value).

Definition at line 598 of file bitSet.c.

References bitSetIntersection(), and bitGraph_t::graph.

Referenced by getStatMat(), and oldGetStatMat().

```
599 {
600   bitSetIntersection (bg->graph[row1], bg->graph[row2], s1);
601   return 0;
602 }
```

C.3.0.10 int bitGraphRowUnion (bitGraph_t * *bg*, int *row1*, int *row2*, bitSet_t * *s1*)

Finds the union of two rows (bitSets) within a bitGraph Input: a bitGraph_t object, first row to be compared, second row to be compared, and a bitSet_t to store the union results. Output: integer success value of 0 (and an altered destination bitSet_t object with a true value wherever one or both source bitSets had a true value).

Definition at line 584 of file bitSet.c.

References bitSetUnion(), and bitGraph_t::graph.

```
585 {
586   bitSetUnion (bg->graph[row1], bg->graph[row2], s1);
587   return 0;
588 }
```

C.3.0.11 int bitGraphSetFalse (bitGraph_t * *bg*, int *x*, int *y*)

Sets a specific bit in a bitGraph false. Input: a bitGraph_t object, the index of the row of the bitGraph_t with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 654 of file bitSet.c.

References bitGraph_t::graph, and setFalse().

```
655 {
656   setFalse (bg->graph[x], y);
657   return 0;
658 }
```

C.3.0.12 int bitGraphSetFalseDiagonal (bitGraph_t * *bg*)

Sets the main diagonal of a bitGraph false. Input: a bitGraph_t object. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 714 of file bitSet.c.

References bitGraph_t::graph, and setFalse().

Referenced by convolve().

```
715 {
716   int i;
717   for (i = 0; i < bg->size; i++)
718     {
719       setFalse (bg->graph[i], i);
720     }
721   return 0;
722 }
```

C.3.0.13 int bitGraphSetFalseSym (bitGraph_t * *bg*, int *x*, int *y*)

Sets a specific bit and its symmetric opposite in a bitGraph false. For instance, given that we wanted to set the 3rd bit in the 5th row false, this would also set the 5th bit in the 3rd row. Input: a bitGraph_t object, the index of the row of the bitGraph with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 669 of file bitSet.c.

References bitGraph_t::graph, and setFalse().

```
670 {
671   setFalse (bg->graph[x], y);
672   setFalse (bg->graph[y], x);
673   return 0;
674 }
```

C.3.0.14 int bitGraphSetTrue (bitGraph_t * *bg*, int *x*, int *y*)

Sets a specific bit in a bitGraph true. Input: a bitGraph_t object, the index of the row of the bitGraph_t with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 641 of file bitSet.c.

References bitGraph_t::graph, and setTrue().

```
642 {
643   setTrue (bg->graph[x], y);
644   return 0;
645 }
```

C.3.0.15 int bitGraphSetTrueDiagonal (bitGraph_t * *bg*)

Sets the main diagonal of a bitGraph true. Input: a bitGraph_t object. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 698 of file bitSet.c.

References bitGraph_t::graph, and setTrue().

```
699 {
700   int i;
701   for (i = 0; i < bg->size; i++)
702     {
703       setTrue (bg->graph[i], i);
704     }
705   return 0;
706 }
```

C.3.0.16 int bitGraphSetTrueSym (bitGraph_t * *bg*, int *x*, int *y*)

Sets a specific bit and its symmetric opposite in a bitGraph true. For instance, given that we wanted to set the 3rd bit in the 5th row true, this would also set the 5th bit in the 3rd row. Input: a bitGraph, the index of the row of the bitGraph with the bit be set, the index of the bit in that row that is to be set. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 685 of file bitSet.c.

References bitGraph_t::graph, and setTrue().

Referenced by alignWordsMat_bit(), main(), and realComparison().

```
686 {
687   setTrue (bg->graph[x], y);
688   setTrue (bg->graph[y], x);
689   return 0;
690 }
```

C.3.0.17 int bitSet3WayIntersection (bitSet_t * *s1*, bitSet_t * *s2*, bitSet_t * *s3*, bitSet_t * *s4*)

Finds the intersection of 3 bitSets. Input: First bitSet to be intersected, second bitset to be intersected. third bitSet to be intersected, a bitSet to store the result of the intersection. Output: Integer success value of 0 (and an altered destination bitSet_t object with a true where all three source bitSets had a true.)

Definition at line 327 of file bitSet.c.

References BSINTERSECTION, bitSet_t::slots, and bitSet_t::tf.

```
329 {
330   int i;
331   if ((s1->slots != s2->slots) || (s1->slots != s3->slots)
```

```
332       || (s1->slots != s4->slots))
333     {
334       fprintf (stderr, "Sets aren't same size!\n");
335       fflush (stderr);
336       exit (0);
337     }
338   for (i = 0; i < s1->slots; i++)
339     {
340       s4->tf[i] = BSINTERSECTION (s1->tf[i], s2->tf[i]);
341       s4->tf[i] = BSINTERSECTION (s3->tf[i], s4->tf[i]);
342     }
343   return 0;
344 }
```

C.3.0.18 int bitSetDifference (bitSet_t * *s1*, bitSet_t * *s2*, bitSet_t * *s3*)

Locates all differences between two bitSets. The result bitSet contains a true at a given bit if the two source bitSets differ at that bit. Input: first bit set to be compared, second bit set to be compared. third bit set to store the results Output: integer success value of 0 (and an altered destination bitSet_t object with a true where the two source bit sets differed).

Definition at line 254 of file bitSet.c.

References bitSet_t::slots, and bitSet_t::tf.

```
255 {
256   int i;
257   if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
258     {
259       fprintf (stderr, "Sets aren't same size!\n");
260       fflush (stderr);
261       exit (0);
262     }
263   for (i = 0; i < s1->slots; i++)
264     {
265       s3->tf[i] = (s1->tf[i] & (~s2->tf[i]));
266     }
267   return 0;
268 }
```

C.3.0.19 int bitSetIntersection (bitSet_t * *s1*, bitSet_t * *s2*, bitSet_t * *s3*)

Finds the intersection of two bitsets. Input: First bitSet to be intersected, second bitSet to be intersected. a bitSet to store the result of the intersection. Output: Integer success value of 0 (and an altered destination bitSet_t object. with a true where both source bitSets had a true).

Definition at line 299 of file bitSet.c.

References BSINTERSECTION, bitSet_t::slots, and bitSet_t::tf.

Referenced by bitGraphRowIntersection(), findCliques(), and maskBitGraph().

```
300 {
```

```
301   int i;
302   if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
303     {
304       fprintf (stderr, "Sets aren't same size!\n");
305       fprintf (stderr, "set 1 slots = %d\n", s1->slots);
306       fprintf (stderr, "set 2 slots = %d\n", s2->slots);
307       fprintf (stderr, "set 3 slots = %d\n", s3->slots);
308       fflush (stderr);
309       exit (0);
310     }
311   for (i = 0; i < s1->slots; i++)
312     {
313       s3->tf[i] = BSINTERSECTION (s1->tf[i], s2->tf[i]);
314     }
315   return 0;
316 }
```

C.3.0.20 int bitSetSum (bitSet_t * *s1*, bitSet_t * *s2*, bitSet_t * *s3*)

Adds two bitSet_t objects together. Currently unknown functionality, not used in existing code.

Definition at line 275 of file bitSet.c.

References bitSet_t::slots, and bitSet_t::tf.

```
276 {
277   int i;
278   if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
279     {
280       fprintf (stderr, "Sets aren't same size!\n");
281       fflush (stderr);
282       exit (0);
283     }
284   for (i = 0; i < s1->slots; i++)
285     {
286       s3->tf[i] = (s1->tf[i] + s2->tf[i]);
287     }
288   return 0;
289 }
```

C.3.0.21 int bitSetUnion (bitSet_t * *s1*, bitSet_t * *s2*, bitSet_t * *s3*)

Finds the union of two bitSets Input: first bit set for the union, second bit set for the union. a bit set in which to store the results Output: an integer success value of 0 (and an altered third bitSet_t with the results of the union.

Definition at line 182 of file bitSet.c.

References BSUNION, bitSet_t::slots, and bitSet_t::tf.

Referenced by bitGraphRowUnion(), and singleLinkage().

```
183 {
184   int i;
185   if ((s1->slots != s2->slots) || (s1->slots != s3->slots))
186     {
187       fprintf (stderr, "Sets aren't same size!\n");
```

```
188       fflush (stderr);
189       exit (0);
190     }
191   for (i = 0; i < s1->slots; i++)
192     {
193       s3->tf[i] = BSUNION (s1->tf[i], s2->tf[i]);
194     }
195   return 0;
196 }
```

C.3.0.22 int checkBit (bitSet_t * *s1*, int *x*)

Finds the value of a specific bit in a bitSet. Input: a bitSet, the number of the bit being queried. Output: the value of the bit being queried (1 or 0).

Definition at line 148 of file bitSet.c.

References BSTEST, and bitSet_t::tf.

Referenced by bitGraphCheckBit(), findCliques(), getStatMat(), maskBitGraph(), nextBitBitSet(), singleLinkage(), and wholeRoundConv().

```
149 {
150   return BSTEST (s1->tf, x);
151 }
```

C.3.0.23 int copyBitGraph (bitGraph_t * *bg1*, bitGraph_t * *bg2*)

Copies the true/false contents of one bit graph into an existing bit graph. Both bit graphs must be the same size, and each corresponding bit set between the two bit graphs must be the same size. Input: source bit graph, destination bitGraph_t object. Output: integer success value of 0 (and an altered destination bit graph).

Definition at line 229 of file bitSet.c.

References copySet(), bitGraph_t::graph, and bitGraph_t::size.

```
230 {
231   int i;
232   if (bg1->size != bg2->size)
233     {
234       fprintf (stderr, "Graphs are not the same size!");
235       fflush (stderr);
236       exit (0);
237     }
238   for (i = 0; i < bg1->size; i++)
239     {
240       copySet (bg1->graph[i], bg2->graph[i]);
241     }
242   return 0;
243 }
```

C.3.0.24 int copySet (bitSet_t * *s1*, bitSet_t * *s2*)

Copies the true/false contents of one bit set into an existing bit set. Both bit sets must be the same size. Input: source bit set, destination bitSet_t object. Output: integer success value of 0 (and an altered destination bitset.

Definition at line 205 of file bitSet.c.

References bitSet_t::slots, and bitSet_t::tf.

Referenced by copyBitGraph(), filterGraph(), and singleLinkage().

```
206 {
207   int i;
208   if (s1->slots != s2->slots)
209     {
210       fprintf (stderr, "Sets are not the same size!");
211       fflush (stderr);
212       exit (0);
213     }
214   for (i = 0; i < s1->slots; i++)
215     {
216       s2->tf[i] = s1->tf[i];
217     }
218   return 0;
219 }
```

C.3.0.25 int countBitGraphNonZero (bitGraph_t * *bg*)

Counts the number of true (non-zero) values in a bitGraph_t object. Input: a bitGraph_t object. Output: the integer number of true (non-zero) values in the bitGraph_t object.

Definition at line 537 of file bitSet.c.

References countSet(), and bitGraph_t::graph.

```
538 {
539   int i;
540   int sum = 0;
541   // Iterate over all bitSets in the bitGraph
542   for (i = 0; i < bg->size; i++)
543     {
544       sum += countSet (bg->graph[i]);
545     }
546   return sum;
547 }
```

C.3.0.26 int countSet (bitSet_t * *s1*)

Counts the number of true values in a bitSet. Input: a bitSet_t object. Output: number of true values in that bitSet_t object.

Definition at line 437 of file bitSet.c.

References bitcount32_precomp(), and bitSet_t::tf.

Referenced by bitSetToCSet(), countBitGraphNonZero(), filterGraph(), filterIter(), findCliques(), getStatMat(), oldGetStatMat(), printBitSet(), singleLinkage(), and wholeCliqueConv().

```
438 {
439   int i;
440   int sum = 0;
441   int (*bitCounter) () = &bitcount32_precomp;
442   // Currently there is no support for 64-bit architectures.
443
444   if (sizeof (bit_t) * 8 != 32)
445     {
446       fprintf (stderr,
447            "\nSorry, no support for 64-bit architectures just yet! - countSet\n");
448       fflush (stderr);
449       exit (0);
450     }
451
452   // Just count the number of true bits in each char, and do this for
453   // (num of chars per int) chars.
454   for (i = 0; i < s1->slots; i++)
455     {
456       sum += bitCounter (s1->tf[i]);
457     }
458   return sum;
459 }
```

C.3.0.27 int deleteBitGraph (bitGraph_t * *bg*)

Deletes a bitGraph_t object from memory. Input: a bitGraph_t object to be deleted. Output: integer success value from 0 (and deletion of a bitGraph_t object).

Definition at line 853 of file bitSet.c.

References deleteBitSet(), and bitGraph_t::graph.

Referenced by main().

```
854 {
855   int i;
856   if (bg != NULL)
857     {
858       if (bg->graph != NULL)
859     {
860       for (i = 0; i < bg->size; i++)
861         {
862           deleteBitSet (bg->graph[i]);
863         }
864       free (bg->graph);
865       bg->graph = NULL;
866     }
867       free (bg);
868       bg = NULL;
869     }
870   return 0;
871 }
```

C.3.0.28 int deleteBitSet (bitSet_t * *s1*)

Performs memory management for the deletion of a bitSet_t structure. Input: a bitSet_t object. Output: integer success value of 1.

Definition at line 159 of file bitSet.c.

References bitSet_t::tf.

Referenced by convolve(), deleteBitGraph(), filterGraph(), findCliques(), getStatMat(), oldGetStatMat(), wholeCliqueConv(), and wholeRoundConv().

```
160 {
161   if (s1->tf != NULL)
162     {
163       free (s1->tf);
164       s1->tf = NULL;
165     }
166   if (s1 != NULL)
167     {
168       free (s1);
169       s1 = NULL;
170     }
171   return 0;
172 }
```

C.3.0.29 int emptyBitGraph (bitGraph_t * *bg1*)

Sets all bits in the bitGraph_t object to false. Input: a bitGraph_t object. Output: integer success value of 0 (and a bitGraph_t with all false bits).

Definition at line 791 of file bitSet.c.

References emptySet(), and bitGraph_t::graph.

```
792 {
793   int i;
794   for (i = 0; i < bg1->size; i++)
795     {
796       emptySet (bg1->graph[i]);
797     }
798   return 0;
799 }
```

C.3.0.30 int emptyBitGraphRow (bitGraph_t * *bg*, int *row*)

Sets all bits in a bitGraph_t row (a bitSet_t object) false. Input: a bitGraph, a row in the bitGraph_t object to be emptied. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 841 of file bitSet.c.

References emptySet(), and bitGraph_t::graph.

```
842 {
```

```
843   emptySet (bg->graph[row]);
844   return 0;
845 }
```

C.3.0.31 int emptySet (bitSet_t * *s1*)

Sets all values in a bitSet to false. Input: a bitSet_t object. Output: integer success value of 1.

Definition at line 136 of file bitSet.c.

References bitSet_t::bytes, and bitSet_t::tf.

Referenced by emptyBitGraph(), emptyBitGraphRow(), filterGraph(), filterIter(), maskBitGraph(), pruneBitGraph(), and searchMemsWithList().

```
137 {
138   memset (s1->tf, 0, s1->bytes);
139   return 0;
140 }
```

C.3.0.32 int fillBitGraph (bitGraph_t * *bg1*)

Sets all bits in the bitGraph_t object to true. Input: a bitGraph_t object. Output: integer success value of 0 (and a bitGraph_t object with all true bits).

Definition at line 775 of file bitSet.c.

References fillSet(), and bitGraph_t::graph.

```
776 {
777   int i;
778   for (i = 0; i < bg1->size; i++)
779     {
780       fillSet (bg1->graph[i]);
781     }
782   return 0;
783 }
```

C.3.0.33 int fillSet (bitSet_t * *s1*)

Sets all values in a bitSet to true. Input: a bitSet. Output: integer success value of 1.

Definition at line 124 of file bitSet.c.

References bitSet_t::bytes, and bitSet_t::tf.

Referenced by convolve(), fillBitGraph(), and wholeRoundConv().

```
125 {
126   memset (s1->tf, ~0, s1->bytes);
127   return 0;
128 }
```

C.3.0.34 int flipBits (bitSet_t * *s1*)

Inverts all values in a bitSet, making all trues false and all falses true. Input: a bitSet. Output: integer success value of 1.

Definition at line 108 of file bitSet.c.

References bitSet_t::tf.

```
109 {
110   int i;
111   for (i = 0; i < s1->slots; i++)
112     {
113       s1->tf[i] = ~s1->tf[i];
114     }
115   return 0;
116 }
```

C.3.0.35 int maskBitGraph (bitGraph_t * *bg1*, bitSet_t * *bs*)

Makes a bitGraph contain only true bits according to the bitmask given. Only locations with the row and column both true in the bitmask can be true if they were initially true. If they were false, they remain false. If the location does not have both the row and the column in the bitmask, it is made false. Note, this is not currently used in Gemoda. Input: a bitGraph, a mask in the form of a bitSet_t object. Output: integer success value of 0 (and an altered bitGraph_t object).

Definition at line 752 of file bitSet.c.

References bitSetIntersection(), checkBit(), emptySet(), and bitGraph_t::graph.

```
753 {
754   int i;
755   for (i = 0; i < bg1->size; i++)
756     {
757       if (checkBit (bs, i))
758     {
759       bitSetIntersection (bg1->graph[i], bs, bg1->graph[i]);
760     }
761       else
762     {
763       emptySet (bg1->graph[i]);
764     }
765     }
766   return 0;
767 }
```

C.3.0.36 bit_t* newBitArray (int *bytes*)

Creates a bit array for use in high-throughput intersections/unions. Input: desired size of bit array in byte. Output: a new bit array in bit_t forma. Note: this should not be called directly; see newBitSet.

Definition at line 20 of file bitSet.c.

Referenced by newBitSet().

```
21 {
22   bit_t *b = (bit_t *) malloc (bytes);
23   if (b == NULL)
24     {
25       fprintf (stderr, "\nMemory error --- couldn't allocate bitArray!"
26            " - newBitArray\n%s\n", strerror (errno));
27       fflush (stderr);
28       exit (0);
29     }
30   // Set them all false
31   memset (b, 0, bytes);
32   return b;
33 }
```

C.3.0.37 bitGraph_t* newBitGraph (int *size*)

Creates a bitGraph_t data structure. Input: the size of the (square) bitGraph_t object. Output: a new bitGraph_t data structure.

Definition at line 807 of file bitSet.c.

References bitGraph_t::graph, newBitSet(), and bitGraph_t::size.

Referenced by alignWordsMat_bit(), main(), and realComparison().

```
808 {
809   bitGraph_t *bg = NULL;
810   int i;
811   bg = (bitGraph_t *) malloc (sizeof (bitGraph_t));
812   if (bg == NULL)
813     {
814       fprintf (stderr, "Memory error - Cannot allocate bitGraph - "
815            "newBitGraph\n%s\n", strerror (errno));
816       fflush (stderr);
817       exit (0);
818     }
819   bg->size = size;
820   bg->graph = (bitSet_t **) malloc (size * sizeof (bitSet_t *));
821   if (bg->graph == NULL)
822     {
823       fprintf (stderr, "Memory error - Cannot allocate bitGraphGraph - "
824            "newBitGraph\n%s\n", strerror (errno));
825       fflush (stderr);
826       exit (0);
827     }
828   for (i = 0; i < size; i++)
829     {
830       bg->graph[i] = newBitSet (size);
831     }
832   return bg;
833 }
```

C.3.0.38 bitSet_t* newBitSet (int *size*)

Creates a bitSet data structure that contains a bit array and information about that bit array that is necessary for quick and efficient access of the array. Input: the desired length of the bit array. Output: a bitSet data structure.

Definition at line 43 of file bitSet.c.

References BSNUMSLOTS, bitSet_t::bytes, bitSet_t::max, newBitArray(), bitSet_t::slots, and bitSet_t::tf.

Referenced by convolve(), filterGraph(), findCliques(), getStatMat(), newBitGraph(), oldGetStatMat(), wholeCliqueConv(), and wholeRoundConv().

```
44 {
45   bitSet_t *s1 = (bitSet_t *) malloc (sizeof (bitSet_t));
46   if (s1 == NULL)
47     {
48       fprintf (stderr, "\nMemory error --- couldn't allocate biSet!"
49            " - newBitSet\n%s\n", strerror (errno));
50       fflush (stderr);
51       exit (0);
52     }
53   // Fill in details about the bitSet, allocate bitSet
54   s1->max = size;
55   s1->slots = BSNUMSLOTS (size);
56   s1->bytes = s1->slots * sizeof (bit_t);
57   s1->tf = newBitArray (s1->bytes);
58   return s1;
59 }
```

C.3.0.39 int nextBitBitSet (bitSet_t * *s1*, int *start*)

Finds the index of the first non-zero bit at-or-after start. Input: a bitSet_t to be searched, the index of the start bit. Output: the index of the first non-zero bit at-or-after start.

Definition at line 468 of file bitSet.c.

References BITSLOT, BSBITSIZE, checkBit(), bitSet_t::max, and bitSet_t::tf.

Referenced by bitSetToCSet(), filterIter(), findCliques(), getStatMat(), pruneBitGraph(), and singleLinkage().

```
469 {
470   // slot is our starting slot, the
471   // slot containing bit 'start'
472   int slot = BITSLOT (start);
473   int i;
474   // stop is the bit to stop it --- it is equal to max, and it is
475   // the index of a bit that does NOT belong to the bitset
476   int stop;
477   bit_t bitFalse;
478   memset (&bitFalse, 0, sizeof (bit_t));
479
480
481   // s1->max is the number of bits in s1
482   // test to see if we're looking too high
483   if (start >= s1->max)
484     {
485       return -1;
486     }
487   // s1->slots is the number of available slots
488   // skip over empty slots
489   while (slot < s1->slots)
490     {
491       /*
492          printf("w");
```

```
493       */
494      if (s1->tf[slot] != bitFalse)
495    {
496      // this slot is not empty
497
498      // if each slot is, say 32 bits and
499      // we asked for nextBitBitSet(s1, 5),
500      // then slot 0 will be non-zero.  but,
501      // instead of starting at 0, start at 5!
502      if (BSBITSIZE * slot > start)
503        {
504          // set start to index of first
505          // bit in this slot
506          start = BSBITSIZE * slot;
507        }
508      // set the stop, with a a check against the 'max'
509      // element of the bitSet_t object
510      if (BSBITSIZE * (slot + 1) > s1->max)
511        {
512          stop = s1->max;
513        }
514      else
515        {
516          stop = BSBITSIZE * (slot + 1);
517        }
518      for (i = start; i < stop; i++)
519        {
520          if (checkBit (s1, i))
521        {
522          return i;
523        }
524        }
525    }
526      slot++;
527    }
528  return -1;
529 }
```

C.3.0.40 int printBinaryBitSet (bitSet_t * *s1*)

Prints a representation of a bitSet_t structure as a string of 1's and 0's. Input: a bitSet_t object to be printed. Output: integer success value of 0 (and the stdout text described above).

Definition at line 611 of file bitSet.c.

References BSTEST, and bitSet_t::tf.

Referenced by printBitGraph().

```
612 {
613   int i;
614   for (i = 0; i < s1->max; i++)
615     {
616       printf ("%d", (BSTEST (s1->tf, i) ? 1 : 0));
617     }
618   return 0;
619 }
```

C.3.0.41 int printBitGraph (bitGraph_t * *bg*)

Prints a representation of a bitGraph using printBinaryBitSet. Input: a bitGraph_t object. Output: integer success value of 0 (and stdout text as described above).

Definition at line 730 of file bitSet.c.

References bitGraph_t::graph, and printBinaryBitSet().

```
731 {
732   int i;
733   for (i = 0; i < bg->size; i++)
734     {
735       printBinaryBitSet (bg->graph[i]);
736       printf ("\n");
737     }
738   return 0;
739 }
```

C.3.0.42 int printBitSet (bitSet_t * *s1*)

Prints a representation of a bitSet_t data structure. Input: a bitSet_t to be displayed. Output: integer success value of 0 (and the stdout text described above).

Definition at line 555 of file bitSet.c.

References BSTEST, and countSet().

```
556 {
557   int i;
558   printf ("bitSet (addr = %d; %d members)\n", (int) s1, countSet (s1));
559   printf ("\tmax = %d\n", s1->max);
560   printf ("\tslots = %d\n", s1->slots);
561   printf ("\tbytes = %d\n", s1->bytes);
562   printf ("\tmembers =");
563
564
565   for (i = 0; i < s1->max; i++)
566     {
567       if (BSTEST (s1->tf, i))
568     {
569       printf (" %d", i);
570     }
571     }
572   printf ("\n");
573   return 0;
574 }
```

C.3.0.43 int setFalse (bitSet_t * *s1*, int *x*)

Sets a specific bit in a bitSet as false. Input: a bitSet, the number of the bit to be set as false. Output: integer success value of 1.

Definition at line 85 of file bitSet.c.

References BSCLEAR, bitSet_t::max, and bitSet_t::tf.

Referenced by bitGraphSetFalse(), bitGraphSetFalseDiagonal(), bitGraphSetFalseSym(), filterIter(), findCliques(), singleCliqueConv(), and singleLinkage().

```
86 {
87   /*
88      if (BSNUMSLOTS(x) > s1->slots) { Conditional changed, 5/25, by MPS: check x against s1->max,
89      should be safer
90    */
91   if (x >= s1->max)
92     {
93       fprintf (stderr, "Set isn't large enough! - setFalse\n");
94       fflush (stderr);
95       exit (0);
96     }
97   BSCLEAR (s1->tf, x);
98   return 0;
99 }
```

C.3.0.44 int setTrue (bitSet_t * *s1*, int *x*)

Sets a specific bit in a bitSet as true. Input: a bitSet, the number of the bit to be set as true. Output: integer success value of 1.

Definition at line 67 of file bitSet.c.

References BSSET, bitSet_t::max, and bitSet_t::tf.

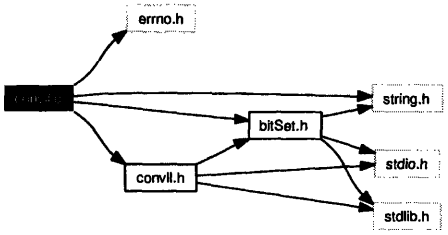
Referenced by bitGraphSetTrue(), bitGraphSetTrueDiagonal(), bitGraphSetTrueSym(), filterIter(), findCliques(), and setStackTrue().

```
68 {
69   if (x >= s1->max)
70     {
71       fprintf (stderr, "Set isn't large enough! - setTrue\n");
72       fflush (stderr);
73       exit (0);
74     }
75   BSSET (s1->tf, x);
76   return 0;
77 }
```

C.4 convll.c File Reference

```
#include <errno.h>
#include <string.h>
#include "convll.h"
#include "bitSet.h"
```

Include dependency graph for convll.c:



Functions

- cll_t * pruneCll (cll_t *head, int *indexToSeq, int p)
- cll_t * pushCll (cll_t *head)
- cll_t * popCll (cll_t *head)
- cll_t * popAllCll (cll_t *head)
- int printCll (cll_t *head)
- cll_t * initheadCll (cll_t *head, cSet_t *newset)
- cll_t * pushcSet (cll_t *head, cSet_t *newset)
- cSet_t * bitSetToCSet (bitSet_t *clique)
- int checkCliquecSet (cSet_t *cliquecSet, int *indexToSeq, int p)
- cll_t * pushClique (bitSet_t *clique, cll_t *head, int *indexToSeq, int p)
- mll_t * pushMemStack (mll_t *head, int cliqueNum)
- mll_t * popMemStack (mll_t *head)
- mll_t * popWholeMemStack (mll_t *head)
- mll_t ** addToStacks (cll_t *node, mll_t **memberStacks)
- mll_t ** fillMemberStacks (cll_t *head, mll_t **memberStacks)
- mll_t ** emptyMemberStacks (mll_t **memberStacks, int size)
- void printMemberStacks (mll_t **memberStacks, int size)
- bitSet_t * setStackTrue (mll_t **memList, int i, bitSet_t *queue)
- bitSet_t * searchMemsWithList (int *list, int listsize, mll_t **memList, int numOffsets, bitSet_t *queue)

- cll_t * singleCliqueConv (cll_t *head, int firstClique, cll_t **firstGuess, int secondClique, cll_t **secondGuess, cll_t *nextPhase, bitSet_t *printStatus, int support)
- mll_t * mergeIntersect (cll_t *first, cll_t *second, mll_t *intersection, bitSet_t *printstatus, int *newSupport)
- int uniqClique (cSet_t *cliquecSet, cll_t *head)
- cll_t * swapNodecSet (cll_t *head, int node, cSet_t *newClique)
- cll_t * removeSupers (cll_t *head, int node, cSet_t *newClique)
- int printCSet (cSet_t *node)
- cll_t * pushConvClique (mll_t *clique, cll_t *head)
- cSet_t * mllToCSet (mll_t *clique)
- cll_t * wholeCliqueConv (cll_t *head, cll_t *node, cll_t **firstGuess, mll_t **memList, int numOffsets, cll_t *nextPhase, bitSet_t *printStatus, int support)
- cll_t * wholeRoundConv (cll_t **head, mll_t **memList, int numOffsets, int support, int length, cll_t **allCliques)
- int yankCll (cll_t **head, cll_t *prev, cll_t **curr, cll_t **allCliques, int length)
- cll_t * completeConv (cll_t **head, int support, int numOffsets, int minLength, int *indexToSeq, int p)
- int printCllPattern (cll_t *node, int length)

Variables

- int cliquecounter = 0

Detailed Description

This file defines a number of functions for handling link lists of motifs, or cliques. The functions defined in this file are called extensively during the convolution stage of the Gemoda algorithm for both the sequence based and real value based software.

Definition in file convll.c.

Function Documentation

C.4.0.45 mll_t** addToStacks (cll_t * *node*, mll_t ** *memberStacks*)

For one clique, it adds membership for that clique to all of its members' member stacks. Input: a specific clique in a clique linked list, an array of member stacks. Output: the array of updated member stacks.

Definition at line 482 of file convll.c.

References cnode::id, cSet_t::members, pushMemStack(), and cnode::set.

Referenced by fillMemberStacks().

```
483 {
484   int i = 0;
485   int cliqueNum = 0;
486
487   // Make sure that we don't reference NULL values
488   if (node->set != NULL)
489     {
490       // Go through each member of the clique's set
491       for (i = 0; i < node->set->size; i++)
492     {
493       // Get the member's number
494       cliqueNum = node->set->members[i];
495       // Go to that member's linked list and push
496       // on the number of the current clique
497       memberStacks[cliqueNum] =
498         pushMemStack (memberStacks[cliqueNum], node->id);
499     }
500     }
501   else
502     {
503       fprintf (stderr, "\nNULL set for clique! - addToStacks\n");
504       fflush (stderr);
505       exit (0);
506     }
507   return memberStacks;
508 }
```

C.4.0.46 cSet_t* bitSetToCSet (bitSet_t * *clique*)

Converts a bitSet_t to a cSet_t for the purposes of pushing it onto a linked list of cliques. The bitSet_t data structure is used for massive comparisons during clique-finding but is unwieldy/inefficient when it is known that the structure is sparse. The cSet_t allows for efficient comparison of sparse bitSet_t's. Use this just before pushing a newly-discovered clique onto a clique linked list. Input: a new clique in the form of a bitSet_t. Output: the same clique in the form of a cSet_t.

Definition at line 212 of file convll.c.

References countSet(), cSet_t::members, nextBitBitSet(), and cSet_t::size.

Referenced by pushClique(), and wholeCliqueConv().

```
213 {
214   int cliqueSize = countSet (clique);
215   int i = 0, start = 0;
216   cSet_t *holder = (cSet_t *) malloc (sizeof (cSet_t));
217
218   // Memory error checking
219   if (holder == NULL)
220     {
221       fprintf (stderr, "\nMemory Error - bitSetToCSet - [1]\n%s\n",
222            strerror (errno));
223       fflush (stderr);
224       exit (0);
225     }
226   // More memory checking
```

```
227   holder->members = (int *) malloc (cliqueSize * sizeof (int));
228   if (holder->members == NULL)
229     {
230       fprintf (stderr, "\nMemory Error - bitSetToCSet - [2]\n%s\n",
231            strerror (errno));
232       fflush (stderr);
233       exit (0);
234     }
235
236   // For each member of the clique in the bitSet,
237   for (i = 0; i < cliqueSize; i++)
238     {
239       // Find the next one, add its location to the members array
240       holder->members[i] = nextBitBitSet (clique, start);
241       // (But check for errors... if we get to the end of the
242       // bitSet, then something is wrong)
243       if (holder->members[i] == -1)
244     {
245       fprintf (stderr, "\nClique error - not enough members\n");
246       fflush (stderr);
247       exit (0);
248     }
249       // Increment to move on in the nextBitBitSet search
250       start = holder->members[i] + 1;
251     }
252
253   holder->size = cliqueSize;
254   return holder;
255 }
```

C.4.0.47 int checkCliquecSet (cSet_t * *cliquecSet*, int * *indexToSeq*, int *p*)

Checks to enforce the -p flag (minimum number of unique input sequences in which the motif occurs). Input: a clique in the form of a cSet_t, pointer to the index/sequence number data structure, the -p flag value. Output: An integer: 1 for success, 0 for failure.

Definition at line 266 of file convll.c.

References cSet_t::members, and cSet_t::size.

Referenced by pushClique().

```
267 {
268   int *seqNums = NULL;
269   int thisSeq = 0, i = 0, j = 0;
270   seqNums = (int *) malloc (p * sizeof (int));
271
272   if (seqNums == NULL)
273     {
274       fprintf (stderr, "Memory error - checkCliquecSet\n%s\n",
275            strerror (errno));
276       fflush (stderr);
277       exit (0);
278     }
279   // Initialize an array of integers of size p to sentinel values of -1
280   for (i = 0; i < p; i++)
281     {
282       seqNums[i] = -1;
283     }
284   j = 0;
```

```
285
286   if (cliquecSet->size < 1)
287     {
288       fprintf (stderr, "\nClique of zero size! - checkCliquecSet\n");
289       fflush (stderr);
290       exit (0);
291     }
292   // Find the first sequence number.
293   seqNums[0] = indexToSeq[cliquecSet->members[0]];
294   // Iterate over the remaining size of the clique
295   for (i = 1; i < cliquecSet->size; i++)
296     {
297       // Find the next sequence number.
298       thisSeq = indexToSeq[cliquecSet->members[i]];
299       // The member list is in monotonic order, so we only need
300       // to compare the current member to the previous member to
301       // find out if it comes from the same sequence.
302       // If it's not from the same sequence, increment the unique
303       // sequence counter (j), store the next sequence number
304       // in the array.
305       // Also check to see if we've already reached the p threshold,
306       // and if so, then bail out.
307       if (thisSeq != seqNums[j])
308     {
309       j++;
310       seqNums[j] = thisSeq;
311       if (j == p - 1)
312         {
313           break;
314         }
315     }
316     }
317
318   // Now just see what the value of the last number in the array is;
319   // if it's the sentinel, then we didn't find instances in p
320   // unique sequences.  If it's not the sentinel, then we've met
321   // the -p criterion.
322   if (seqNums[p - 1] == -1)
323     {
324       free (seqNums);
325       return (0);
326     }
327   else
328     {
329       free (seqNums);
330       return (1);
331     }
332 }
```

C.4.0.48 cll_t* completeConv (cll_t ** *head*, int *support*, int *numOffsets*, int *minLength*, int * *indexToSeq*, int *p*)

Performs complete convolution given the starting list of cliques. Input: a pointer to the head of the initial clique linked list, the minimum support criterion value, the number of offsets in the sequence set, the minimum length of motifs (which is the length of motifs in the initial clique linked list), the index/Sequence data structure, and the value of the -p flag to prune based on unique sequence occurrences. Output: a linked list of all maximal cliques based on the initial clique linked list.

Definition at line 1417 of file convll.c.

References emptyMemberStacks(), fillMemberStacks(), popAllCll(), pruneCll(), and

wholeRoundConv().

Referenced by convolve().

```
1419 {
1420   int i = 0;
1421   mll_t **memList = NULL;
1422   cll_t *nextPhase = NULL;
1423   cll_t *allCliques = NULL;
1424   int length = minLength;
1425   memList = (mll_t **) malloc (numOffsets * sizeof (mll_t *));
1426   if (memList == NULL)
1427     {
1428       fprintf (stderr, "Memory error - completeConv\n%s\n", strerror (errno));
1429       fflush (stderr);
1430       exit (0);
1431     }
1432   // The number of offsets will never change, so this can be defined
1433   // now, though we will have to change what is in these arrays later.
1434   for (i = 0; i < numOffsets; i++)
1435     {
1436       memList[i] = NULL;
1437     }
1438
1439   // NOTE: This assumes that the elemPats all meet the support criterion
1440
1441   // So we'll do this as long as the head is non-null.. that means that
1442   // the initial set of cliques must be non-null.  Those are then
1443   // convolved and the linked list for the next round is set to head,
1444   // so this continues until the linked list for the "next round" at
1445   // the end of some round is null.
1446   while (*head != NULL)
1447     {
1448       // First we get the inverse information for this round: find
1449       // out which cliques each offset is a member of.
1450       memList = fillMemberStacks (*head, memList);
1451       // printf("numOffsets.bak = %d\n",numOffsets);
1452       // // Then we convolve a whole round.
1453       nextPhase =
1454     wholeRoundConv (head, memList, numOffsets, support, length,
1455             &allCliques);
1456       // Do some housekeeping.
1457       memList = emptyMemberStacks (memList, numOffsets);
1458       popAllCll (*head);
1459       // Enforce the -p flag for subsequent rounds.
1460       if (p > 1)
1461     {
1462       nextPhase = pruneCll (nextPhase, indexToSeq, p);
1463     }
1464       // And move on to the next round of convolution.
1465       *head = nextPhase;
1466       length++;
1467     }
1468
1469   free (memList);
1470
1471   return allCliques;
1472 }
```

C.4.0.49 mll_t** emptyMemberStacks (mll_t ** *memberStacks*, int *size*)

After we have performed a round of convolution, this "empties" the member stacks by popping all nodes off each member linked list. Input: array of member linked

lists, the size of that array (total number of offsets). Output: the array of now-empty member linked lists.

Definition at line 538 of file convll.c.

References popWholeMemStack().

Referenced by completeConv().

```
539 {
540   int i = 0;
541
542   for (i = 0; i < size; i++)
543     {
544       memberStacks[i] = popWholeMemStack (memberStacks[i]);
545     }
546
547   return memberStacks;
548 }
```

C.4.0.50 mll_t** fillMemberStacks (cll_t * *head*, mll_t ** *memberStacks*)

Fills the entire memberStacks data structure by calling addToStacks for each clique in the clique linked list. Input: head of a clique linked list, array of member linked lists. Output: the array of updated member linked lists.

Definition at line 517 of file convll.c.

References addToStacks(), and cnode::next.

Referenced by completeConv().

```
518 {
519   cll_t *curr = head;
520   // Just go down the linked list calling addToStacks
521   while (curr != NULL)
522     {
523       memberStacks = addToStacks (curr, memberStacks);
524       curr = curr->next;
525     }
526
527   return memberStacks;
528 }
```

C.4.0.51 cll_t* initheadCll (cll_t * *head*, cSet_t * *newset*)

Initializes the empty head of a linked list by adding a set to that head. Note: this is only called immediately after pushing onto a cll, because the push always creates a new empty head. This function should not be called by the user; see pushcSet. Input: head of a linked list, pointer to a cSet_t list of clique members. Output: head of a linked list.

Definition at line 172 of file convll.c.

References cnode::set.

Referenced by pushcSet().

```
173 {
174   // Check to make sure that the head is not already initialized.
175   if (head->set != NULL)
176     {
177       printf ("Stack head already initialized!");
178       exit (0);
179     }
180   // Make the head's set pointer point to the new set.
181   head->set = newset;
182   return head;
183 }
```

C.4.0.52 mll_t* mergeIntersect (cll_t * *first*, cll_t * *second*, mll_t * *intersection*, bitSet_t * *printstatus*, int * *newSupport*)

Convolves two cliques in a non-commutative manner. It finds which members of the first clique are immediately followed by a member in the second clique. Input: pointer to the location in the linked list of the first clique to be convolved, pointer to the location in the linked list of the second clique to be convolved, a member linked list used to store the intersection of the two cliques, the printstatus bitSet, and a pointer to an integer with the support of the clique formed by convolution. Output: a member linked list with the intersection of the two cliques, plus the side effect of that intersection's cardinality being stored in the integer pointed to by newSupport.

Definition at line 759 of file convll.c.

References cSet_t::members, pushMemStack(), and cnode::set.

Referenced by singleCliqueConv().

```
761 {
762
763   int i = 0, j = 0, status = 0;
764
765   // Make sure we are still in-bounds, otherwise we bail out
766   // We'll refer to the offset currently being analyzed from the
767   // first clique as the 'first offset' and the offset currently
768   // being analyzed from the second clique as the 'second offset'
769   while ((i < first->set->size) && (j < second->set->size))
770     {
771       // If the second offset is earlier than the first offset plus
772       // one, then we move on to the next possible second offset
773       if ((first->set->members[i] + 1) > second->set->members[j])
774     {
775       j++;
776     }
777       // If the second offset is later than the first offset plus
778       // one, then we move on the next possible first offset
779       else if ((first->set->members[i] + 1) < second->set->members[j])
780     {
781       i++;
782     }
783       // Otherwise, the second offset is equal to the first offset
784       // plus one, so we have an extendable node.  Push that on
785       // to the intersection stack, move both the first and second
786       // offsets to their respective next possible offsets, and
787       // increment the support counter for the new clique (status)
788       else
789     {
790       intersection = pushMemStack (intersection, first->set->members[i]);
```

```
791       i++;
792       j++;
793       status++;
794     }
795     }
796
797   // Send the value of the clique's new support out of this function
798   *newSupport = status;
799   return intersection;
800 }
```

C.4.0.53 cSet_t* mllToCSet (mll_t * *clique*)

Turns a member linked list used to store the intersection of two cliques into something more useful: a cSet_t structure. Input: a clique in mll_t form. Output: a clique in cSet_t form.

Definition at line 1145 of file convll.c.

References mnode::cliqueMembership, cSet_t::members, mnode::next, and cSet_t::size.

Referenced by pushConvClique().

```
1146 {
1147   int sizecount = 0, i = 0;
1148   cSet_t *cliqueCset = malloc (sizeof (cSet_t));
1149   mll_t *head = clique;
1150   if (cliqueCset == NULL)
1151     {
1152       fprintf (stderr, "Memory error - mllToCSet cSet\n%s\n",
1153            strerror (errno));
1154       fflush (stderr);
1155       exit (0);
1156     }
1157   // First count up how many members there are in the member linked list
1158   while (head != NULL)
1159     {
1160       sizecount++;
1161       head = head->next;
1162     }
1163
1164   head = clique;
1165   cliqueCset->size = sizecount;
1166   cliqueCset->members = (int *) malloc (sizecount * sizeof (int));
1167
1168   if (cliqueCset->members == NULL)
1169     {
1170       fprintf (stderr, "Memory error - mllTlCSet cliquemembers\n%s\n",
1171            strerror (errno));
1172       fflush (stderr);
1173       exit (0);
1174     }
1175   // In order to stay in the same format as with bitSet translation to
1176   // cSet, we ensure that the ids of the members are ascending with
1177   // ascending index number in the cSet.  This is accomplished by noting
1178   // that since the intersection members are pushed onto the stack,
1179   // a LIFO operation, that the first intersected nodes off the stack
1180   // will have the highest ids, so we will put them at the end of
1181   // the members array with the higher index values.
1182   for (i = sizecount - 1; i >= 0; i--)
1183     {
```

```
1184       cliqueCset->members[i] = head->cliqueMembership;
1185       head = head->next;
1186     }
1187
1188   return cliqueCset;
1189 }
```

C.4.0.54 cll_t* popAllCll (cll_t * *head*)

Shortcut function to pop all of the members of a linked list. Input: head of a linked list. Output: head of a now-empty linked list.

Definition at line 109 of file convll.c.

References popCll().

Referenced by completeConv(), and main().

```
110 {
111   while (head != NULL)
112     {
113       head = popCll (head);
114     }
115   return head;
116 }
```

C.4.0.55 cll_t* popCll (cll_t * *head*)

Removes the head of the clique linked list, returns the new head of the clique linked list, and frees the memory occupied by the old head. Input: head of a linked list. Output: head of a linked list.

Definition at line 66 of file convll.c.

References cSet_t::members, cnode::next, and cnode::set.

Referenced by popAllCll().

```
67 {
68   // by default the new head is NULL...is important later
69   cll_t *newHead = NULL;
70   if (head == NULL)
71     {
72       fprintf (stderr, "\nCan't pop a null linked list\n");
73       fflush (stderr);
74       exit (0);
75     }
76   // unless this is the end of the linked list, set the new head
77   // to the next member of the list.  Otherwise, since by default the
78   // new head is NULL, it will properly return an empty list
79   if (head->next != NULL)
80     {
81       newHead = head->next;
82     }
83   // Check to see if there is a set.  If there is, and there are members,
84   // then first free the members.  And if there is a set, then free it.
85   if (head->set != NULL)
86     {
```

```
87       if (head->set->members != NULL)
88     {
89       free (head->set->members);
90       head->set->members = NULL;
91     }
92       free (head->set);
93       head->set = NULL;
94     }
95   // Both the members and set have been freed, so now can free the cll_t
96   // without leaking anything.
97
98   free (head);
99   head = NULL;
100   return newHead;
101 }
```

C.4.0.56 mll_t* popMemStack (mll_t * *head*)

Pops the head off of a single member linked list. Input: head of a member linked list. Output: the new head of a member linked list after popping one item.

Definition at line 440 of file convll.c.

References mnode::next.

Referenced by popWholeMemStack().

```
441 {
442   // by default the new head is NULL...is important later
443   mll_t *newHead = NULL;
444   if (head == NULL)
445     {
446       fprintf (stderr, "\nCan't pop a null linked list - popMemStack\n");
447       fflush (stderr);
448       exit (0);
449     }
450   if (head->next != NULL)
451     {
452       newHead = head->next;
453     }
454   free (head);
455   head = NULL;
456   return newHead;
457 }
```

C.4.0.57 mll_t* popWholeMemStack (mll_t * *head*)

Pops all items off of a member linked list. Input: head of a member linked list. Output: empty head of a member linked list.

Definition at line 465 of file convll.c.

References popMemStack().

Referenced by emptyMemberStacks(), and singleCliqueConv().

```
466 {
467   while (head != NULL)
468     {
```

```
469       head = popMemStack (head);
470     }
471   return head;
472 }
```

C.4.0.58 int printCll (cll_t * *head*)

Prints the members (cliques) of a linked list in the format: *id* = unique id number of clique within linked list; *Length* = number of members of clique, if available; *Size* = length of each member of clique; *Members* = newline-separated list of members of the clique. Input: head of a linked list. Output: Gives text output, returns (meaningless) exit value.

Definition at line 128 of file convll.c.

References cnode::id, cnode::length, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

```
129 {
130   int i = 0;
131   cll_t *curr = head;
132   while (curr != NULL)
133     {
134       printf ("id = %d\n", curr->id);
135       // Make sure the clique is nonzero in size before attempting
136       // to print it
137       if ((curr->set != NULL) && (curr->set->size > 0))
138     {
139       if (curr->length >= 0)
140         {
141           printf ("Length = %d\n", curr->length);
142         }
143       printf ("Size = %d\n", curr->set->size);
144       printf ("Members = \n");
145       for (i = 0; i < curr->set->size; i++)
146         {
147           printf ("\t%d\n", curr->set->members[i]);
148         }
149       printf ("*********************************************\n");
150     }
151       else
152     {
153       fprintf (stderr, "\nClique has no members! -- printCll\n");
154       fflush (stderr);
155       exit (0);
156     }
157       curr = curr->next;
158     }
159   return EXIT_SUCCESS;
160 }
```

C.4.0.59 int printCllPattern (cll_t * *node*, int *length*)

Prints out the contents of a clique linked list node in this format: *support* = number of motif occurrences (*id* = some id number); *members* = newline-separated list of offsets. Input: a specific node to be output, the length of the motif inside it. Output: text per above, and an integer success value.

Definition at line 1482 of file convll.c.

References cnode::id, cSet_t::members, cnode::set, and cSet_t::size.

```
1483 {
1484   int i = 0;
1485
1486   printf ("\nSupport = %d\t(id = %d)\n", node->set->size, node->id);
1487   printf ("Members = \n");
1488   for (i = 0; i < node->set->size; i++)
1489     {
1490       printf ("\t%d\n", node->set->members[i]);
1491     }
1492   return 1;
1493 }
```

C.4.0.60 int printCSet (cSet_t * *node*)

Prints out the contents of a cSet_t in the following format: *support* = number of nodes in clique; *members* = newline-separated list of nodes in clique. Input: a clique in the form of a cSet_t object. Output: in text, the contents of the cSet_t object. An integer is returned as well, with 1 indicating success.

Definition at line 1068 of file convll.c.

References cSet_t::members, and cSet_t::size.

```
1069 {
1070   int i = 0;
1071   if (node->size == 0)
1072     {
1073       fprintf (stderr, "cSet has no members! - printCSet\n");
1074       fflush (stderr);
1075       exit (0);
1076     }
1077   else
1078     {
1079       printf ("\nSupport = %d\n", node->size);
1080       printf ("Members = \n");
1081       for (i = 0; i < node->size; i++)
1082     {
1083       printf ("\t%d\n", node->members[i]);
1084     }
1085       return 1;
1086     }
1087 }
```

C.4.0.61 void printMemberStacks (mll_t ** *memberStacks*, int *size*)

Prints the contents of the member stacks. Input: array of member linked lists, size of that array (total number of offsets). Output: only text output/no return value.

Definition at line 557 of file convll.c.

References mnode::cliqueMembership, and mnode::next.

```
558 {
```

```
559   int i = 0;
560   mll_t *curr = NULL;
561
562   for (i = 0; i < size; i++)
563     {
564       curr = memberStacks[i];
565       printf ("Offset %d: ", i);
566       while (curr != NULL)
567     {
568       printf ("%d,", curr->cliqueMembership);
569       curr = curr->next;
570     }
571       printf ("\n");
572     }
573 }
```

C.4.0.62 cll_t* pruneCll (cll_t * *head*, int * *indexToSeq*, int *p*)

Prunes a motif linked list of all motifs without support in at least

unique source sequences. Input: head of a motif linked list, pointer to a structure that dereferences offset indices to sequence numbers, minimum number of unique source sequences in which a motif must occur. Output: head of a (potentially altered) motif linked list.

Definition at line 514 of file newConv.c.

References cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by completeConv(), and convolve().

```
515 {
516   int i = 0, j = 0, thisSeq = 0;
517   int *seqNums = NULL;
518   cll_t * curr = head;
519   cll_t * prev = NULL;
520   cll_t * storage = NULL;
521
522     // We'll do this similar to the pruneBitGraph function... we will
523     // keep track of which source sequence each motif occurrence was in.
524     // Again, since the occurrences are listed monotonically, we only
525     // need to compare the last non-sentinel index to the current
526     // sequence number.
527     seqNums = (int *) malloc (p * sizeof (int));
528   if (seqNums == NULL)
529     {
530       fprintf (stderr, "Memory error - pruneCll\n%s\n", strerror (errno));
531       fflush (stderr);
532       exit (0);
533     }
534   while (curr != NULL)
535     {
536
537     // First make sure the set size is at least p.
538     // This is redundant, but extremely simple and not expensive,
539     // so we'll leave it in just as a check.
540     if (curr->set->size < p)
541     {
542       if (prev != NULL)
543         {
544           prev->next = curr->next;
545         }
```

```
546       else
547         {
548           head = curr->next;
549         }
550       storage = curr->next;
551       free (curr->set->members);
552       free (curr->set);
553       free (curr);
554       curr = storage;
555       continue;
556     }
557       for (i = 0; i < p; i++)
558     {
559       seqNums[i] = -1;
560     }
561       j = 0;
562       seqNums[0] = indexToSeq[curr->set->members[0]];
563
564     // Note, we've checked to make sure size > p, and we know
565     // p must be 2 or greater, so we can start at 1 without
566     // worrying about segfaulting
567     for (i = 1; i < curr->set->size; i++)
568     {
569       thisSeq = indexToSeq[curr->set->members[i]];
570       if (thisSeq != seqNums[j])
571         {
572           j++;
573           seqNums[j] = thisSeq;
574           if (j == p - 1)
575         {
576           break;
577         }
578         }
579     }
580
581     // Same story as before... if the last number is -1,
582     // then we didn't have enough to fill up the <p> different
583     // slots, so this doesn't meet our criterion.
584     if (seqNums[p - 1] == -1)
585     {
586       if (prev != NULL)
587         {
588           prev->next = curr->next;
589         }
590       else
591         {
592           head = curr->next;
593         }
594       storage = curr->next;
595       free (curr->set->members);
596       free (curr->set);
597       free (curr);
598       curr = storage;
599     }
600       else
601     {
602       prev = curr;
603       curr = curr->next;
604     }
605     }
606   free (seqNums);
607   return (head);
608 }
```

C.4.0.63 cll_t* pushClique (bitSet_t * *clique*, cll_t * *head*, int * *indexToSeq*, int *p*)

Pushes a bitSet onto a clique linked list, performing all necessary manipulations in order to do so. Input: new clique in the form of a bitSet_t, head of a linked list, pointer to the index/sequence number data structure, integer value of the -p flag. Output: head of an updated clique linked list.

Definition at line 345 of file convll.c.

References bitSetToCSet(), checkCliquecSet(), cliquecounter, and pushcSet().

Referenced by findCliques(), and singleLinkage().

```
346 {
347   cSet_t *cliquecSet = NULL;
348
349   // Change the bitSet_t to a cSet_t
350   cliquecSet = bitSetToCSet (clique);
351   // If the -p flag has been assigned a value, then check the clique
352   // and only proceed if that criterion is met.  Otherwise, free the
353   // memory that we had allocated up to this point.
354   if (p > 1)
355     {
356       if (checkCliquecSet (cliquecSet, indexToSeq, p))
357     {
358       cliquecounter++;
359       /*
360          printf("%d\n",cliquecounter);
361        */
362       /*
363          fflush(stdout);
364        */
365       head = pushcSet (head, cliquecSet);
366     }
367       else
368     {
369       free (cliquecSet->members);
370       free (cliquecSet);
371     }
372       // If the -p flag wasn't set, then just push the cSet onto the linked
373       // list.
374     }
375   else
376     {
377       cliquecounter++;
378       /*
379          printf("%d\n",cliquecounter);
380        */
381       /*
382          fflush(stdout);
383        */
384       head = pushcSet (head, cliquecSet);
385     }
386   return head;
387 }
```

C.4.0.64 cll_t* pushCll (cll_t * *head*)

Pushes a new, empty head onto a linked list of cliques. Note: this should always be followed by a call to initheadCll, as the head pushed on here is empty and will be

meaningless without any members. This function should NOT be used by the user; see pushcSet. Input: head of a linked list. Output: head of a linked list.

Definition at line 28 of file convll.c.

References cnode::id, cnode::length, cnode::next, cnode::set, and cnode::stat.

Referenced by pushcSet().

```
29 {
30   // Make a pointer, verify memory
31   cll_t *a = NULL;
32   a = (cll_t *) malloc (sizeof (cll_t));
33   if (a == NULL)
34     {
35       fprintf (stderr, "\nMemory Error - pushCll\n%s\n", strerror (errno));
36       fflush (stderr);
37       exit (0);
38     }
39   // Initialize id (sequential) and pointer to next item, but not
40   // the cSet with the clique members
41   if (head == NULL)
42     {
43       a->id = 0;
44       a->next = NULL;
45     }
46   else
47     {
48       a->next = head;
49       a->id = head->id + 1;
50     }
51   a->set = NULL;
52   a->length = -1;
53   a->stat = -1;
54   return a;
55 }
```

C.4.0.65 cll_t* pushConvClique (mll_t * *clique*, cll_t * *head*)

Pushes a freshly-convolved clique, currently in mll_t form, onto the clique linked list for the next level. Also checks to make sure that the convolved clique is unique, and if it isn't, it takes appropriate action. Input: a convolved clique in mll_t form, the head of a clique linked list for the next level. Output: (potentially new) head of the clique linked list for the next level.

Definition at line 1099 of file convll.c.

References cSet_t::members, mllToCSet(), pushcSet(), removeSupers(), swapNodecSet(), and uniqClique().

Referenced by singleCliqueConv().

```
1100 {
1101   int status = 0;
1102   cSet_t *cliquecSet = NULL;
1103
1104   // First change the clique to something we can used more easily
1105   cliquecSet = mllToCSet (clique);
1106   // Then check to make sure it's unique by finding out its status
1107   status = uniqClique (cliquecSet, head);
```

```
1108
1109   // printf("Candidate:\n");
1110   // printCSet(cliquecSet);
1111
1112   // If we get -2, then this clique is a subset, so just free
1113   // the cSet we just made and move on.
1114   if (status == -2)
1115     {
1116       free (cliquecSet->members);
1117       free (cliquecSet);
1118       cliquecSet = NULL;
1119     }
1120   // If we get -1, then this is a unique clique, so push it on.
1121   else if (status == -1)
1122     {
1123       head = pushcSet (head, cliquecSet);
1124     }
1125   // Otherwise, this clique is a superset, so we'll first remove
1126   // all of the other cliques of which this is a superset.  Then
1127   // we'll swap out the first clique of which this is a superset
1128   // with this current clique.  The clique being removed is free'd
1129   // within the swapNode function.
1130   else
1131     {
1132       head = removeSupers (head, status, cliquecSet);
1133       head = swapNodecSet (head, status, cliquecSet);
1134     }
1135   return head;
1136 }
```

C.4.0.66 cll_t* pushcSet (cll_t * *head*, cSet_t * *newset*)

Function that pushes the contents of a cSet (set of members of a clique) onto a linked list of cliques. Input: head of a linked list, new clique in the form of a cSet_t. Output: head of a linked list.

Definition at line 192 of file convll.c.

References initheadCll(), and pushCll().

Referenced by pushClique(), and pushConvClique().

```
193 {
194   head = pushCll (head);
195   head = initheadCll (head, newset);
196   return head;
197 }
```

C.4.0.67 mll_t* pushMemStack (mll_t * *head*, int *cliqueNum*)

This begins code for the member linked lists. A single one of these linked lists functions somewhat similarly to the clique linked lists, though with less information stored. Functionally, an array of member linked lists is used to access the "inverse" of what is contained in the clique linked lists. That is, we would like to be able to look up the cliques that a given node is a member of, so we have an array of member linked lists of size equal to the number of nodes.

This function pushes a single clique membership onto a node's member stack. Input: the head of a single member linked list, a clique number to be added. Output: the head of a single member linked list.

Definition at line 404 of file convll.c.

References mnode::cliqueMembership, and mnode::next.

Referenced by addToStacks(), and mergeIntersect().

```
405 {
406   mll_t *a = NULL;
407   a = (mll_t *) malloc (sizeof (mll_t));
408   // Memory error checking
409   if (a == NULL)
410     {
411       fprintf (stderr, "\nMemory Error - pushMemStack: %s\n",
412            strerror (errno));
413       fflush (stderr);
414       exit (0);
415     }
416   if (head == NULL)
417     {
418       a->next = NULL;
419     }
420   else
421     {
422       a->next = head;
423     }
424   // Store the number of the clique of which the node is a member.
425   // Note that we assume no duplication, which is guaranteed
426   // by our method of filling the member stacks, which is quite simple:
427   // go through all members of a clique (which have no duplicates
428   // because they are constructed from merge-intersections or from
429   // bitSet_t's) and add that clique to each node's membership list.
430   a->cliqueMembership = cliqueNum;
431   return a;
432 }
```

C.4.0.68 cll_t* removeSupers (cll_t * *head*, int *node*, cSet_t * *newClique*)

This function finds all cliques in a linked list of which the proposed clique is a superset. It starts looking AFTER the first clique which has already been found to be a subset. In some senses, it is just a continuation of the uniqclique function in order to take advantage of the fact that though a proposed clique can only be a subset of one existing next-level clique, it can be a superset of many existing next- level cliques. Input: head of a clique linked list, the id of the first node found to be a subset of the proposed clique, and the proposed clique (in cSet_t form). Output: the head of the clique linked list with all but the first subset (which was passed as an argument) removed. This function is now ready for swapNode to be called.

Definition at line 952 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by pushConvClique().

```
953 {
```

```
954   int foundStatus = 0;
955   cll_t *curr = head;
956   cll_t *prev = NULL;
957   int i = 0, j = 0, breakFlag = 0;
958
959   while (curr != NULL)
960     {
961       if (curr->id == node)
962     {
963       foundStatus = 1;
964       break;
965     }
966       curr = curr->next;
967     }
968
969   if (foundStatus == 0)
970     {
971       fprintf (stderr, "\nFirst clique not found! (removeSupers)\n");
972       fflush (stderr);
973       exit (0);
974     }
975   // Now this is trickier, to remove nodes from the middle of a linked
976   // list; this means that we need to remember which node we were just
977   // at so that we can connect it to the node after the one we are
978   // about to delete.
979   prev = curr;
980   curr = curr->next;
981
982   // This code is similar to that in uniqClique.
983   // Descend through all members of the next level's linked list.
984   while (curr != NULL)
985     {
986       i = 0;
987       j = 0;
988       breakFlag = 0;
989       // The proposed convolved clique will be referred to as the
990       // 'first' clique, and the current clique being analyzed
991       // in the next level is the 'second' clique.
992       // Continue if we have more members in both cliques.  We will
993       // have already broken out if it is not possible for this
994       // second clique to be a subset of the first.
995       while ((i < newClique->size) && (j < curr->set->size))
996     {
997       // If the current member of the first clique is
998       // less than the current member of the second clique
999       // then it is still possible that the first is a
1000       // superset of the second, so move on to the next
1001       // member.
1002       if (newClique->members[i] < curr->set->members[j])
1003         {
1004           i++;
1005         }
1006       // If the current member of the first clique is greater
1007       // than the current member of the second clique, then
1008       // the proposed second clique cannot be a subset since
1009       // its members are all in ascending order.  We also
1010       // know that since the first clique already has
1011       // a subset in this linked list, the current node
1012       // cannot possibly be a superset of the proposed
1013       // clique, so we can just disregard that.  Thus,
1014       // we make a flag signifying this and break out.
1015       else if (newClique->members[i] > curr->set->members[j])
1016         {
1017           breakFlag = 1;
1018           break;
1019         }
1020       else
1021         {
```

```
1022          i++;
1023          j++;
1024        }
1025    }
1026      // If the breakflag is 1, then we know
1027      // that there is a member of the second clique not in the
1028      // first, and so the second is not a subset.  If the breakflag
1029      // is 0 but j is less than the second clique's size, then
1030      // we must have broken because we ran out of members in the
1031      // first clique... thus, there is a member of the second
1032      // clique not in the first.  Thus, only if the breakflag is
1033      // 0 and j is equal to the size of the second clique do we
1034      // know that every member of the second clique is in the first
1035      // and that the second clique can thus be removed.
1036      if ((breakFlag == 0) && (j == curr->set->size))
1037    {
1038      // Make the previous clique point to the next one
1039      // instead of the current one.
1040      prev->next = curr->next;
1041      // Free all of the memory used by the current clique.
1042      free (curr->set->members);
1043      free (curr->set);
1044      free (curr);
1045      curr = prev->next;
1046    }
1047      else
1048    {
1049      // Otherwise, the current second clique is not a
1050      // subset of the first, and we advance the prev and
1051      // curr pointers.
1052      prev = curr;
1053      curr = curr->next;
1054    }
1055    }
1056  return head;
1057 }
```

C.4.0.69 bitSet_t* searchMemsWithList (int * *list*, int *listsize*, mll_t ** *memList*, int *numOffsets*, bitSet_t * *queue*)

Creates one large queue by calling "setStackTrue" for each member of a list of offsets. This then creates the union of clique membership for all offsets in the list being searched. Input: an array of offset numbers, the length of that array, an array of member linked lists, the length of that array (the total number of offsets), and a bitSet_t to store the union/queue. Output: the union/queue in a bitSet_t structure.

Definition at line 611 of file convll.c.

References emptySet(), and setStackTrue().

Referenced by wholeCliqueConv().

```
613 {
614   int i = 0;
615   emptySet (queue);
616
617   // Go through each offset in the list
618   for (i = 0; i < listsize; i++)
619     {
620       // Check to make sure that's a valid offset number, and if so
621       // then set its stack true in the queue.
```

```
622       if (list[i] + 1 < numOffsets)
623     {
624       queue = setStackTrue (memList, list[i] + 1, queue);
625     }
626       else
627     {
628       fprintf (stderr, "\nInvalid offset number! - searchMemsWithList\n");
629       fprintf (stderr, "\nlist[i]+1 (%d) >= numOffsets (%d)\n",
630            list[i] + 1, numOffsets);
631       fflush (stderr);
632       exit (0);
633     }
634     }
635
636   return queue;
637 }
```

C.4.0.70 bitSet_t* setStackTrue (mll_t ** *memList*, int *i*, bitSet_t * *queue*)

Adds all of the members of a given stack to a "queue" in the form of a bitSet_t data structure. That is, for each clique in the member linked list, it sets the corresponding bit in the bitSet_t true. Input: array of member linked lists, an integer indicating a specific member linked list, and a bitSet_t of length >= the number of cliques in the current clique linked list. Ouput: the updated bitSet_t object.

Definition at line 585 of file convll.c.

References mnode::cliqueMembership, mnode::next, and setTrue().

Referenced by searchMemsWithList().

```
586 {
587   mll_t *curr = memList[i];
588
589   // Traverse down the member linked list
590   while (curr != NULL)
591     {
592       // Set the bit in queue corresponding to the current clique
593       // membership true
594       setTrue (queue, curr->cliqueMembership);
595       curr = curr->next;
596     }
597
598   return queue;
599 }
```

C.4.0.71 cll_t* singleCliqueConv (cll_t * *head*, int *firstClique*, cll_t ** *firstGuess*, int *secondClique*, cll_t ** *secondGuess*, cll_t * *nextPhase*, bitSet_t * *printStatus*, int *support*)

Convolves one single clique against one other single clique. Note that this is non-commutative, so exchanging firstClique and secondClique will not give the same results. The "guess" pointers keep the location of the previous clique in the linked list so that we don't have to search the linked list from the beginning/end every time. We

exploit our earlier tidiness in that we can reasonably guess that we will monotonically traverse down cliques. Input: head of the current clique linked list, the id number of the first clique, a pointer to a guess at the first clique, the id number of the second clique, a pointer to a guess at the second clique, the head of the clique linked list for the next round of convolution, a bitSet indicating which cliques should be output as maximal, and the minimum support flag. Output: the head of clique linked list for the next round of convolution (which may have changed if the two cliques could be convolved).

Definition at line 657 of file convll.c.

References cnode::id, mergeIntersect(), cnode::next, popWholeMemStack(), pushConvClique(), cnode::set, setFalse(), and cSet_t::size.

Referenced by wholeCliqueConv().

```
660 {
661   cll_t *first = NULL, *second = NULL;
662   mll_t *survivingMems = NULL;
663   // int flag = 0;
664   int newSupport = 0;
665   // cll_t *checker = head;
666
667   // Check to make sure we're looking for legitimate cliques.
668   if ((firstClique > head->id) || (secondClique > head->id))
669     {
670       fprintf (stderr, "\nNonexistent clique! - singleCliqueConv\n");
671       fflush (stderr);
672       exit (0);
673     }
674   // Our guesses depend on monotonic traversal.  If we don't find
675   // the first clique, then bail out.
676   while ((*firstGuess)->id != firstClique)
677     {
678       if ((*firstGuess)->next != NULL)
679     {
680       *firstGuess = (*firstGuess)->next;
681     }
682       else
683     {
684       fprintf (stderr, "\nFirst clique not found! - singleCliqueConv\n");
685       fflush (stderr);
686       exit (0);
687     }
688     }
689   first = *firstGuess;
690
691   // Our guesses depend on monotonic traversal.  If we don't find
692   // the second clique, then bail out.
693   while ((*secondGuess)->id != secondClique)
694     {
695       if ((*secondGuess)->next != NULL)
696     {
697       (*secondGuess) = (*secondGuess)->next;
698     }
699       else
700     {
701       fprintf (stderr, "\nSecond clique not found! - singleCliqueConv\n");
702       fflush (stderr);
703       exit (0);
704     }
705     }
706   second = *secondGuess;
```

```
707   // Find out what the surviving members are when the first clique
708   // is convolved with the second clique
709   survivingMems =
710     mergeIntersect (first, second, survivingMems, printStatus, &newSupport);
711
712   // If the first clique is subsumed by the second, then it is not
713   // maximal, so don't print it.
714   // printStatus true means print it!
715   if (newSupport == first->set->size)
716     {
717       setFalse (printStatus, first->id);
718     }
719   // If the second clique is subsumed by the first, then it is not
720   // maximal, so don't print it.
721   if (newSupport == second->set->size)
722     {
723       setFalse (printStatus, second->id);
724     }
725
726   // If the support of the clique just formed by convolution meets the
727   // support criterion, then push it on to the linked list for
728   // the next phase of convolution.
729   if (newSupport >= support)
730     {
731       // printf("Push %d and %d\n",first->id,second->id);
732       nextPhase = pushConvClique (survivingMems, nextPhase);
733       // printf("---------\n");
734       // printCll(nextPhase);
735       // printf("---------\n");
736     }
737   // Pop the surviving members; they are no longer needed, as they
738   // either didn't meet the support criterion or have been pushed on
739   // already
740   survivingMems = popWholeMemStack (survivingMems);
741
742   return nextPhase;
743 }
```

C.4.0.72 cll_t* swapNodecSet (cll_t * *head*, int *node*, cSet_t * *newClique*)

Swaps out a node in a linked list that has been found to be a subset of a node that is not yet in the list. Input: the head of a clique linked list, a specific node within that linked list that is to be removed, and the new clique that is the superset of the node to be removed (in cSet_t form). Output: the head of the altered clique linked list.

Definition at line 904 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, and cnode::set.

Referenced by pushConvClique().

```
905 {
906   int foundflag = 0;
907   cll_t *curr = head;
908
909   // First we find the node that needs to be swapped out
910   while (curr != NULL)
911     {
912       if (curr->id == node)
913     {
914       foundflag = 1;
915       break;
```

```
916     }
917       curr = curr->next;
918     }
919
920   // If we can't find it, then we get upset and exit.
921   if (foundflag == 0)
922     {
923       fprintf (stderr, "\nClique not found! (in swapNode)\n");
924       fflush (stderr);
925       exit (0);
926     }
927   // Then we free the useless clique's members and its set data structure
928   // before pointing its set to the new clique.
929   free (curr->set->members);
930   free (curr->set);
931   curr->set = newClique;
932   return head;
933
934 }
```

C.4.0.73 int uniqClique (cSet_t * *cliquecSet*, cll_t * *head*)

Before we push a convolved clique onto the stack for the next level, this function ensures that it is not subsumed by and does not subsume any other clique currently on that stack. Input: a candidate clique for the next level in cSet_t form, and the head of the clique linked list for the next level. Output: an integer indicating the status of the proposed clique with respect to the next level: -1 if the clique is unique, -2 if the clique is a subset/duplicate of an existing clique, or a clique id in the range [0,numcliques) representing the first clique of which the proposed one is a superset. Note that by executing this each time a clique is added to the next level, we ensure that if the new clique is not unique, it can only be a superset or a subset of some other clique; it cannot be both a strictly superset of one and a strictly subset of another. One of those other two cliques would have been identified in previous steps as being super- or sub-sets, so it is impossible for one clique now to be both a super and a subset.

Definition at line 821 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by pushConvClique().

```
822 {
823   int i = 0, j = 0;
824   int asubbflag = 1, bsubaflag = 1;
825
826   // Descend through all members of the next level's linked list
827   while (head != NULL)
828     {
829       asubbflag = 1;
830       bsubaflag = 1;
831       i = 0;
832       j = 0;
833       // The proposed convolved clique will be referred to as the
834       // "first" clique, and the current clique being analyzed
835       // in the next level is the "second" clique.
836       // Continue if we have more members in both cliques AND if it
837       // is still possible for one clique to be a subset of
```

```
838       // the other.
839       while ((i < cliquecSet->size) && (j < head->set->size) &&
840          ((asubbflag == 1) || (bsubaflag == 1)))
841     {
842       // If the current member of the first clique is less
843       // than the current member of the second clique,
844       // it is impossible for the first clique to be a
845       // subset of the second (since the members are
846       // traversed in ascending order.
847       if (cliquecSet->members[i] < head->set->members[j])
848         {
849           i++;
850           asubbflag = 0;
851         }
852       // Similarly, if the current member of the second
853       // clique is less than the current member of the
854       // second clique, the second can't be a subset
855       // of the first.
856       else if (cliquecSet->members[i] > head->set->members[j])
857         {
858           j++;
859           bsubaflag = 0;
860         }
861       // Otherwise, they matched this time, so move them
862       // both on.
863       else
864         {
865           i++;
866           j++;
867         }
868     }
869
870       // If the proposed clique is a subset of some other clique
871       // in the next level, then return -2, and it won't be added.
872       // (Note, this also is how exact duplicates are handled.)
873       if ((asubbflag == 1) && (i == cliquecSet->size))
874     {
875       return (-2);
876     }
877       // If the proposed clique is a superset of some other clique(s)
878       // in the next level, then return the id of the first clique
879       // of which it is a superset.
880       if ((bsubaflag == 1) && (j == head->set->size))
881     {
882       return (head->id);
883     }
884       // If the proposed clique has not been found to be a superset
885       // or a subset yet, then move on to the next clique in
886       // the next level.
887       head = head->next;
888     }
889   // If we've gotten here, we've checked all cliques in the previous
890   // level and haven't found the proposed clique to be a superset or
891   // a subset... if so, then we're all good, so return a -1.
892   return (-1);
893 }
```

C.4.0.74 cll_t* wholeCliqueConv (cll_t * *head*, cll_t * *node*, cll_t ** *firstGuess*, mll_t ** *memList*, int *numOffsets*, cll_t * *nextPhase*, bitSet_t * *printStatus*, int *support*)

Convolves one single clique against all possible cliques that could possibly be convolved. It does not attempt to convolve all other cliques, but prunes that set by first

looking at the offsets that are in the clique, then collecting all of the cliques who have members that are one greater than the offsets in this clique, and then convolving those cliques in a sort of "queue" using the bitSet_t data structure. Input: the head of the clique linked list for the current level, the current node being convolved against in the linked list, the location of the previous node in the form of a pointer to a "guess", an array of member linked lists, the length of that array, the head of the clique linked list for the next level, a bitSet_t for the printStatus of maximality, and the support criterion. Output: the head of the (possibly modified) clique linked list for the next level.

Definition at line 1208 of file convll.c.

References bitSetToCSet(), countSet(), deleteBitSet(), cnode::id, cSet_t::members, newBitSet(), searchMemsWithList(), cnode::set, singleCliqueConv(), and cSet_t::size.

Referenced by wholeRoundConv().

```
1211 {
1212   bitSet_t *queue = NULL;
1213   cSet_t *cliquesToSearch = NULL;
1214   int i = 0;
1215   cll_t **secondGuess = NULL;
1216
1217   // This bitSet will be used to create a "queue" of the different
1218   // cliques that must be convolved against the current primary clique.
1219   // A bitset is used to make it easy to deal with duplicates, where
1220   // multiple clique members' next offsets
1221   // are all members of some other specific clique.
1222   queue = newBitSet (head->id + 1);
1223   queue =
1224     searchMemsWithList (node->set->members, node->set->size, memList,
1225             numOffsets, queue);
1226   // We'll use this "secondGuess" to store where the previous clique
1227   // being convolved was... since we will progressing monotonically
1228   // in descending order, this will save us some time in traversing the
1229   // linked list looking for the clique that we want.
1230   secondGuess = (cll_t **) malloc (sizeof (cll_t *));
1231   if (secondGuess == NULL)
1232     {
1233       fprintf (stderr, "Memory error - wholeCliqueConv\n%s\n",
1234            strerror (errno));
1235       fflush (stderr);
1236       exit (0);
1237     }
1238   // If the offsets that we are looking for are in no other cliques,
1239   // we can just bail out now.
1240   if (countSet (queue) == 0)
1241     {
1242       deleteBitSet (queue);
1243       return nextPhase;
1244     }
1245   // Otherwise, we start our secondGuess at the head and get going.
1246   *secondGuess = head;
1247
1248   // We change the bitSet to something more useful.
1249   cliquesToSearch = bitSetToCSet (queue);
1250
1251   // Note that we start from the end of the cSet member list so that
1252   // we can convolve the highest-id cliques first, which are at the
1253   // beginning of our stack of cliques.
1254   for (i = cliquesToSearch->size - 1; i >= 0; i--)
1255     {
```

```
1256       nextPhase = singleCliqueConv (head, node->id, firstGuess,
1257                     cliquesToSearch->members[i], secondGuess,
1258                     nextPhase, printStatus, support);
1259     }
1260
1261   // And then we free everything that we created
1262   deleteBitSet (queue);
1263   free (cliquesToSearch->members);
1264   free (cliquesToSearch);
1265   free (secondGuess);
1266   return nextPhase;
1267 }
```

C.4.0.75 cll_t* wholeRoundConv (cll_t ** *head*, mll_t ** *memList*, int *numOffsets*, int *support*, int *length*, cll_t ** *allCliques*)

Performs convolution on all cliques in a linked list by repeatedly calling wholeCliqueConv. Input: pointer to the head of a clique linked list for the current level, array of member linked lists, length of that array, minimum support threshold, the current length of motifs, and a pointer to a linked list containing all cliques that will be printed out. Output: the head of the clique linked list for the next level of convolution.

Definition at line 1279 of file convll.c.

References checkBit(), deleteBitSet(), fillSet(), cnode::id, newBitSet(), cnode::next, wholeCliqueConv(), and yankCll().

Referenced by completeConv().

```
1281 {
1282   bitSet_t *printStatus = NULL;
1283   cll_t *curr = *head;
1284   cll_t *prev = NULL;
1285   cll_t *nextPhase = NULL;
1286   cll_t **firstGuess = NULL;
1287
1288   // Create a bitset to keep track of print status for this level.
1289   // It starts off all true, and gets changed to false if the patterns
1290   // are not maximal.
1291   printStatus = newBitSet ((*head)->id + 1);
1292   fillSet (printStatus);
1293   firstGuess = (cll_t **) malloc (sizeof (cll_t *));
1294   if (firstGuess == NULL)
1295     {
1296       fprintf (stderr, "Memory error - wholeRoundConv\n%s\n",
1297            strerror (errno));
1298       fflush (stderr);
1299       exit (0);
1300     }
1301   // Start off at the head.
1302   *firstGuess = *head;
1303   // Convolve a whole clique at a time, traversing the linked list.
1304   // Note that firstGuess gets altered within the function.
1305   while (curr != NULL)
1306     {
1307       nextPhase =
1308     wholeCliqueConv (*head, curr, firstGuess, memList, numOffsets,
1309              nextPhase, printStatus, support);
1310       curr = curr->next;
1311     }
1312
```

```
1313   // Now go back to the head for printing output
1314   curr = *head;
1315
1316   // printf("\n**************************************************\n");
1317   // printf("Length = %d", length);
1318   // printf("\n**************************************************\n");
1319
1320   // For each clique that is still 'true' in printStatus and is thus
1321   // maximal, perform some sort of output.  Yankcll will pull out the
1322   // clique and save it for printing at a later time.
1323   while (curr != NULL)
1324     {
1325       if (checkBit (printStatus, curr->id))
1326     {
1327       // This is the line that makes the allCliques output.
1328       // Can either printcll, or add to allCliques.
1329       // printCllPattern(curr, length);
1330       yankCll (head, prev, &curr, allCliques, length);
1331     }
1332       else
1333     {
1334       prev = curr;
1335       curr = curr->next;
1336     }
1337     }
1338
1339   // And clean up.
1340   deleteBitSet (printStatus);
1341   free (firstGuess);
1342   return nextPhase;
1343 }
```

C.4.0.76 int yankCll (cll_t ** *head*, cll_t * *prev*, cll_t ** *curr*, cll_t ** *allCliques*, int *length*)

Removes a clique from within a linked list in order to save it for later printing. This is done so that the cliques are not printed as they are convolved, but rather after all rounds of convolution are complete. Input: a pointer to the head of the current linked list, the clique prior to the one that is to be yanked (NULL if the clique to be yanked is the head), the clique that is to be yanked, a pointer to the head of the list with all cliques that are to be printed, and the length of the current motif. Output: Nothing is returned beyond a success integer, but it alters the current level cll_t, the value of curr, and the linked list of all cliques that are to be printed.

Definition at line 1359 of file convll.c.

References cnode::id, and cnode::next.

Referenced by convolve(), and wholeRoundConv().

```
1361 {
1362   if (*curr == NULL)
1363     {
1364       fprintf (stderr, "\nCan't yank from end of cll!\n");
1365       fflush (stderr);
1366       exit (0);
1367     }
1368   // If we're not on the head, change the previous node's "next".
1369   // If we are on the head, make the new head be our current node's "next".
1370   if (prev != NULL)
```

```
1371     {
1372       prev->next = (*curr)->next;
1373     }
1374   else
1375     {
1376       *head = (*curr)->next;
1377     }
1378
1379   // Change next in curr, then change id and length information in curr
1380   (*curr)->next = *allCliques;
1381
1382   if (*allCliques != NULL)
1383     {
1384       (*curr)->id = (*allCliques)->id + 1;
1385     }
1386   else
1387     {
1388       (*curr)->id = 0;
1389     }
1390
1391   (*curr)->length = length;
1392
1393   *allCliques = *curr;
1394
1395   if (prev != NULL)
1396     {
1397       *curr = prev->next;
1398     }
1399   else
1400     {
1401       *curr = *head;
1402     }
1403   return (1);
1404 }
```

Variable Documentation

C.4.0.77 int cliquecounter = 0

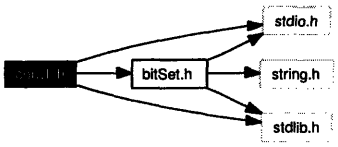
Definition at line 335 of file convll.c.

Referenced by pushClique().

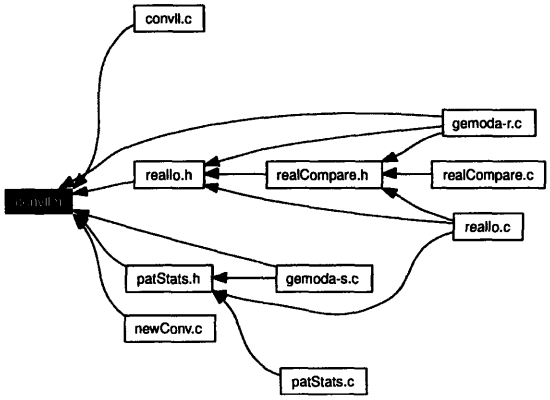
C.5 convll.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include "bitSet.h"
```

Include dependency graph for convll.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct cSet_t
- struct cnode
- struct mnode

Typedefs

- typedef cnode cll_t
- typedef mnode mll_t

Functions

- cll_t * pushCll (cll_t *head)

- cll_t * popCll (cll_t *head)
- cll_t * popAllCll (cll_t *head)
- int printCll (cll_t *head)
- cll_t * initheadCll (cll_t *head, cSet_t *newset)
- cll_t * pushcSet (cll_t *head, cSet_t *newset)
- cll_t * pushClique (bitSet_t *clique, cll_t *head, int *indexToSeq, int p)
- mll_t * pushMemStack (mll_t *head, int cliqueNum)
- mll_t * popMemStack (mll_t *head)
- mll_t * popWholeMemStack (mll_t *head)
- mll_t ** addToStacks (cll_t *node, mll_t **memberStacks)
- mll_t ** fillMemberStacks (cll_t *head, mll_t **memberStacks)
- mll_t ** emptyMemberStacks (mll_t **memberStacks, int size)
- void printMemberStacks (mll_t **memberStacks, int size)
- bitSet_t * searchMemsWithList (int *list, int listsize, mll_t **memList, int numOffsets, bitSet_t *queue)
- bitSet_t * setStackTrue (mll_t **memList, int i, bitSet_t *queue)
- cll_t * singleCliqueConv (cll_t *head, int firstClique, cll_t **firstGuess, int secondClique, cll_t **secondGuess, cll_t *nextPhase, bitSet_t *printStatus, int support)
- mll_t * mergeIntersect (cll_t *first, cll_t *second, mll_t *intersection, bitSet_t *printStatus, int *newSupport)
- cll_t * pushConvClique (mll_t *clique, cll_t *head)
- cSet_t * mllToCSet (mll_t *clique)
- cSet_t * bitSetToCSet (bitSet_t *clique)
- cll_t * wholeCliqueConv (cll_t *head, cll_t *node, cll_t **firstGuess, mll_t **memList, int numOffsets, cll_t *nextPhase, bitSet_t *printStatus, int support)
- cll_t * wholeRoundConv (cll_t **head, mll_t **memList, int numOffsets, int support, int length, cll_t **allCliques)
- cll_t * completeConv (cll_t **head, int support, int numOffsets, int minLength, int *indexToSeq, int p)
- int printCllPattern (cll_t *node, int length)
- int uniqClique (cSet_t *clique, cll_t *head)
- cll_t * swapNodecSet (cll_t *head, int node, cSet_t *newClique)
- int yankCll (cll_t **head, cll_t *prev, cll_t **curr, cll_t **allCliques, int length)
- cll_t * removeSupers (cll_t *head, int node, cSet_t *newClique)

Detailed Description

This header file contains declarations and definitions for dealing with different kinds of sets that are used throughout the convolution stage of Gemoda.

Definition in file convll.h.

Typedef Documentation

C.5.0.78 typedef struct cnode cll_t

This data structure is a linked list for storing cliques. Each member of the linked list has a set, an ID number, a length (which gives the number of characters in the motif), a pointer to the next member of the linked list, and a floating-point number for storing statistical information.

C.5.0.79 typedef struct mnode mll_t

This data structure is just a link to list of integers used for bookkeeping during the convolution stage.

Function Documentation

C.5.0.80 mll_t** addToStacks (cll_t * *node*, mll_t ** *memberStacks*)

For one clique, it adds membership for that clique to all of its members' member stacks. Input: a specific clique in a clique linked list, an array of member stacks. Output: the array of updated member stacks.

Definition at line 425 of file convll.c.

References cnode::id, cSet_t::members, pushMemStack(), and cnode::set.

Referenced by fillMemberStacks().

C.5.0.81 cSet_t* bitSetToCSet (bitSet_t * *clique*)

Converts a bitSet_t to a cSet_t for the purposes of pushing it onto a linked list of cliques. The bitSet_t data structure is used for massive comparisons during clique-finding but is unwieldy/inefficient when it is known that the structure is sparse. The cSet_t allows for efficient comparison of sparse bitSet_t's. Use this just before pushing a newly-discovered clique onto a clique linked list. Input: a new clique in the form of a bitSet_t. Output: the same clique in the form of a cSet_t.

Definition at line 193 of file convll.c.

References countSet(), cSet_t::members, nextBitBitSet(), and cSet_t::size.

Referenced by pushClique(), and wholeCliqueConv().

C.5.0.82 cll_t* completeConv (cll_t ** *head*, int *support*, int *numOffsets*, int *minLength*, int * *indexToSeq*, int *p*)

Performs complete convolution given the starting list of cliques. Input: a pointer to the head of the initial clique linked list, the minimum support criterion value, the number of offsets in the sequence set, the minimum length of motifs (which is the length of motifs in the initial clique linked list), the index/Sequence data structure, and the value of the -p flag to prune based on unique sequence occurrences. Output: a linked list of all maximal cliques based on the initial clique linked list.

Definition at line 1267 of file convll.c.

References emptyMemberStacks(), fillMemberStacks(), popAllCll(), pruneCll(), and wholeRoundConv().

Referenced by convolve().

C.5.0.83 mll_t** emptyMemberStacks (mll_t ** *memberStacks*, int *size*)

After we have performed a round of convolution, this "empties" the member stacks by popping all nodes off each member linked list. Input: array of member linked lists, the size of that array (total number of offsets). Output: the array of now-empty member linked lists.

Definition at line 474 of file convll.c.

References popWholeMemStack().

Referenced by completeConv().

C.5.0.84 mll_t** fillMemberStacks (cll_t * *head*, mll_t ** *memberStacks*)

Fills the entire memberStacks data structure by calling addToStacks for each clique in the clique linked list. Input: head of a clique linked list, array of member linked lists. Output: the array of updated member linked lists.

Definition at line 455 of file convll.c.

References addToStacks(), and cnode::next.

Referenced by completeConv().

C.5.0.85 cll_t* initheadCll (cll_t * *head*, cSet_t * *newset*)

Initializes the empty head of a linked list by adding a set to that head. Note: this is only called immediately after pushing onto a cll, because the push always creates a new empty head. This function should not be called by the user; see pushcSet. Input: head of a linked list, pointer to a cSet_t list of clique members. Output: head of a linked list.

Definition at line 156 of file convll.c.

References cnode::set.

Referenced by pushcSet().

C.5.0.86 mll_t* mergeIntersect (cll_t * *first*, cll_t * *second*, mll_t * *intersection*, bitSet_t * *printstatus*, int * *newSupport*)

Convolves two cliques in a non-commutative manner. It finds which members of the first clique are immediately followed by a member in the second clique. Input: pointer to the location in the linked list of the first clique to be convolved, pointer to the location in the linked list of the second clique to be convolved, a member linked list used to store the intersection of the two cliques, the printstatus bitSet, and a pointer to an integer with the support of the clique formed by convolution. Output: a member linked list with the intersection of the two cliques, plus the side effect of that intersection's cardinality being stored in the integer pointed to by newSupport.

Definition at line 671 of file convll.c.

References cSet_t::members, pushMemStack(), cnode::set, and cSet_t::size.

Referenced by singleCliqueConv().

C.5.0.87 cSet_t* mllToCSet (mll_t * *clique*)

Turns a member linked list used to store the intersection of two cliques into something more useful: a cSet_t structure. Input: a clique in mll_t form. Output: a clique in cSet_t form.

Definition at line 1022 of file convll.c.

References mnode::cliqueMembership, cSet_t::members, mnode::next, and cSet_t::size.

Referenced by pushConvClique().

C.5.0.88 cll_t* popAllCll (cll_t * *head*)

Shortcut function to pop all of the members of a linked list. Input: head of a linked list. Output: head of a now-empty linked list.

Definition at line 101 of file convll.c.

References popCll().

Referenced by completeConv(), and main().

C.5.0.89 cll_t* popCll (cll_t * *head*)

Removes the head of the clique linked list, returns the new head of the clique linked list, and frees the memory occupied by the old head. Input: head of a linked list. Output: head of a linked list.

Definition at line 60 of file convll.c.

References cSet_t::members, cnode::next, and cnode::set.

Referenced by popAllCll().

C.5.0.90 mll_t* popMemStack (mll_t * *head*)

Pops the head off of a single member linked list. Input: head of a member linked list. Output: the new head of a member linked list after popping one item.

Definition at line 388 of file convll.c.

References mnode::next.

Referenced by popWholeMemStack().

C.5.0.91 mll_t* popWholeMemStack (mll_t * *head*)

Pops all items off of a member linked list. Input: head of a member linked list. Output: empty head of a member linked list.

Definition at line 410 of file convll.c.

References popMemStack().

Referenced by emptyMemberStacks(), and singleCliqueConv().

C.5.0.92 int printCll (cll_t * *head*)

Prints the members (cliques) of a linked list in the format: *id* = unique id number of clique within linked list; *Length* = number of members of clique, if available; *Size* = length of each member of clique; *Members* = newline-separated list of members of the clique. Input: head of a linked list. Output: Gives text output, returns (meaningless) exit value.

Definition at line 118 of file convll.c.

References cnode::id, cnode::length, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

C.5.0.93 int printCllPattern (cll_t * *node*, int *length*)

Prints out the contents of a clique linked list node in this format: *support* = number of motif occurrences (*id* = some id number); *members* = newline-separated list of offsets. Input: a specific node to be output, the length of the motif inside it. Output: text per above, and an integer success value.

Definition at line 1328 of file convll.c.

References cnode::id, cSet_t::members, cnode::set, and cSet_t::size.

C.5.0.94 void printMemberStacks (mll_t ** *memberStacks*, int *size*)

Prints the contents of the member stacks. Input: array of member linked lists, size of that array (total number of offsets). Output: only text output/no return value.

Definition at line 491 of file convll.c.

References mnode::cliqueMembership, and mnode::next.

C.5.0.95 cll_t* pushClique (bitSet_t * *clique*, cll_t * *head*, int * *indexToSeq*, int *p*)

Pushes a bitSet onto a clique linked list, performing all necessary manipulations in order to do so. Input: new clique in the form of a bitSet_t, head of a linked list, pointer to the index/sequence number data structure, integer value of the -p flag. Output: head of an updated clique linked list.

Definition at line 314 of file convll.c.

References bitSetToCSet(), checkCliquecSet(), cliquecounter, cSet_t::members, and pushcSet().

Referenced by findCliques(), and singleLinkage().

C.5.0.96 cll_t* pushCll (cll_t * *head*)

Pushes a new, empty head onto a linked list of cliques. Note: this should always be followed by a call to initheadCll, as the head pushed on here is empty and will be meaningless without any members. This function should NOT be used by the user; see pushcSet. Input: head of a linked list. Output: head of a linked list.

Definition at line 26 of file convll.c.

References cnode::id, cnode::length, cnode::next, cnode::set, and cnode::stat.

Referenced by pushcSet().

C.5.0.97 cll_t* pushConvClique (mll_t * *clique*, cll_t * *head*)

Pushes a freshly-convolved clique, currently in mll_t form, onto the clique linked list for the next level. Also checks to make sure that the convolved clique is unique, and if it isn't, it takes appropriate action. Input: a convolved clique in mll_t form, the head of a clique linked list for the next level. Output: (potentially new) head of the clique linked list for the next level.

Definition at line 980 of file convll.c.

References cSet_t::members, mllToCSet(), pushcSet(), removeSupers(), swapNodecSet(), and uniqClique().

Referenced by singleCliqueConv().

C.5.0.98 cll_t* pushcSet (cll_t * *head*, cSet_t * *newset*)

Function that pushes the contents of a cSet (set of members of a clique) onto a linked list of cliques. Input: head of a linked list, new clique in the form of a cSet_t. Output: head of a linked list.

Definition at line 174 of file convll.c.

References initheadCll(), and pushCll().

Referenced by pushClique(), and pushConvClique().

C.5.0.99 mll_t* pushMemStack (mll_t * *head*, int *cliqueNum*)

This begins code for the member linked lists. A single one of these linked lists functions somewhat similarly to the clique linked lists, though with less information stored. Functionally, an array of member linked lists is used to access the "inverse" of what is contained in the clique linked lists. That is, we would like to be able to look up the cliques that a given node is a member of, so we have an array of member linked lists of size equal to the number of nodes.

This function pushes a single clique membership onto a node's member stack. Input: the head of a single member linked list, a clique number to be added. Output: the head of a single member linked list.

Definition at line 358 of file convll.c.

References mnode::cliqueMembership, and mnode::next.

Referenced by addToStacks(), and mergeIntersect().

C.5.0.100 cll_t* removeSupers (cll_t * *head*, int *node*, cSet_t * *newClique*)

This function finds all cliques in a linked list of which the proposed clique is a superset. It starts looking AFTER the first clique which has already been found to be a subset. In some senses, it is just a continuation of the uniqclique function in order to take advantage of the fact that though a proposed clique can only be a subset of one existing next-level clique, it can be a superset of many existing next- level cliques. Input: head of a clique linked list, the id of the first node found to be a subset of the proposed clique, and the proposed clique (in cSet_t form). Output: the head of the clique linked list with all but the first subset (which was passed as an argument) removed. This function is now ready for swapNode to be called.

Definition at line 849 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by pushConvClique().

C.5.0.101 bitSet_t* searchMemsWithList (int * *list*, int *listsize*, mll_t ** *memList*, int *numOffsets*, bitSet_t * *queue*)

Creates one large queue by calling "setStackTrue" for each member of a list of offsets. This then creates the union of clique membership for all offsets in the list being searched. Input: an array of offset numbers, the length of that array, an array of member linked lists, the length of that array (the total number of offsets), and a bitSet_t to store the union/queue. Output: the union/queue in a bitSet_t structure.

Definition at line 540 of file convll.c.

References emptySet(), and setStackTrue().

Referenced by wholeCliqueConv().

C.5.0.102 bitSet_t* setStackTrue (mll_t ** *memList*, int *i*, bitSet_t * *queue*)

Adds all of the members of a given stack to a "queue" in the form of a bitSet_t data structure. That is, for each clique in the member linked list, it sets the corresponding bit in the bitSet_t true. Input: array of member linked lists, an integer indicating a specific member linked list, and a bitSet_t of length >= the number of cliques in the current clique linked list. Ouput: the updated bitSet_t object.

Definition at line 516 of file convll.c.

References mnode::cliqueMembership, mnode::next, and setTrue().

Referenced by searchMemsWithList().

C.5.0.103 cll_t* singleCliqueConv (cll_t * *head*, int *firstClique*, cll_t ** *firstGuess*, int *secondClique*, cll_t ** *secondGuess*, cll_t * *nextPhase*, bitSet_t * *printStatus*, int *support*)

Convolves one single clique against one other single clique. Note that this is non-commutative, so exchanging firstClique and secondClique will not give the same results. The "guess" pointers keep the location of the previous clique in the linked list so that we don't have to search the linked list from the beginning/end every time. We exploit our earlier tidiness in that we can reasonably guess that we will monotonically traverse down cliques. Input: head of the current clique linked list, the id number of the first clique, a pointer to a guess at the first clique, the id number of the second clique, a pointer to a guess at the second clique, the head of the clique linked list for the next round of convolution, a bitSet indicating which cliques should be output as maximal, and the minimum support flag. Output: the head of clique linked list for the next round of convolution (which may have changed if the two cliques could be convolved).

Definition at line 580 of file convll.c.

References cnode::id, mergeIntersect(), cnode::next, popWholeMemStack(), pushConvClique(), cnode::set, setFalse(), and cSet_t::size.

Referenced by wholeCliqueConv().

C.5.0.104 cll_t* swapNodecSet (cll_t * *head*, int *node*, cSet_t * *newClique*)

Swaps out a node in a linked list that has been found to be a subset of a node that is not yet in the list. Input: the head of a clique linked list, a specific node within that linked list that is to be removed, and the new clique that is the superset of the node to be removed (in cSet_t form). Output: the head of the altered clique linked list.

Definition at line 804 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, and cnode::set.

Referenced by pushConvClique().

C.5.0.105 int uniqClique (cSet_t * *cliquecSet*, cll_t * *head*)

Before we push a convolved clique onto the stack for the next level, this function ensures that it is not subsumed by and does not subsume any other clique currently on that stack. Input: a candidate clique for the next level in cSet_t form, and the head of the clique linked list for the next level. Output: an integer indicating the status of the proposed clique with respect to the next level: -1 if the clique is unique, -2 if the clique is a subset/duplicate of an existing clique, or a clique id in the range [0,numcliques) representing the first clique of which the proposed one is a superset.

Note that by executing this each time a clique is added to the next level, we ensure that if the new clique is not unique, it can only be a superset or a subset of some other clique; it cannot be both a strictly superset of one and a strictly subset of another. One of those other two cliques would have been identified in previous steps as being super- or sub-sets, so it is impossible for one clique now to be both a super and a subset.

Definition at line 729 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by pushConvClique().

C.5.0.106 cll_t* wholeCliqueConv (cll_t * *head*, cll_t * *node*, cll_t ** *firstGuess*, mll_t ** *memList*, int *numOffsets*, cll_t * *nextPhase*, bitSet_t * *printStatus*, int *support*)

Convolves one single clique against all possible cliques that could possibly be convolved. It does not attempt to convolve all other cliques, but prunes that set by first looking at the offsets that are in the clique, then collecting all of the cliques who have members that are one greater than the offsets in this clique, and then convolving those cliques in a sort of "queue" using the bitSet_t data structure. Input: the head of the clique linked list for the current level, the current node being convolved against in the linked list, the location of the previous node in the form of a pointer to a "guess", an array of member linked lists, the length of that array, the head of the clique linked list for the next level, a bitSet_t for the printStatus of maximality, and the support criterion. Output: the head of the (possibly modified) clique linked list for the next level.

Definition at line 1081 of file convll.c.

References bitSetToCSet(), countSet(), deleteBitSet(), cnode::id, cSet_t::members, newBitSet(), searchMemsWithList(), cnode::set, singleCliqueConv(), and cSet_t::size.

Referenced by wholeRoundConv().

C.5.0.107 cll_t* wholeRoundConv (cll_t ** *head*, mll_t ** *memList*, int *numOffsets*, int *support*, int *length*, cll_t ** *allCliques*)

Performs convolution on all cliques in a linked list by repeatedly calling wholeCliqueConv. Input: pointer to the head of a clique linked list for the current level, array of member linked lists, length of that array, minimum support threshold, the current length of motifs, and a pointer to a linked list containing all cliques that will be printed out. Output: the head of the clique linked list for the next level of convolution.

Definition at line 1148 of file convll.c.

References checkBit(), deleteBitSet(), fillSet(), cnode::id, newBitSet(), cnode::next, wholeCliqueConv(), and yankCll().

Referenced by completeConv().

C.5.0.108 int yankCll (cll_t ** *head*, cll_t * *prev*, cll_t ** *curr*, cll_t ** *allCliques*, int *length*)

Removes a clique from within a linked list in order to save it for later printing. This is done so that the cliques are not printed as they are convolved, but rather after all rounds of convolution are complete. Input: a pointer to the head of the current linked list, the clique prior to the one that is to be yanked (NULL if the clique to be yanked is the head), the clique that is to be yanked, a pointer to the head of the list with all cliques that are to be printed, and the length of the current motif. Output: Nothing is returned beyond a success integer, but it alters the current level cll_t, the value of curr, and the linked list of all cliques that are to be printed.

Definition at line 1221 of file convll.c.

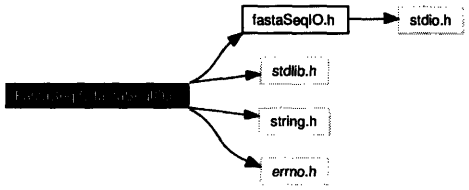
References cnode::id, and cnode::next.

Referenced by convolve(), and wholeRoundConv().

C.6 FastaSeqIO/fastaSeqIO.c File Reference

```
#include "fastaSeqIO.h"
#include <stdlib.h>
#include <string.h>
#include <errno.h>
```

Include dependency graph for fastaSeqIO.c:



Data Structures

- struct sSize_t

Defines

- #define BUFFER 100000
- #define BIG_BUFFER 1000000

Functions

- int printFSeqSubSeq (fSeq_t *seq, int start, int stop)
- long measureLine (FILE *INPUT)
- long CountFSeqs (FILE *INPUT)
- long countLines (FILE *INPUT)
- int initAofFSeqs (fSeq_t *aos, int numSeq)
- char ** ReadFile (FILE *INPUT, int *n)
- fSeq_t * ReadTxtSeqs (FILE *INPUT, int *numberOfSequences)
- fSeq_t * ReadFSeqs (FILE *INPUT, int *numberOfSequences)
- int FreeFSeqs (fSeq_t *arrayOfSequences, int numberOfSequences)
- int WriteFSeqA (FILE *MY_FILE, fSeq_t *arrayOfSequences, int start, int stop)

Define Documentation

C.6.0.109 #define BIG_BUFFER 1000000

Definition at line 11 of file fastaSeqIO.c.

C.6.0.110 #define BUFFER 100000

Definition at line 10 of file fastaSeqIO.c.

Function Documentation

C.6.0.111 long CountFSeqs (FILE * *INPUT*)

Definition at line 44 of file fastaSeqIO.c.

```
45 {
46     long start;
47     long count = 0;
48     int myChar;
49     int newLine = 1;
50     start = ftell(INPUT);
51     myChar = fgetc(INPUT);
52     while (myChar != EOF) {
53         if (newLine == 1 && myChar == '>') {
54             count++;
55         }
56         if (myChar == '\n') {
57             newLine = 1;
58         } else {
59             newLine = 0;
60         }
61         myChar = fgetc(INPUT);
62     }
63     fseek(INPUT, start, SEEK_SET);
64     return count;
65 }
```

C.6.0.112 long countLines (FILE * *INPUT*)

Definition at line 69 of file fastaSeqIO.c.

Referenced by ReadFile().

```
70 {
71     long start;
72     long count = 1;
73     int myChar;
74     int status = 0;
75     start = ftell(INPUT);
76     myChar = fgetc(INPUT);
77     while (myChar != EOF) {
78         if (myChar == '\n') {
79             count++;
```

```
80              status = 1;
81          } else {
82              status = 0;
83          }
84          myChar = fgetc(INPUT);
85      }
86      if (status == 1) {
87          count--;
88      }
89      fseek(INPUT, start, SEEK_SET);
90      return count;
91 }
```

C.6.0.113 int FreeFSeqs (fSeq_t * *arrayOfSequences*, int *numberOfSequences*)

Definition at line 304 of file fastaSeqIO.c.

References fSeq_t::label, and fSeq_t::seq.

Referenced by main().

```
305 {
306     int i;
307     for (i = 0; i < numberOfSequences; i++) {
308         if (arrayOfSequences[i].label != NULL) {
309             free(arrayOfSequences[i].label);
310         }
311         arrayOfSequences[i].label = NULL;
312
313         if (arrayOfSequences[i].seq != NULL) {
314             free(arrayOfSequences[i].seq);
315         }
316         arrayOfSequences[i].seq = NULL;
317     }
318     if (arrayOfSequences != NULL) {
319         free(arrayOfSequences);
320     }
321     arrayOfSequences = NULL;
322     return EXIT_SUCCESS;
323 }
```

C.6.0.114 int initAofFSeqs (fSeq_t * *aos*, int *numSeq*)

Definition at line 94 of file fastaSeqIO.c.

References fSeq_t::label, and fSeq_t::seq.

Referenced by ReadFSeqs(), and ReadTxtSeqs().

```
95 {
96     int i;
97     for (i = 0; i < numSeq; i++) {
98         aos[i].seq = NULL;
99         aos[i].label = NULL;
100     }
101     return 1;
102 }
```

C.6.0.115 long measureLine (FILE * *INPUT*)

Definition at line 25 of file fastaSeqIO.c.

Referenced by ReadFile().

```
26 {
27     long start;
28     long count = 0;
29     int myChar;
30     start = ftell(INPUT);
31     myChar = fgetc(INPUT);
32     count++;
33     while (myChar != '\n' && myChar != EOF) {
34         count++;
35         myChar = fgetc(INPUT);
36     }
37     fseek(INPUT, start, SEEK_SET);
38     return count;
39 }
```

C.6.0.116 int printFSeqSubSeq (fSeq_t * *seq*, int *start*, int *stop*)

Definition at line 14 of file fastaSeqIO.c.

References fSeq_t::seq.

```
14                                                  {
15     int i;
16     for(i=start; i<stop; i++){
17         putchar(seq->seq[i]);
18     }
19     return 0;
20 }
```

C.6.0.117 char** ReadFile (FILE * *INPUT*, int * *n*)

Definition at line 105 of file fastaSeqIO.c.

References countLines(), and measureLine().

Referenced by ReadFSeqs(), readRealData(), and ReadTxtSeqs().

```
106 {
107     char **buf = NULL;
108     long nl;
109     long tls = 0;
110     int i=0;
111
112     nl = countLines(INPUT);
113     if( nl == 0){
114         fprintf(stderr, "\nNo sequences! Error!\n\n");
115         fflush(stderr);
116         return NULL;
117     }
118     buf = (char **) malloc ( (int)(nl+1) * sizeof(char *));
119     if ( buf == NULL){
120         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
```

```
121         fflush(stderr);
122         exit(0);
123     }
124
125     //  measure the first line
126     tls = measureLine(INPUT) + 1;
127     if(tls != 0){
128         buf[i] = (char *) malloc ( tls * sizeof(char));
129         if ( buf[i] == NULL){
130             fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
131             fflush(stderr);
132             exit(0);
133         }
134     }
135     fgets(buf[i], tls, INPUT);
136     do{
137         if(buf[i][ strlen(buf[i])-1 ] == '\n'){
138             buf[i][ strlen(buf[i])-1 ] = '\0';
139         }
140         tls = measureLine(INPUT) + 1;
141         if(tls != 0){
142             i++;
143             buf[i] = (char *) malloc ( tls * sizeof(char) );
144             if ( buf[i] == NULL){
145                 fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
146                 fflush(stderr);
147                 exit(0);
148             }
149         }
150     }while( fgets(buf[i], tls, INPUT) != NULL );
151     free(buf[i]);
152     buf = (char **) realloc ( buf, i * sizeof(char *) );
153     if ( buf == NULL){
154         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
155         fflush(stderr);
156         return NULL;
157     }
158     //  I think that 'i' might actually be the # of lines
159     //  plus one here?  somehow line 131 isn't being freed,
160     //  or at least 2 bytes of it.
161     *n = i;
162     return buf;
163 }
```

C.6.0.118 fSeq_t* ReadFSeqs (FILE * *INPUT*, int * *numberOfSequences*)

Definition at line 199 of file fastaSeqIO.c.

References initAofFSeqs(), fSeq_t::label, ReadFile(), fSeq_t::seq, sSize_t::size, sSize_t::start, and sSize_t::stop.

Referenced by main().

```
199                                             {
200     int i,j,k;
201     int nl, ns=0;
202     char **buf = NULL;
203     fSeq_t *aos;
204     sSize_t *ss;
205     sSize_t *ll;
206
207     buf = ReadFile(INPUT, &nl);
```

```
208     if(buf == NULL){
209         return NULL;
210     }
211
212     //  Count how many sequences we have
213     for( j=0 ; j<nl ; j++){
214         if(buf[j][0] == '>'){
215             ns++;
216         }
217     }
218     ss = (sSize_t *) malloc ( ns * sizeof(sSize_t) );
219     if(ss == NULL){
220         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
221         fflush(stderr);
222         exit(0);
223     }
224     ll = (sSize_t *) malloc ( ns * sizeof(sSize_t) );
225     if(ll == NULL){
226         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
227         fflush(stderr);
228         exit(0);
229     }
230
231     //  find the first sequence
232     k=0;
233     while( buf[k][0] != '>'){
234         k++;
235     }
236
237     //  record how large each sequence is
238     i = -1;
239     for( j=k ; j<nl ; j++){
240         if(buf[j][0] == '>'){
241             i++;
242             ll[i].start = j;
243             ll[i].stop = j;
244             ll[i].size = strlen( buf[j] );;
245             ss[i].start = j+1;
246             ss[i].size = 0;
247         }else{
248             ss[i].stop = j;
249             ss[i].size += strlen( buf[j] );;
250         }
251     }
252
253     aos = (fSeq_t *) malloc ( ns * sizeof(fSeq_t));
254     if( aos == NULL){
255         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
256         fflush(stderr);
257         exit(0);
258     }
259     initAofFSeqs(aos, ns);
260
261     for ( i=0 ; i<ns ; i++ ){
262         if( ll[i].size > 0 ){
263             aos[i].label = (char *) malloc ( (ll[i].size+1) * sizeof(char) );
264             if( aos[i].label == NULL){
265                 fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
266                 fflush(stderr);
267                 exit(0);
268             }
269             aos[i].label[0] = '\0';
270             for ( j=ll[i].start ; j<=ll[i].stop ; j++ ){
271
272                 //  both instances of strcat here are using
273                 //  .label/.seq's that are NULL and that is
274                 //  throwing a memory error in valgrind
275                 aos[i].label = strcat ( aos[i].label, buf[j] );
```

```
276             }
277         }
278         if( ss[i].size > 0 ){
279             aos[i].seq = (char *) malloc ( (ss[i].size+1) * sizeof(char) );
280             if( aos[i].seq == NULL){
281                 fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
282                 fflush(stderr);
283                 exit(0);
284             }
285             aos[i].seq[0] = '\0';
286             for ( j=ss[i].start ; j<=ss[i].stop ; j++ ){
287                 aos[i].seq = strcat ( aos[i].seq, buf[j] );
288             }
289         }
290     }
291     free(ll);
292     free(ss);
293
294     for ( i=0 ; i<nl ; i++ ){
295         free(buf[i]);
296     }
297     free(buf);
298
299     *numberOfSequences = ns;
300     return aos;
301 }
```

C.6.0.119 fSeq_t* ReadTxtSeqs (FILE * *INPUT*, int * *numberOfSequences*)

Definition at line 172 of file fastaSeqIO.c.

References initAofFSeqs(), ReadFile(), and fSeq_t::seq.

```
172                                                 {
173     int i;
174     int nl;
175     char **buf = NULL;
176     fSeq_t *aos;
177
178     buf = ReadFile(INPUT, &nl);
179     if(buf == NULL){
180         return NULL;
181     }
182     aos = (fSeq_t *) malloc ( nl * sizeof(fSeq_t));
183     if( aos == NULL){
184         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
185         fflush(stderr);
186         exit(0);
187     }
188     initAofFSeqs(aos, nl);
189     for ( i=0 ; i<nl ; i++ ){
190         aos[i].seq = buf[i];
191     }
192     free(buf);
193     *numberOfSequences = nl;
194     return (aos);
195 }
```

C.6.0.120 int WriteFSeqA (FILE * *MY_FILE*, fSeq_t * *arrayOfSequences*, int *start*, int *stop*)

Definition at line 330 of file fastaSeqIO.c.

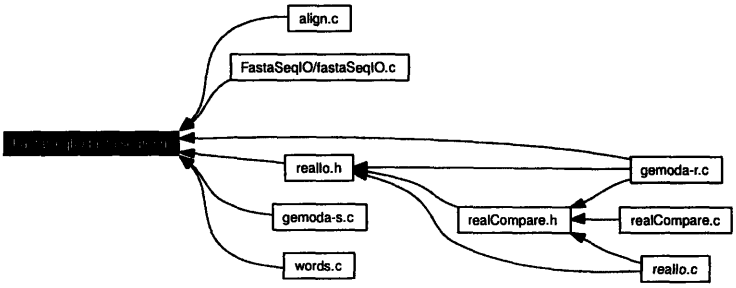
```
331 {
332     int i;
333     for (i = start; i <= stop; i++) {
334         fprintf(MY_FILE, "%s\n", arrayOfSequences[i].label);
335         fprintf(MY_FILE, "%s\n", arrayOfSequences[i].seq);
336     }
337     return EXIT_SUCCESS;
338 }
```

C.7 FastaSeqIO/fastaSeqIO.h File Reference

`#include <stdio.h>`

Include dependency graph for fastaSeqIO.h:

This graph shows which files directly or indirectly include this file:



Data Structures

- struct fSeq_t

Functions

- int printFSeqSubSeq (fSeq_t *seq, int start, int stop)
- long measureLine (FILE *INPUT)
- long countLines (FILE *INPUT)
- long CountFSeqs (FILE *INPUT)
- int initAofFSeqs (fSeq_t *aos, int numSeq)
- fSeq_t * ReadFSeqs (FILE *INPUT, int *numberOfSequences)
- int FreeFSeqs (fSeq_t *arrayOfSequences, int numberOfSequences)
- int WriteFSeqA (FILE *MY_FILE, fSeq_t *arrayOfSequences, int start, int stop)
- fSeq_t * ReadTxtSeqs (FILE *INPUT, int *numberOfSequences)

Function Documentation

C.7.0.121 long CountFSeqs (FILE * *INPUT*)

Definition at line 44 of file fastaSeqIO.c.

C.7.0.122 long countLines (FILE * *INPUT*)

Definition at line 69 of file fastaSeqIO.c.

Referenced by ReadFile().

C.7.0.123 int FreeFSeqs (fSeq_t * *arrayOfSequences*, int *numberOfSequences*)

Definition at line 306 of file fastaSeqIO.c.

References fSeq_t::label, and fSeq_t::seq.

Referenced by main().

C.7.0.124 int initAofFSeqs (fSeq_t * *aos*, int *numSeq*)

Definition at line 94 of file fastaSeqIO.c.

References fSeq_t::label, and fSeq_t::seq.

Referenced by ReadFSeqs(), and ReadTxtSeqs().

C.7.0.125 long measureLine (FILE * *INPUT*)

Definition at line 25 of file fastaSeqIO.c.

Referenced by ReadFile().

C.7.0.126 int printFSeqSubSeq (fSeq_t * *seq*, int *start*, int *stop*)

Definition at line 14 of file fastaSeqIO.c.

References fSeq_t::seq.

C.7.0.127 fSeq_t* ReadFSeqs (FILE * *INPUT*, int * *numberOfSequences*)

Definition at line 199 of file fastaSeqIO.c.

References initAofFSeqs(), fSeq_t::label, ReadFile(), fSeq_t::seq, sSize_t::size, sSize_t::start, and sSize_t::stop.

Referenced by main().

C.7.0.128 fSeq_t* ReadTxtSeqs (FILE * *INPUT*, int * *numberOfSequences*)

Definition at line 172 of file fastaSeqIO.c.

References initAofFSeqs(), ReadFile(), and fSeq_t::seq.

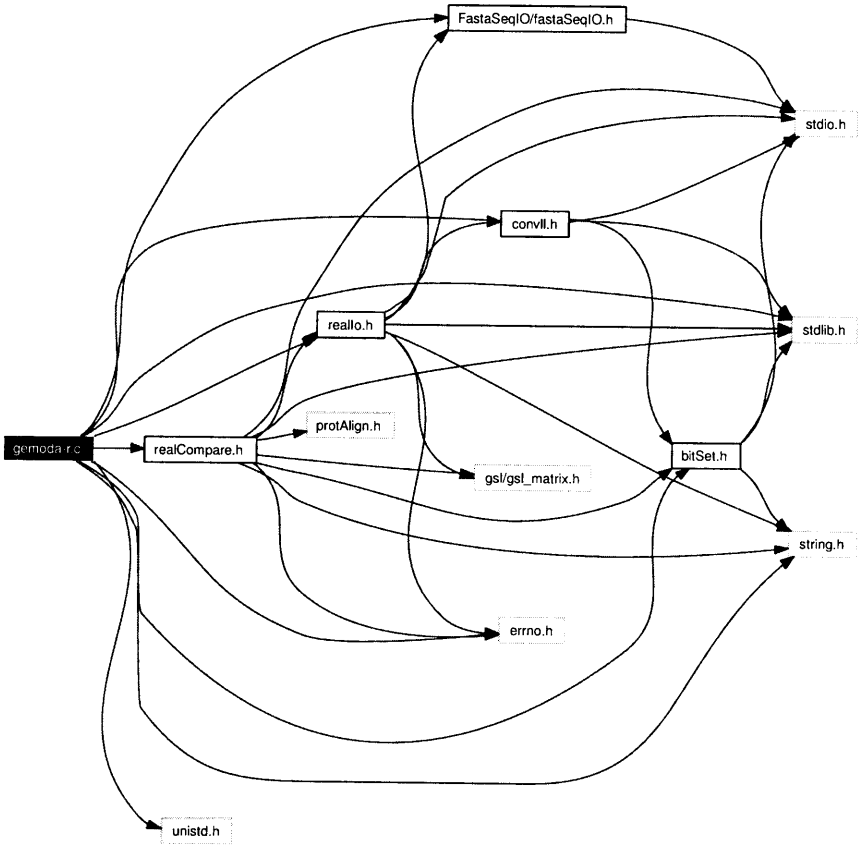
C.7.0.129 int WriteFSeqA (FILE * *MY_FILE*, fSeq_t * *arrayOfSequences*, int *start*, int *stop*)

Definition at line 332 of file fastaSeqIO.c.

C.8 gemoda-r.c File Reference

```
#include "bitSet.h"
#include "convll.h"
#include "FastaSeqIO/fastaSeqIO.h"
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include "realIo.h"
#include "realCompare.h"
```

Include dependency graph for gemoda-r.c:



Functions

- void usage (char **argv)

- cll_t * convolve (bitGraph_t *bg, int support, int R, int *indexToSeq, int p, int clusterMethod, int **offsetToIndex, int numberOfSequences, int noConvolve, FILE *OUTPUT_FILE)
- bitGraph_t * pruneBitGraph (bitGraph_t *bg, int *indexToSeq, int **offsetToIndex, int numOfSeqs, int p)
- int countExtraParams (char *s)
- double * parseExtraParams (char *s, int numParams)
- int main (int argc, char **argv)

Detailed Description

This file contains the main routine for the real valued version of Gemoda. There are also some accessory functions for printing information on how to use Gemoda and run it from the commandline.

Definition in file gemoda-r.c.

Function Documentation

C.8.0.130 cll_t* convolve (bitGraph_t * *bg*, int *support*, int *R*, int * *indexToSeq*, int *p*, int *clusterMethod*, int ** *offsetToIndex*, int *numberOfSequences*, int *noConvolve*, FILE * *OUTPUT_FILE*)

Our outer convolution function. This function will call preliminary functions, cluster the data, and then call the main convolution function. This is the interface between the main gemoda-<x> code and the generic code that gets all of the work done. Input: the bitGraph to be clustered and convolved, the minimum support necessary for a motif to be returned, a flag indicating whether recursive filtering should be used, a pointer to the data structure that dereferences offset indices to sequence numbers, the number of unique source sequences that a motif must be present in, and a number indicating the clustering method that is to be used. Output: the final motif linked list with all motifs that are to be given as output to the user.

Definition at line 625 of file newConv.c.

Referenced by main().

```
629 {
630   bitSet_t * cand = NULL;
631   bitSet_t * mask = NULL;
632   bitSet_t * Q = NULL;
633   int size = bg->size;
634   cll_t * elemPats = NULL;
635   cll_t * allCliques = NULL;
636   cll_t * curr = NULL;
637
638     // contains indices (rows) containing the threshold value.
```

```
639     cand = newBitSet (size);
640   mask = newBitSet (size);
641   Q = newBitSet (size);
642   fillSet (cand);
643   fillSet (mask);
644
645     // Note that we prune based on p before setting the diagonal false.
646     if (p > 1)
647     {
648       bg =
649     pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
650     }
651
652     // Now we set the main diagonal false for clustering and filtering.
653     bitGraphSetFalseDiagonal (bg);
654   filterGraph (bg, support, R);
655   fprintf (OUTPUT_FILE, "Graph filtered!  Now clustering...\n");
656   fflush (NULL);
657   if (clusterMethod == 0)
658     {
659       findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
660     }
661   else
662     {
663       singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
664              p);
665     }
666   fprintf (OUTPUT_FILE,
667        "Clusters found!  Now filtering clusters (if option set)...\n");
668   fflush (NULL);
669   if (p > 1)
670     {
671       elemPats = pruneCll (elemPats, indexToSeq, p);
672     }
673   deleteBitSet (cand);
674   deleteBitSet (mask);
675   deleteBitSet (Q);
676
677     // Now let's convolve what we made.
678     if (noConvolve == 0)
679     {
680       fprintf (OUTPUT_FILE, "Now convolving...\n");
681       fflush (NULL);
682       allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
683     }
684
685   else
686     {
687       curr = elemPats;
688       while (curr != NULL)
689     {
690       yankCll (&elemPats, NULL, &curr, &allCliques, 0);
691     }
692     }
693   return allCliques;
694 }
```

C.8.0.131 int countExtraParams (char * *s*)

Definition at line 91 of file gemoda-r.c.

Referenced by main().

```
92 {
```

```
93   int i = 0;
94   int numParams = 1;
95   for (i = 0; i < strlen (s); i++)
96     {
97       if (s[i] == ',')
98     {
99       numParams++;
100     }
101     }
102   return numParams;
103 }
```

C.8.0.132 int main (int *argc*, char ** *argv*)

This is the main routine of the real value Gemoda code. The code runs similarly to the sequence Gemoda code: there is a comparison phase, followed by a clustering phase, followed by a convolution phase. Only the comparison phase is unique to the real value Gemoda. Of course, since the data are formatted so differently, there are vastly different pieces of code in the front matter. In particular, there is no hashing of words obviously. As well, we use the GNU scientific library to store real value data as matrices that can be easily manipulated.

Definition at line 160 of file gemoda-r.c.

References calcStatAllCliqs(), convolve(), countExtraParams(), cumDMatrix(), deleteBitGraph(), freeD(), freeRdh(), getStatMat(), rdh_t::indexToSeq, rdh_t::offsetToIndex, outputRealPats(), outputRealPatsWCentroid(), parseExtraParams(), popAllCll(), readRealData(), realComparison(), bitGraph_t::size, rdh_t::size, sortByStats(), and usage().

```
161 {
162   int inputOption = 0;
163   char *sequenceFile = NULL;
164   FILE *SEQUENCE_FILE = NULL;
165   char *outputFile = NULL;
166   FILE *OUTPUT_FILE = NULL;
167   int L = 0;
168   int status = 0;
169   double g = 0;
170   int sup = 2;
171   int R = 1;
172   int P = 0;
173   int compFunc = 0;
174   double *extraParams = NULL;
175   int numExtraParams = 0;
176   int i = 0, j = 0;
177   /*
178      int j, k, i, l;
179    */
180   int noConvolve = 0;
181   int samp = 1;
182   int supportDim = 0, lengthDim = 0;
183   bitGraph_t *oam = NULL;
184   unsigned int **d = NULL;
185   int oamSize = 0;
186
187   cll_t *allCliques = NULL;
188   /*
189      cll_t *curCliq = NULL;
```

```
190    */
191   /*
192      int curSeq;
193    */
194   /*
195      int curPos;
196    */
197   int clusterMethod = 0;
198   int joelOutput = 0;
199
200   // gemoda-r new stuff
201   rdh_t *data = NULL;
202
203   /*
204      Get command-line options
205    */
206   while ((inputOption = getopt (argc, argv, "p:m:e:i:o:l:g:k:c:njs:")) != EOF)
207     {
208       switch (inputOption)
209     {
210       // Comparison metric
211     case 'm':
212       compFunc = atoi (optarg);
213       break;
214       // Input file
215     case 'i':
216       sequenceFile = optarg;
217       break;
218       // Output file
219     case 'o':
220       outputFile =
221         (char *) malloc ((strlen (optarg) + 1) * sizeof (char));
222       if (outputFile == NULL)
223         {
224           fprintf (stderr, "Error allocating memory for options.\n");
225           exit (EXIT_FAILURE);
226         }
227       else
228         {
229           strcpy (outputFile, optarg);
230         }
231       break;
232       // Minimum motif length
233     case 'l':
234       L = atoi (optarg);
235       break;
236       // Minimum motif similarity score
237     case 'g':
238       g = atof (optarg);
239       status++;
240       break;
241       // Minimum support (number of motif occurrences)
242     case 'k':
243       sup = atoi (optarg);
244       break;
245
246 /*****************************************************************
247  * Recursive initial pruning: an option for clique finding.
248  *   It takes all nodes with less than the minimum
249  *   number of support and removes all of their nodes, and does this
250  *   recursively so that nodes that are connected to many sparsely connected
251  *   nodes will be removed and not left in the
252  * This option is deprecated as it is at worst no-gain and at best useful.
253  *   It will be on by default for clique-finding, but can be turned
254  *   back off with some
255  *   minor tweaking.  For almost all cases in which it does not speed
256  *   up computations, it will have a trivial time to perform.  Thus, if
257  *   clique-finding is turned on, then R is set to 1 by default.
```

```
258         case 'r':
259             R = 1;
260             break;
261 ***************************************************************************/
262       // Optional pruning parameter to require at motif occurrences
263       // in at least P distinct input sequences
264
265     case 'p':
266       P = atoi (optarg);
267       break;
268
269       // Clustering method.
270     case 'c':
271       clusterMethod = atoi (optarg);
272       break;
273       // Extra parameters for comparison function
274     case 'e':
275       numExtraParams = countExtraParams (optarg);
276       extraParams = parseExtraParams (optarg, numExtraParams);
277       break;
278     case 'n':
279       noConvolve = 1;
280       break;
281     case 'j':
282       joelOutput = 1;
283       break;
284     case 's':
285       samp = atoi (optarg);
286       break;
287       // Catch-all.
288     case '?':
289       fprintf (stderr, "Unknown option `-%c'.\n", optopt);
290       usage (argv);
291       return EXIT_SUCCESS;
292     default:
293       usage (argv);
294       return EXIT_SUCCESS;
295     }
296     }
297   // Require an input file, a nonzero length, and a similarity threshold
298   // to be set.
299   if (sequenceFile == NULL || L == 0 || status < 1)
300     {
301       usage (argv);
302       return EXIT_SUCCESS;
303     }
304   // Open the sequence file
305   if ((SEQUENCE_FILE = fopen (sequenceFile, "r")) == NULL)
306     {
307       fprintf (stderr, "Couldn't open file %s; %s\n", sequenceFile,
308            strerror (errno));
309       exit (EXIT_FAILURE);
310     }
311   // Open the output file
312   if (outputFile != NULL)
313     {
314       if ((OUTPUT_FILE = fopen (outputFile, "w")) == NULL)
315     {
316       fprintf (stderr, "Couldn't open file %s; %s\n", outputFile,
317            strerror (errno));
318       exit (EXIT_FAILURE);
319     }
320     }
321   else
322     {
323       OUTPUT_FILE = stdout;
324     }
325
```

```
326
327
328   // Verbosity in output helps to distinguish output files.
329   fprintf (OUTPUT_FILE, "Input file = %s\n", sequenceFile);
330   fprintf (OUTPUT_FILE, "l = %d, k = %d, g = %f\n", L, sup, g);
331   if (P > 1)
332     {
333       fprintf (OUTPUT_FILE, "Minimum # of sequences with motif = %d\n", P);
334     }
335   if (R > 0)
336     {
337       fprintf (OUTPUT_FILE, "Recursive pruning is ON.\n");
338     }
339
340   data = readRealData (SEQUENCE_FILE);
341   fclose (SEQUENCE_FILE);
342   // printf("size = %d,indexSize = %d\n",data->size,data->indexSize);
343   // printf("size1 = %d,size2 = %d\n",data->seq[0]->size1,data->seq[0]->size2);
344   // for(i = 0; i < 2; i++) {
345   // for(j = 0; j < 3; j++) {
346   // printf("%lf,%lf,%lf\n",gsl_matrix_get(data->seq[i],j,0),
347   // gsl_matrix_get(data->seq[i],j,1),
348   // gsl_matrix_get(data->seq[i],j,2));}}
349   oam = realComparison (data, L, g, compFunc, extraParams);
350   // printf("oam->size = %d\n", oam->size);
351   if ((samp > 0) && (clusterMethod == 0))
352     {
353       // We are currently using one gap per sequence, as done in
354       // realCompare.c's call to initRdhIndex in realComparison.
355       // Note that this is data->size, NOT oam->size.
356       d =
357     getStatMat (oam, sup, L, &supportDim, &lengthDim, data->size, samp,
358             OUTPUT_FILE);
359     }
360   else
361     {
362       d = NULL;
363       supportDim = 0;
364     }
365
366   allCliques =
367     convolve (oam, sup, R, data->indexToSeq, P, clusterMethod,
368           data->offsetToIndex, data->size, noConvolve, OUTPUT_FILE);
369
370   oamSize = oam->size;
371   // Do some early memory cleanup since this is so big.
372   deleteBitGraph (oam);
373
374   if ((samp > 0) && (clusterMethod == 0))
375     {
376       cumDMatrix (d, allCliques, supportDim, lengthDim, oamSize, data->size);
377       calcStatAllCliqs (d, allCliques, oamSize - data->size);
378       allCliques = sortByStats (allCliques);
379     }
380
381   if (joelOutput == 0)
382     {
383       outputRealPats (data, allCliques, L, OUTPUT_FILE, d);
384     }
385   else
386     {
387       outputRealPatsWCentroid (data, allCliques, L, OUTPUT_FILE, extraParams,
388                    compFunc);
389     }
390
391   freeD (d, supportDim);
392   freeRdh (data);
393   allCliques = popAllCll (allCliques);
```

```
394   fclose (OUTPUT_FILE);
395
396   return 0;
397 }
```

C.8.0.133 double* parseExtraParams (char * *s*, int *numParams*)

This was borrowed from the old gemoda-p code, there it used to parse filenames, here we are parsing comma-separated lists of doubles that are useful for SpecConnect.

Definition at line 110 of file gemoda-r.c.

Referenced by main().

```
111 {
112   int i = 0, j = 0, k = 0;
113   int startLength = 0;
114   double *extraParams = NULL;
115   char *paramString = NULL;
116
117   extraParams = (double *) malloc (numParams * sizeof (double));
118   if (extraParams == NULL)
119     {
120       fprintf (stderr, "Can't allocate extra params!\n");
121       exit (0);
122     }
123   j = 0;
124   k = 0;
125   startLength = strlen (s);
126   for (i = 0; i < startLength; i++)
127     {
128       if (s[i] == ',')
129     {
130       // We've found an end.  So point the pointer to
131       // the beginning of the previous string.
132       paramString = &s[k];
133       // Terminate the string where the comma used to be
134       s[i] = '\0';
135       // Update the location for the next string beginning
136       k = i + 1;
137       // Convert to a double and update the param number.
138       extraParams[j] = atof (paramString);
139       j++;
140     }
141     }
142   // Don't forget to do the last one, which isn't comma-terminated.
143   paramString = &s[k];
144   extraParams[j] = atof (paramString);
145   return (extraParams);
146 }
```

C.8.0.134 bitGraph_t* pruneBitGraph (bitGraph_t * *bg*, int * *indexToSeq*, int ** *offsetToIndex*, int *numOfSeqs*, int *p*)

Simple function (non-recursive) to prune off the first level of motifs that will not meet the "minimum number of unique sequences" criterion. This could have been implemented as above, but it may have gotten a little expensive with less yield, so only the first run through is done here. Input: a bit graph to be pruned, a pointer to

the structure that dereferences offset indices to sequence numbers, a pointer to the structure that dereferences seq/position to offsets, the number of unique sequences in the input set, and the minimum number of unique sequences that must contain the motif. Output: a pruned bitGraph.

Definition at line 402 of file newConv.c.

Referenced by convolve().

```
404 {
405   int i = 0, j = 0, nextBit = 0;
406   int *seqNums = NULL;
407
408     // Since we don't immediately know which node is in which source
409     // sequence, we can't just count them up regularly.  Instead, we'll
410     // need to keep track of which sequences they come from and
411     // increment _something_.  What we chose to do here is just make
412     // an array of integers of length = <p>.  Then, we try to put the
413     // source sequence number of each neighbor (including itself, since
414     // the main diagonal is still true at this time) into the next slot
415     // Since we will monotonically search the bitSet, we can just
416     // move on to the first bit in the next sequence using the
417     // offsetToIndex structure so that we know the next sequence number
418     // to be put in is always unique.
419     seqNums = (int *) malloc (p * sizeof (int));
420   if (seqNums == NULL)
421     {
422       fprintf (stderr, "Memory error - pruneBitGraph\n%s\n",
423         strerror (errno));
424       fflush (stderr);
425       exit (0);
426     }
427
428     // So, for each row in the bitgraph...
429     for (i = 0; i < bg->size; i++)
430     {
431
432     // Make sure the whole array is -1 sentinels.
433     for (j = 0; j < p; j++)
434     {
435       seqNums[j] = -1;
436     }
437       j = 0;
438
439     // Find the first neighbor of this bit.
440     nextBit = nextBitBitSet (bg->graph[i], 0);
441       if (nextBit == -1)
442     {
443       continue;
444     }
445       else
446     {
447
448         // and put its sequence number in the array of ints.
449         seqNums[0] = indexToSeq[nextBit];
450     }
451
452     // If it's the last sequence, then bail out so that we don't
453     // segfault in the next step.
454     if (seqNums[0] >= numOfSeqs - 1)
455     {
456       emptySet (bg->graph[i]);
457       continue;
458     }
459
```

```
460     // Find the next neighbor of this bit, STARTING AT the first
461     // bit in the next sequence.
462     nextBit =
463     nextBitBitSet (bg->graph[i],
464                offsetToIndex[indexToSeq[nextBit] + 1][0]);
465
466     // And iterate this until we run out of neighbors.
467     while (nextBit >= 0)
468     {
469       j++;
470       seqNums[j] = indexToSeq[nextBit];
471
472         // Or until this new neighbor will fill up the array
473         if (j == p - 1)
474         {
475           break;
476         }
477
478         // Or until this new neighbor is in the last sequence.
479         if (seqNums[j] >= numOfSeqs - 1)
480         {
481           break;
482         }
483
484         // Get the next neighbor!
485         nextBit =
486         nextBitBitSet (bg->graph[i],
487                offsetToIndex[indexToSeq[nextBit] + 1][0]);
488     }
489
490     // If we didn't have enough unique sequences, and either a) we
491     // were in the nth-to-last sequence and there were no
492     // neighbors after it, or b) we were in the last sequence,
493     // then the last number will still be our sentinel, -1.  If
494     // the last number is not a sentinel, then we have at least
495     // p distinct sequence occurrences, so we're OK.
496     if (seqNums[p - 1] == -1)
497     {
498       emptySet (bg->graph[i]);
499     }
500     }
501   free (seqNums);
502   return (bg);
503 }
```

C.8.0.135 void usage (char ** *argv*)

This function tells the user how to run Gemoda. The function displays all the available flags and gives an example of how to use the commandline to run the code.

Definition at line 35 of file gemoda-r.c.

Referenced by main().

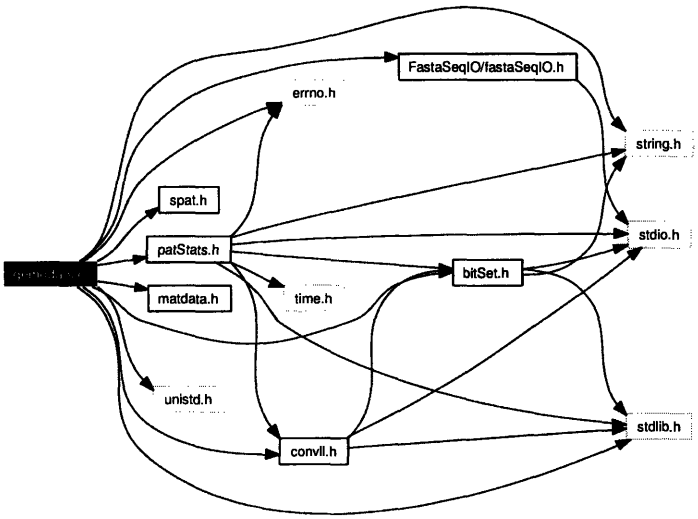
```
36 {
37   fprintf (stdout,
38        "Usage: %s -i <Fasta sequence file> "
39        "-l <word size> \n\t-k <support> -g <threshold> "
  "-m <matrix name> [-z] \n\t[-c <cluster method [0|1]>]"
  "[-p <unique support>] \n\n\n"
40        "Required flags and input:\n\n"
41        "-i <Fasta sequence file>:\n\t"
42        "File containing all sequences to be searched, in Fasta format.\n\n"
43        "-l <word size>:\n\t"
```

```
44        "Minimum length of motifs; also the sliding window length\n\t"
45        "over which all motifs must meet the similarity criterion\n\n"
46        "-k <support>:\n\t"
47        "Minimum number of motif occurrences.\n\n"
48        "-g <threshold>:\n\t"
49        "Similarity threshold.  Two windows, when scored with the\n\t"
50        " similarity matrix defined by the -m flag, must have at least\n\t"
51        " this score in order to be deemed 'connected'.  This criterion\n\t"
52        " must be met over all sliding windows of length l.\n\n"
53        "-c <cluster method [0|1]>:\n\t"
54        "The clustering method to be used after evaluating the "
55        "\n\tsimilarity of the unique words in the input.  Note that the "
56        "\n\tclustering method will have a significant impact on both the "
57        "\n\tresults that one obtains and the computation time.\n\n\t"
58        "0: clique-finding\n\t\t"
59        "Uses established methods to find all maximal cliques in the "
60        "\n\t\tdata.  This will give the most thorough results (that are "
61        "\n\t\tprovably exhaustive), but will also give less-significant "
62        "\n\t\tresults in addition to the most interesting and most\n\t"
63        "significant ones.  The results are deterministic but may take some "
64        "\n\t\ttime on data sets with high similarity or if the similarity "
65        "\n\t\tthreshold is set extremely low.\n\t"
66        "1: single-linkage clustering\n\t\t"
67        "Uses a single-linkage-type clustering where all nodes that "
68        "\n\t\tare connected are put in the same cluster.  This method is "
69        "\n\t\talso deterministic and will be faster than clique-finding, "
70        "\n\t\tbut it loses guarantees of exhaustiveness in searching the "
71        "\n\t\tdata set.\n\n",
72        "-p <unique support>:\n\t"
73        "A pruning parameter that requires the motif to occur in "
74        "\n\tat least <unique support> different input sequences.  Note "
75        "\n\tthat this parameter must be less than or equal to the total "
76        "\n\tsupport parameter set by the -k flag.\n\n", argv[0]);
77    fprintf (stdout, "\n");
78 }
```

C.9 gemoda-s.c File Reference

```
#include "bitSet.h"
#include "spat.h"
#include "convll.h"
#include "matdata.h"
#include "FastaSeqIO/fastaSeqIO.h"
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include "patStats.h"
```

Include dependency graph for gemoda-s.c:



Functions

- void usage (char **argv)
- void matrixlist (void)
- void getMatrixByName (char name[], int mat[][MATRIX_SIZE])
- bitGraph_t * alignWordsMat_bit (sPat_t *words, int wc, int mat[][MATRIX_SIZE], int threshold)

- sPat_t * countWords2 (fSeq_t *seq, int numSeq, int L, int *numWords)
- cll_t * convolve (bitGraph_t *bg, int support, int R, int *indexToSeq, int p, int clusterMethod, int **offsetToIndex, int numberOfSequences, int noConvolve, FILE *OUTPUT_FILE)
- bitGraph_t * pruneBitGraph (bitGraph_t *bg, int *indexToSeq, int **offsetToIndex, int numOfSeqs, int p)
- int main (int argc, char **argv)

Detailed Description

This file houses the main routine for the sequence based Gemoda algorithm. In addition, there are a few helper functions which are used to inform the user how to run the software.

The Gemoda algorithm has three stages: comparison, clustering, and convolution. These three stages are called in serial from the main routine in this file.

Definition in file gemoda-s.c.

Function Documentation

C.9.0.136 bitGraph_t* alignWordsMat_bit (sPat_t * *words*, int *wc*, int *mat*[][MATRIX_SIZE], int *threshold*)

This uses the function above. Here, we have an array of words (sPat_t objects) and we compare (align) them all. If their score is above 'threshold' then we will set a bit to 'true' in a bitGraph_t that we create. A bitGraph_t is essentially an adjacency matrix, where each member of the matrix contains only a single bit: are the words equal, true or false? The function traverses the words by doing and all by all comparison; however, we only do the upper diagonal. The function makes use of alignMat and needs to be passed a scoring matrix that the user has chosen which is appropriate for the context of whatever data sent the user is looking at.

Definition at line 88 of file align.c.

References alignMat(), bitGraphSetTrueSym(), mat, and newBitGraph().

Referenced by main().

```
90 {
91   bitGraph_t * sg = NULL;
92   int score;
93   int i, j;
94
95     // Assign a new bitGraph_t object, with (wc x wc) possible
96     // true/false values
97     sg = newBitGraph (wc);
98   for (i = 0; i < wc; i++)
99     {
100       for (j = i; j < wc; j++)
```

```
101     {
102
103         // Get the score for the alignment of word i and word j
104         score =
105         alignMat (words[i].string, words[j].string, words[i].length, mat);
106
107         // If that score is greater than threshold, set
108         // a bit to 'true' in our bitGraph_t object
109         if (score >= threshold)
110         {
111
112         // We use 'bitGraphSetTrueSym' because, if i=j,
113         // then j=i for most applications.  However, this
114         // can be relaxed for masochists.
115         bitGraphSetTrueSym (sg, i, j);
116         }
117     }
118     }
119
120     // Return a pointer to this new bitGraph_t object
121     return sg;
122 }
```

C.9.0.137 cll_t* convolve (bitGraph_t * *bg*, int *support*, int *R*, int * *indexToSeq*, int *p*, int *clusterMethod*, int ** *offsetToIndex*, int *numberOfSequences*, int *noConvolve*, FILE * *OUTPUT_FILE*)

Our outer convolution function. This function will call preliminary functions, cluster the data, and then call the main convolution function. This is the interface between the main gemoda-<x> code and the generic code that gets all of the work done. Input: the bitGraph to be clustered and convolved, the minimum support necessary for a motif to be returned, a flag indicating whether recursive filtering should be used, a pointer to the data structure that dereferences offset indices to sequence numbers, the number of unique source sequences that a motif must be present in, and a number indicating the clustering method that is to be used. Output: the final motif linked list with all motifs that are to be given as output to the user.

Definition at line 625 of file newConv.c.

References bitGraphSetFalseDiagonal(), completeConv(), deleteBitSet(), fillSet(), filterGraph(), findCliques(), newBitSet(), pruneBitGraph(), pruneCll(), single-Linkage(), bitGraph_t::size, and yankCll().

```
629 {
630   bitSet_t * cand = NULL;
631   bitSet_t * mask = NULL;
632   bitSet_t * Q = NULL;
633   int size = bg->size;
634   cll_t * elemPats = NULL;
635   cll_t * allCliques = NULL;
636   cll_t * curr = NULL;
637
638     // contains indices (rows) containing the threshold value.
639     cand = newBitSet (size);
640   mask = newBitSet (size);
641   Q = newBitSet (size);
```

```
642   fillSet (cand);
643   fillSet (mask);
644
645     // Note that we prune based on p before setting the diagonal false.
646     if (p > 1)
647     {
648       bg =
649     pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
650     }
651
652     // Now we set the main diagonal false for clustering and filtering.
653     bitGraphSetFalseDiagonal (bg);
654   filterGraph (bg, support, R);
655   fprintf (OUTPUT_FILE, "Graph filtered!  Now clustering...\n");
656   fflush (NULL);
657   if (clusterMethod == 0)
658     {
659       findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
660     }
661   else
662     {
663       singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
664              p);
665     }
666   fprintf (OUTPUT_FILE,
667        "Clusters found!  Now filtering clusters (if option set)...\n");
668   fflush (NULL);
669   if (p > 1)
670     {
671       elemPats = pruneCll (elemPats, indexToSeq, p);
672     }
673   deleteBitSet (cand);
674   deleteBitSet (mask);
675   deleteBitSet (Q);
676
677     // Now let's convolve what we made.
678     if (noConvolve == 0)
679     {
680       fprintf (OUTPUT_FILE, "Now convolving...\n");
681       fflush (NULL);
682       allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
683     }
684
685   else
686     {
687       curr = elemPats;
688       while (curr != NULL)
689     {
690       yankCll (&elemPats, NULL, &curr, &allCliques, 0);
691     }
692     }
693   return allCliques;
694 }
```

C.9.0.138 sPat_t* countWords2 (fSeq_t * *seq*, int *numSeq*, int *L*, int * *numWords*)

Counts words of size L in the input FastA sequences, hashes all of the words, and returns an array of sPat_t objects.

Definition at line 373 of file words.c.

References sHashEntry_t::data, destroySHash(), sHashEntry_t::idx, initSHash(), s-

HashEntry_t::key, sHashEntry_t::L, sPat_t::length, sOffset_t::next, sPat_t::offset, sOffset_t::pos, sOffset_t::prev, searchSHash(), sOffset_t::seq, sieve3(), sPat_t::string, and sPat_t::support.

Referenced by main().

```
374 {
375   int i, j;
376   int totalChars = 0;
377   int hashSize;
378   sHashEntry_t newEntry;
379   sHashEntry_t *ep;
380   sHash_t wordHash;
381   sPat_t *words = NULL;
382   int wc = 0;
383   int prev = -1;
384   int l;
385
386
387   // Count the total number of characters.  This
388   // is the upper limit on how many words we can have
389   for (i = 0; i < numSeq; i++)
390     {
391       totalChars += strlen (seq[i].seq);
392     }
393
394   // Get a prime number for the size of the hash table
395   hashSize = sieve3 ((long) (2 * totalChars));
396   wordHash = initSHash (hashSize);
397
398   // Chop up each sequence and hash out the words of size L
399   for (i = 0; i < numSeq; i++)
400     {
401       prev = -1;
402
403       // skip sequences that are too short to have
404       // a pattern
405       if (strlen (seq[i].seq) < L)
406     {
407       continue;
408     }
409       for (j = 0; j < strlen (seq[i].seq) - L + 1; j++)
410     {
411
412       // Make a hash table entry for this word
413       newEntry.key = &(seq[i].seq[j]);
414       newEntry.data = 1;
415       newEntry.idx = wc;
416       newEntry.L = L;
417
418       // Check to see if it's already in the hash table
419       ep = searchSHash (&newEntry, &wordHash, 0);
420       if (ep == NULL)
421         {
422
423           // If it's not, create an entry for it
424           ep = searchSHash (&newEntry, &wordHash, 1);
425
426           // Increase the size of our word array
427           words = (sPat_t *) realloc (words, (wc + 1) * sizeof (sPat_t));
428           if (words == NULL)
429         {
430           fprintf (stderr, "Error!\n");
431           fflush (stderr);
432         }
433           // Add the new word
```

```
434           words[wc].string = &(seq[i].seq[j]);
435           words[wc].length = L;
436           words[wc].support = 1;
437           words[wc].offset =
438         (sOffset_t *) malloc (1 * sizeof (sOffset_t));
439           if (words[wc].offset == NULL)
440         {
441           fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
442           fflush (stderr);
443           exit (0);
444         }
445           words[wc].offset[0].seq = i;
446           words[wc].offset[0].pos = j;
447           words[wc].offset[0].prev = prev;
448           words[wc].offset[0].next = -1;
449
450           if (prev != -1)
451         {
452           words[prev].offset[words[prev].support - 1].next = wc;
453         }
454           prev = wc;
455           wc++;
456
457         }
458       else
459         {
460
461           // If it is, increase the count for this word
462           ep->data++;
463
464           // add a new offset to the word array
465           l = words[ep->idx].support;
466           words[ep->idx].offset =
467         (sOffset_t *) realloc (words[ep->idx].offset,
468                                (l + 1) * sizeof (sOffset_t));
469           words[ep->idx].offset[l].seq = i;
470           words[ep->idx].offset[l].pos = j;
471           words[ep->idx].offset[l].prev = prev;
472           words[ep->idx].offset[l].next = -1;
473
474           // Update the next/prev
475           if (prev != -1)
476         {
477           words[prev].offset[words[prev].support - 1].next = ep->idx;
478         }
479           prev = ep->idx;
480
481           // Have to put this down here for cases when we create
482           // a word and it is immeadiately followed by itself!!
483           words[ep->idx].support += 1;
484         }
485     }
486     }
487
488
489   destroySHash (&wordHash);
490   *numWords = wc;
491   return words;
492 }
```

C.9.0.139 void getMatrixByName (char *name*[], int *mat*[][MATRIX_SIZE])

Referenced by main().

C.9.0.140 int main (int *argc*, char ** *argv*)

This is the main routine of the Gemoda source code. The routine performs basic operations such as parsing the input from the user and opening input files. Then, the function hashes words of length L. The unique words are aligned against each other to produce an adjacency matrix that says whether the unique word i is sufficiently similar, based on the user supplied threshold, to the unique word j. This adjacency matrix is then dereferenced into an adjacency matrix in which each index of the matrix represents a unique position in the input sequences, rather than a unique word. This dereferencing is required for the convolution stage. Finally, this adjacency matrix is convolved and the final motifs are returned as a linked list. The routine then closes all input and output files and frees up dynamically allocated memory.

Definition at line 187 of file gemoda-s.c.

References alignWordsMat_bit(), bitGraphCheckBit(), bitGraphSetTrueSym(), calcStatAllCliqs(), convolve(), countWords2(), cumDMatrix(), deleteBitGraph(), FreeFSeqs(), getMatrixByName(), getStatMat(), cnode::length, mat, MATRIX_SIZE, matrixlist(), cSet_t::members, newBitGraph(), cnode::next, sPat_t::offset, popAllCll(), sOffset_t::pos, ReadFSeqs(), sOffset_t::seq, cnode::set, cSet_t::size, bitGraph_t::size, sortByStats(), cnode::stat, and usage().

```
188 {
189   int inputOption = 0;
190   char *sequenceFile = NULL;
191   char *outputFile = NULL;
192   char *matName = NULL;
193   FILE * SEQUENCE_FILE = NULL;
194   FILE * OUTPUT_FILE = NULL;
195   int L = 0;
196   int numberOfSequences = 0;
197   fSeq_t * mySequences = NULL;
198   fSeq_t * (*seqReadFunct) () = &ReadFSeqs;
199   sPat_t * words = NULL;
200   int wc;
201   int status = 0;
202   int g = 0;
203   int sup = 2;
204   int R = 1;
205   int P = 0;
206   int (*mat)[MATRIX_SIZE] = NULL;
207   int noConvolve = 0;
208   int j, k, i, l;
209   bitGraph_t * bg = NULL;
210   bitGraph_t * oam = NULL;
211
212     // new
213   int **offsetToIndex = NULL;
214   int *indexToSeq = NULL;
215   int *indexToPos = NULL;
216   int numberOfOffsets = 0;
217   int pos1, pos2;
218
219     // int *prevRowArray;
220     sOffset_t * offset1, *offset2;
221   cll_t * allCliques = NULL;
222   cll_t * curCliq = NULL;
223   int curSeq;
224   int curPos;
```

```
225  int clusterMethod = 0;
226
227    // patStats
228  int samp = 1;
229  unsigned int **d = NULL;
230  int supportDim = 0, lengthDim = 0;
231  int oamSize = 0;
232
233    /*
234        Get command-line options
235     */
236    while ((inputOption = getopt (argc, argv, "i:o:l:g:k:m:p:zc:ns:")) != EOF)
237    {
238      switch (inputOption)
239    {
240
241        // Input file
242    case 'i':
243      sequenceFile = optarg;
244      seqReadFunct = &ReadFSeqs;
245      break;
246
247        // Output file
248    case 'o':
249      outputFile =
250        (char *) malloc ((strlen (optarg) + 1) * sizeof (char));
251      if (outputFile == NULL)
252        {
253          fprintf (stderr, "Error allocating memory for options.\n");
254          exit (EXIT_FAILURE);
255        }
256      else
257        {
258          strcpy (outputFile, optarg);
259        }
260      break;
261
262        // Minimum motif length
263    case 'l':
264      L = atoi (optarg);
265      break;
266
267        // Minimum motif similarity score
268    case 'g':
269      g = atoi (optarg);
270      status++;
271      break;
272
273        // Minimum support (number of motif occurrences)
274    case 'k':
275      sup = atoi (optarg);
276      break;
277
278        // Similarity matrix used to find similarity score
279    case 'm':
280      getMatrixByName (optarg, &mat);
281      matName = (char *) malloc (strlen (optarg) * sizeof (char));
282      if (matName == NULL)
283        {
284          fprintf (stderr, "Error allocating memory for options.\n");
285          exit (EXIT_FAILURE);
286        }
287      else
288        {
289          strcpy (matName, optarg);
290        }
291      break;
292
```

```
293 /*****************************************************************
294  * Recursive initial pruning: an option for clique finding.
295  *   It takes all nodes with less than the minimum
296  *   number of support and removes all of their nodes, and does this
297  *   recursively so that nodes that are connected to many sparsely connected
298  *   nodes will be removed and not left in the
299  * This option is deprecated as it is at worst no-gain and at best useful.
300  *   It will be on by default for clique-finding, but can be turned
301  *   back off with some
302  *   minor tweaking.  For almost all cases in which it does not speed
303  *   up computations, it will have a trivial time to perform.  Thus, if
304  *   clique-finding is turned on, then R is set to 1 by default.
305        case 'r':
306            R = 1;
307            break;
308 *****************************************************************************/
309        // Optional pruning parameter to require at motif occurrences
310        // in at least P distinct input sequences
311     case 'p':
312       P = atoi (optarg);
313       break;
314
315        // Clustering method.
316     case 'c':
317       clusterMethod = atoi (optarg);
318       break;
319     case 'n':
320       noConvolve = 1;
321       break;
322     case 's':
323       samp = atoi (optarg);
324       break;
325
326        // Catch-all.
327     case '?':
328       fprintf (stderr, "Unknown option `-%c'.\n", optopt);
329       usage (argv);
330       return EXIT_SUCCESS;
331     case 'z':
332       matrixlist ();
333       return EXIT_SUCCESS;
334     default:
335       usage (argv);
336       return EXIT_SUCCESS;
337     }
338     }
339
340     // Require a similarity matrix
341     if (mat == NULL)
342     {
343       usage (argv);
344       return EXIT_SUCCESS;
345     }
346
347     // Require an input file, a nonzero length, and a similarity threshold
348     // to be set.
349     if (sequenceFile == NULL || L == 0 || status < 1)
350     {
351       usage (argv);
352       return EXIT_SUCCESS;
353     }
354
355     // Open the sequence file
356     if ((SEQUENCE_FILE = fopen (sequenceFile, "r")) == NULL)
357     {
358       fprintf (stderr, "Couldn't open file %s; %s\n", sequenceFile,
359         strerror (errno));
360       exit (EXIT_FAILURE);
```

```
361     }
362
363     // Open the output file
364     if (outputFile != NULL)
365     {
366       if ((OUTPUT_FILE = fopen (outputFile, "w")) == NULL)
367     {
368       fprintf (stderr, "Couldn't open file %s; %s\n", outputFile,
369             strerror (errno));
370       exit (EXIT_FAILURE);
371     }
372     }
373   else
374     {
375       OUTPUT_FILE = stdout;
376     }
377
378     // Allocate some sequences
379     mySequences = seqReadFunct (SEQUENCE_FILE, &numberOfSequences);
380   if (mySequences == NULL)
381     {
382       fprintf (stderr, "\nError reading your sequences/text.");
383       fprintf (stderr, "\nCheck the format/size of the file.");
384       fprintf (stderr, "\nERROR:  %s\n", strerror (errno));
385       return EXIT_FAILURE;
386     }
387
388     // Close the input files
389     fclose (SEQUENCE_FILE);
390
391     // Verbosity in output helps to distinguish output files.
392     fprintf (OUTPUT_FILE, "\nMatrix used = %s\n", matName);
393   fprintf (OUTPUT_FILE, "Input file = %s\n", sequenceFile);
394   fprintf (OUTPUT_FILE, "l = %d, k = %d, g = %d\n", L, sup, g);
395   if (P > 1)
396     {
397       fprintf (OUTPUT_FILE, "Minimum # of sequences with motif = %d\n", P);
398     }
399   if (R > 0)
400     {
401       fprintf (OUTPUT_FILE, "Recursive pruning is ON.\n");
402     }
403
404     // Find the unique words in the input.
405     words = countWords2 (mySequences, numberOfSequences, L, &wc);
406
407     /*
408        fprintf(stderr, "Counted %d words\n", wc);
409      */
410     /*
411        fflush(stderr);
412      */
413
414     // Align the words that we just found by applying the similarity
415     // matrix to each pair of them.  Note that
416     // bg is the adjacency matrix of words, but we
417     // need an adjacency matrix of offsets instead.
418     bg = alignWordsMat_bit (words, wc, mat, g);
419   fprintf (OUTPUT_FILE, "\nAligned!  Creating offset matrix...\n");
420   fflush (NULL);
421
422     // Create an intermediate translation matrix
423     // to store the offset number of each sequence number/position.
424     //
425     // Note that this matrix is better called "Index to offset", and
426     // the other matrices are better called "offset to Seq" and
427     // "offset to Pos"
428     offsetToIndex = (int **) malloc (numberOfSequences * sizeof (int *));
```

```
429   if (offsetToIndex == NULL)
430     {
431       fprintf (stderr,
432         "Unable to allocate memory - offsetToIndex in gemoda.c\n%s\n",
433         strerror (errno));
434       fflush (stderr);
435       exit (0);
436     }
437   for (i = 0; i < numberOfSequences; i++)
438     {
439
440     // MPS 5/23/05: Added in "-L+2" to make there only be one
441     // blank between sequences.
442     offsetToIndex[i] =
443     malloc ((strlen (mySequences[i].seq) - L + 2) * sizeof (int));
444       if (offsetToIndex[i] == NULL)
445     {
446       fprintf (stderr,
447             "Unable to allocate memory - offsetToIndex[%d] in gemoda.c\n%s\n",
448             i, strerror (errno));
449       fflush (stderr);
450       exit (0);
451     }
452
453     // MPS 5/23/05: Added in "-L+2" to make there only be one
454     // blank between sequences.
455     for (j = 0; j < (strlen (mySequences[i].seq) - L + 2); j++)
456     {
457       offsetToIndex[i][j] = numberOfOffsets;
458       numberOfOffsets++;
459     }
460     }
461
462     // Now create translation matrices such that we can get the sequence
463     // or position number of a given offset.
464     indexToSeq = (int *) malloc (numberOfOffsets * sizeof (int));
465   if (indexToSeq == NULL)
466     {
467       fprintf (stderr,
468         "Unable to allocate memory - indexToSeq in gemoda.c\n%s\n",
469         strerror (errno));
470       fflush (stderr);
471       exit (0);
472     }
473   indexToPos = (int *) malloc (numberOfOffsets * sizeof (int));
474   if (indexToPos == NULL)
475     {
476       fprintf (stderr,
477         "Unable to allocate memory - indexToPos in gemoda.c\n%s\n",
478         strerror (errno));
479       fflush (stderr);
480       exit (0);
481     }
482   k = 0;
483   for (i = 0; i < numberOfSequences; i++)
484     {
485
486     // MPS 5/23/05: Added in "-L+2" to make there only be one
487     // blank between sequences.
488     for (j = 0; j < (strlen (mySequences[i].seq) - L + 2); j++)
489     {
490       indexToSeq[k] = i;
491       indexToPos[k] = j;
492       k++;
493     }
494     }
495
496     // Now make an offset adjacency matrix!
```

```
497     //
498     oam = newBitGraph (numberOfOffsets);
499
500     // Go through each unique word
501     for (i = 0; i < wc; i++)
502     {
503       offset1 = words[i].offset;
504
505     // Go through each occurrence
506     for (k = 0; k < words[i].support; k++)
507     {
508
509         // Use the offsetToIndex translation to get the offset
510         // of the first occurrence
511         pos1 = offsetToIndex[offset1[k].seq][offset1[k].pos];
512
513         // And go through each word in the first offset to
514         // find words that meet the similarity threshold
515         for (j = 0; j < wc; j++)
516         {
517           if (bitGraphCheckBit (bg, i, j))
518         {
519           offset2 = words[j].offset;
520
521             // And find all of their occurrences,
522             // using offsetToIndex to get the
523             // offsets, and then setting those
524             // locations in the offset adjacency
525             // matrix true.
526             for (l = 0; l < words[j].support; l++)
527             {
528               pos2 = offsetToIndex[offset2[l].seq][offset2[l].pos];
529               bitGraphSetTrueSym (oam, pos1, pos2);
530             }
531         }
532         }
533     }
534     }
535   fprintf (OUTPUT_FILE, "Offset matrix created...");
536   deleteBitGraph (bg);
537   if ((samp > 0) && (clusterMethod == 0))
538     {
539       fprintf (OUTPUT_FILE, " taking preliminary statistics.\n");
540       fflush (NULL);
541       d =
542     getStatMat (oam, sup, L, &supportDim, &lengthDim, numberOfSequences,
543             samp, OUTPUT_FILE);
544       fprintf (OUTPUT_FILE, "Now filtering...\n");
545       fflush (NULL);
546     }
547   else
548     {
549       fprintf (OUTPUT_FILE, " now filtering.\n");
550       fflush (NULL);
551       d = NULL;
552       supportDim = 0;
553     }
554
555     // Now we're convolving on offsets
556     allCliques =
557     convolve (oam, sup, R, indexToSeq, P, clusterMethod, offsetToIndex,
558           numberOfSequences, noConvolve, OUTPUT_FILE);
559
560     // Do some early memory cleanup to limit usage
561     oamSize = oam->size;
562   deleteBitGraph (oam);
563   fprintf (OUTPUT_FILE, "Convolved!  Now making output...\n");
564   fflush (NULL);
```

```
565   if ((samp > 0) && (clusterMethod == 0))
566     {
567       cumDMatrix (d, allCliques, supportDim, lengthDim, oamSize,
568           numberOfSequences);
569       calcStatAllCliqs (d, allCliques, numberOfOffsets - numberOfSequences);
570       allCliques = sortByStats (allCliques);
571     }
572
573     // walk over the cliques and give some output in the format:
574     // pattern <pattern id num>: len=<motif length> sup=<motif instances>
575     // <sequence num> <position num> <motif instance>
576     // ...
577     curCliq = allCliques;
578
579     i = 0;
580   while (curCliq != NULL)
581     {
582       fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d", i,
583         curCliq->length + L, curCliq->set->size);
584       if (d != NULL)
585     {
586       fprintf (OUTPUT_FILE, "\tsignif=%le\n", curCliq->stat);
587     }
588       else
589     {
590       fprintf (OUTPUT_FILE, "\n");
591     }
592
593     for (j = 0; j < curCliq->set->size; j++)
594     {
595       pos1 = curCliq->set->members[j];
596       curSeq = indexToSeq[pos1];
597       curPos = indexToPos[pos1];
598       fprintf (OUTPUT_FILE, "   %d\t%d\t", curSeq, curPos);
599       for (k = curPos; k < curPos + curCliq->length + L; k++)
600         {
601           fprintf (OUTPUT_FILE, "%c", mySequences[curSeq].seq[k]);
602         }
603       fprintf (OUTPUT_FILE, "\n");
604     }
605       fprintf (OUTPUT_FILE, "\n\n");
606       curCliq = curCliq->next;
607       i++;
608     }
609
610     // And do some memory cleanup
611     // And cleanup of probability stuff...
612     /*
613         free(letterfreqs); delete_augmented_matrix(augmat);
614      */
615     allCliques = popAllCll (allCliques);
616   free (indexToSeq);
617   indexToSeq = NULL;
618   free (indexToPos);
619   indexToPos = NULL;
620   for (i = 0; i < numberOfSequences; i++)
621     {
622       free (offsetToIndex[i]);
623       offsetToIndex[i] = NULL;
624     }
625
626     // Free'ing added by MPS, 6/4
627     for (i = 0; i < wc; i++)
628     {
629       free (words[i].offset);
630     }
631   free (words);
632
```

```
633     // End free'ing added by MPS
634     free (offsetToIndex);
635   offsetToIndex = NULL;
636
637     // ----------------------------------------
638
639     // Free up fastaSequences
640     FreeFSeqs (mySequences, numberOfSequences);
641   fclose (OUTPUT_FILE);
642   return 0;
643 }
```

C.9.0.141 void matrixlist (void)

This function prints a list of the matrices that Gemoda can use to do the alignment of words. Most of these matrices are appropriate for amino acid sequences. In addition, there are matrices for DNA sequences and an identity matrix that is appropriate for other sequences, such as the analysis of English text. The matrix is selected using the -m flag.

Definition at line 99 of file gemoda-s.c.

Referenced by main().

```
100 {
101   fprintf (stdout, "\nThe following similarity matrices are installed "
102         "with the default Gemoda installation.\n  Most of these "
103         "were obtained from publically available BLAST distributions. \n\n"
104          "dna_idmat:\n\t"
105         "Identity matrix for DNA: returns 1 when A,C,G,T are "
106         "compared to \n\tthemselves, 0 otherwise.\n\n"
107         "identity_aa:\n\t"
108         "Identity matrix for amino acids: returns 1 when any \n\t"
109         "letter but J,O,U are compared to themselves, and 0 "
110         "otherwise.\n\n"  "idmat:\n\t"
111         "Similar to identity_aa, but it returns 10 in place "
112         "of 1.\n\n"  "est_idmat:\n\t"
113         "Similar to idmat, but it returns -10 in place of 0. "  "\n\n"
114         "pam100:\n"  "pam110:\n"  "pam120:\n"  "pam130:\n"
115         "pam140:\n"  "pam150:\n"  "pam160:\n"  "pam190:\n"
116         "pam200:\n"  "pam210:\n"  "pam220:\n"  "pam230:\n"
117         "pam240:\n"  "pam250:\n"  "pam260:\n"  "pam280:\n"
118         "pam290:\n"  "pam300:\n"  "pam310:\n"  "pam320:\n"
119         "pam330:\n"  "pam340:\n"  "pam360:\n"  "pam370:\n"
120         "pam380:\n"  "pam390:\n"  "pam400:\n"  "pam430:\n"
121         "pam440:\n"  "pam450:\n"  "pam460:\n"  "pam490:\n"
122         "pam500:\n\t"
123         "PAM matrices for various evolutionary distances.\n\n"
124         "blosum30:\n"  "blosum35:\n"  "blosum40:\n"  "blosum45:\n"
125         "blosum50:\n"  "blosum55:\n"  "blosum60:\n"  "blosum62:\n"
126         "blosum65:\n"  "blosum70:\n"  "blosum75:\n"  "blosum80:\n"
127         "blosum85:\n"  "blosum90:\n"  "blosum100:\n\t"
128         "BLOSUM matrices for various evolutionary distances.\n\n"
129         "blosumn:\n\t"  "BLOSUM matrix of unknown origin.\n\n"
130         "dayhoff:\n\t"
131         "'Vanilla-flavored' pam250, very similar to pam250.\n\n"
132         "phat_t75_b73:\n"  "phat_t80_b78:\n"  "phat_t85_b82:\n\t"
133         "BLOSUM-clustered scoring matrix with target frequency\n\t"
134         "PHDhtm clustering = {75,80,85}percent and background frequency\n\t"
135          "Persson-Argos clustering = {73,78,82}percent.\n\t"
136         "From Ng, Henikoff, & Henikoff, Bioinformatics 16: 760.\n\n"
137         "coil_mat:\n"  "alpha_mat:\n"  "beta_mat:\n\t"
```

```
138         "Three structure-specific matrices described by Luthy,\n\t"
139         "McLachlan, and Eisenberg in Proteins 10, 229-239, obtained from AAindex.\n\n");
140   fprintf (stdout, "\n");
141 }
```

C.9.0.142 bitGraph_t* pruneBitGraph (bitGraph_t * *bg*, int * *indexToSeq*, int ** *offsetToIndex*, int *numOfSeqs*, int *p*)

Simple function (non-recursive) to prune off the first level of motifs that will not meet the "minimum number of unique sequences" criterion. This could have been implemented as above, but it may have gotten a little expensive with less yield, so only the first run through is done here. Input: a bit graph to be pruned, a pointer to the structure that dereferences offset indices to sequence numbers, a pointer to the structure that dereferences seq/position to offsets, the number of unique sequences in the input set, and the minimum number of unique sequences that must contain the motif. Output: a pruned bitGraph.

Definition at line 402 of file newConv.c.

References emptySet(), bitGraph_t::graph, and nextBitBitSet().

```
404 {
405   int i = 0, j = 0, nextBit = 0;
406   int *seqNums = NULL;
407
408     // Since we don't immediately know which node is in which source
409     // sequence, we can't just count them up regularly.  Instead, we'll
410     // need to keep track of which sequences they come from and
411     // increment _something_.  What we chose to do here is just make
412     // an array of integers of length = <p>.  Then, we try to put the
413     // source sequence number of each neighbor (including itself, since
414     // the main diagonal is still true at this time) into the next slot
415     // Since we will monotonically search the bitSet, we can just
416     // move on to the first bit in the next sequence using the
417     // offsetToIndex structure so that we know the next sequence number
418     // to be put in is always unique.
419     seqNums = (int *) malloc (p * sizeof (int));
420   if (seqNums == NULL)
421     {
422       fprintf (stderr, "Memory error - pruneBitGraph\n%s\n",
423         strerror (errno));
424       fflush (stderr);
425       exit (0);
426     }
427
428     // So, for each row in the bitgraph...
429     for (i = 0; i < bg->size; i++)
430     {
431
432     // Make sure the whole array is -1 sentinels.
433     for (j = 0; j < p; j++)
434     {
435       seqNums[j] = -1;
436     }
437       j = 0;
438
439     // Find the first neighbor of this bit.
440     nextBit = nextBitBitSet (bg->graph[i], 0);
441       if (nextBit == -1)
442     {
```

```
443       continue;
444     }
445       else
446     {
447
448         // and put its sequence number in the array of ints.
449         seqNums[0] = indexToSeq[nextBit];
450     }
451
452     // If it's the last sequence, then bail out so that we don't
453     // segfault in the next step.
454     if (seqNums[0] >= numOfSeqs - 1)
455     {
456       emptySet (bg->graph[i]);
457       continue;
458     }
459
460     // Find the next neighbor of this bit, STARTING AT the first
461     // bit in the next sequence.
462     nextBit =
463     nextBitBitSet (bg->graph[i],
464                offsetToIndex[indexToSeq[nextBit] + 1][0]);
465
466     // And iterate this until we run out of neighbors.
467     while (nextBit >= 0)
468     {
469       j++;
470       seqNums[j] = indexToSeq[nextBit];
471
472         // Or until this new neighbor will fill up the array
473         if (j == p - 1)
474         {
475           break;
476         }
477
478         // Or until this new neighbor is in the last sequence.
479         if (seqNums[j] >= numOfSeqs - 1)
480         {
481           break;
482         }
483
484         // Get the next neighbor!
485         nextBit =
486         nextBitBitSet (bg->graph[i],
487                offsetToIndex[indexToSeq[nextBit] + 1][0]);
488     }
489
490     // If we didn't have enough unique sequences, and either a) we
491     // were in the nth-to-last sequence and there were no
492     // neighbors after it, or b) we were in the last sequence,
493     // then the last number will still be our sentinel, -1.  If
494     // the last number is not a sentinel, then we have at least
495     // p distinct sequence occurrences, so we're OK.
496     if (seqNums[p - 1] == -1)
497     {
498       emptySet (bg->graph[i]);
499     }
500     }
501   free (seqNums);
502   return (bg);
503 }
```

C.9.0.143 void usage (char ** *argv*)

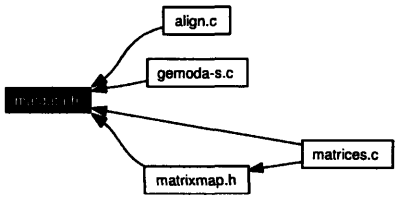
This function describes the basic usage of Gemoda. It is invoked whenever the user submits poor input parameters or selects the help parameter. The function prints a list of possible parameters for Gemoda.

Definition at line 32 of file gemoda-s.c.

```
33 {
34   fprintf (stdout, "Usage: %s -i <Fasta sequence file> "
35         "-l <word size> \n\t-k <support> -g <threshold>"
36         "-m <matrix name> [-z] \n\t[-c <cluster method [0|1]>]"
37         "[-p <unique support>] \n\n\n"
38          "Required flags and input:\n\n"
39         "-i <Fasta sequence file>:\n\t"
40         "File containing all sequences to be searched, in Fasta format.\n\n"
41          "-l <word size>:\n\t"
42         "Minimum length of motifs; also the sliding window length\n\t"
43         "over which all motifs must meet the similarity criterion\n\n"
44         "-k <support>:\n\t"  "Minimum number of motif occurrences.\n\n"
45         "-g <threshold>:\n\t"
46         "Similarity threshold.  Two windows, when scored with the\n\t"
47         " similarity matrix defined by the -m flag, must have at least\n\t"
48
49         " this score in order to be deemed 'connected'.  This criterion\n\t"
50          " must be met over all sliding windows of length 1.\n\n"
51         "-m <matrix name>:\n\t"
52         "Name of the similarity matrix to be used to compare windows.\n\t"
53          "Use -z to see a list of matrices installed by default.\n\n\n"
54         "Optional flags and input:\n\n"  "-z:\n\t"
55         "Lists all of the similarity matrices available with the\n\t"
56         "initial installation of Gemoda.  Note that this overrides\n\t"
57         "all other options and will only give this output.\n\n"
58         "-c <cluster method [0|1]>:\n\t"
59         "The clustering method to be used after evaluating the "
60         "\n\tsimilarity of the unique words in the input.  Note that the "
61
62         "\n\tclustering method will have a significant impact on both the "
63          "\n\tresults that one obtains and the computation time.\n\n\t"
64         "0: clique-finding\n\t\t"
65         "Uses established methods to find all maximal cliques in the "
66         "\n\t\tdata.  This will give the most thorough results (that are "
67
68         "\n\t\tprovably exhaustive), but will also give less-significant "
69          "\n\t\tresults in addition to the most interesting and most\n\t"
70
71         "significant ones.  The results are deterministic but may take some "
72
73         "\n\t\ttime on data sets with high similarity or if the similarity "
74          "\n\t\tthreshold is set extremely low.\n\t"
75         "1: single-linkage clustering\n\t\t"
76         "Uses a single-linkage-type clustering where all nodes that "
77         "\n\t\tare connected are put in the same cluster.  This method is "
78
79         "\n\t\talso deterministic and will be faster than clique-finding, "
80
81         "\n\t\tbut it loses guarantees of exhaustiveness in searching the "
82          "\n\t\tdata set.\n\n"  "-p <unique support>:\n\t"
83         "A pruning parameter that requires the motif to occur in "
84         "\n\tat least <unique support> different input sequences.  Note "
85
86         "\n\tthat this parameter must be less than or equal to the total "
87          "\n\tsupport parameter set by the -k flag.\n\n", argv[0]);
88   fprintf (stdout, "\n");
89 }
```

C.10 matdata.h File Reference

This graph shows which files directly or indirectly include this file:



Defines

- #define MATRIX_SIZE 23

Detailed Description

This file defines the size of the scoring matrices so that we don't have to pound-include the whole matrices.h file due to worries about incompatibilities with earlier extern variable declarations.

Definition in file matdata.h.

Define Documentation

C.10.0.144 #define MATRIX_SIZE 23

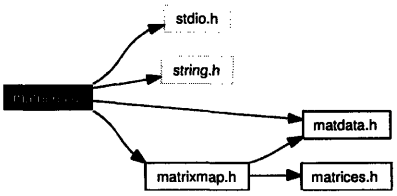
Definition at line 10 of file matdata.h.

Referenced by main().

C.11 matrices.c File Reference

```
#include <stdio.h>
#include <string.h>
#include "matdata.h"
#include "matrixmap.h"
```

Include dependency graph for matrices.c:



Defines

- #define DEFAULT_MATRIX blosum62

Functions

- void getMatrixByName (char name[], const int(**matp)[MATRIX_SIZE])

Detailed Description

This file contains functions for handling scoring matrices used for the sequence based Gemoda.

Definition in file matrices.c.

Define Documentation

C.11.0.145 #define DEFAULT_MATRIX blosum62

Definition at line 7 of file matrices.c.

Referenced by getMatrixByName().

Function Documentation

C.11.0.146 void getMatrixByName (char *name*[], const int ** *matp*[MATRIX_SIZE])

A simple function to take the matrix name argument given as input to gemoda and return the physical memory location of that matrix by using the matrix_map construct. Input: a string containing the matrix name a pointer to a two-dimensional array. Output: None, though the value of the pointer given as input is changed to reflect the location of the matrix

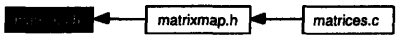
Definition at line 34 of file matrices.c.

References DEFAULT_MATRIX, and matrix_map.

```
35 {
36   int i;
37   for (i = 0; matrix_map[i].name != NULL; i++)
38     {
39       if (strcmp (name, matrix_map[i].name) == 0)
40     {
41       break;
42     }
43     }
44   if (matrix_map[i].name != NULL)
45     {
46       *matp = (matrix_map[i].mat);
47     }
48   else
49     {
50       *matp = (DEFAULT_MATRIX);
51     }
52 }
```

C.12 matrices.h File Reference

This graph shows which files directly or indirectly include this file:



Variables

- const int aaOrder []
- const int dna_idmat [MATRIX_SIZE][MATRIX_SIZE]
- const int identity_aa [MATRIX_SIZE][MATRIX_SIZE]
- const int idmat [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum100 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum30 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum35 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum40 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum45 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum50 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum55 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum60 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum62 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum65 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum70 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum75 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum80 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum85 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum90 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosumn [MATRIX_SIZE][MATRIX_SIZE]
- const int dayhoff [MATRIX_SIZE][MATRIX_SIZE]
- const int pam100 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam110 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam120 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam130 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam140 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam150 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam160 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam190 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam200 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam210 [MATRIX_SIZE][MATRIX_SIZE]

- const int pam220 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam230 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam240 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam250 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam260 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam280 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam290 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam300 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam310 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam320 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam330 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam340 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam360 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam370 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam380 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam390 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam400 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam430 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam440 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam450 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam460 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam490 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam500 [MATRIX_SIZE][MATRIX_SIZE]
- const int phat_t75_b73 [MATRIX_SIZE][MATRIX_SIZE]
- const int phat_t80_b78 [MATRIX_SIZE][MATRIX_SIZE]
- const int phat_t85_b82 [MATRIX_SIZE][MATRIX_SIZE]
- const int alpha_mat [MATRIX_SIZE][MATRIX_SIZE]
- const int beta_mat [MATRIX_SIZE][MATRIX_SIZE]
- const int coil_mat [MATRIX_SIZE][MATRIX_SIZE]

Detailed Description

This file contains a number of scoring matrices, most of which are intended for comparing amino acid sequences; however a few are for DNA. In general, if a user wants to add their own matrix for use with Gemoda, they should add it to this file and recompile Gemoda.

Note that users are not restricted to 23x23 matrices. By changing aaOrder, you can easily make matrices for comparing ANSII strings with up to 256 different characters.

All of the matrices below were obtained directly from BLAST/WU-BLAST; they are all also part of the public domain, so there is nothing intrinsic to BLAST with respect to the matrices. It was just the easiest way to get all of the matrices into our software.

The most popular matrix for amino acid sequences is blosum62.

A good location for getting new scoring matrices, such as those based on structural data, is the AAIndex. URLs tend to change, so rather than us listing it here, Google it!

Definition in file matrices.h.

Variable Documentation

C.12.0.147 const int aaOrder[]

Definition at line 32 of file matrices.h.

Referenced by alignMat().

C.12.0.148 const int alpha_mat[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1398 of file matrices.h.

C.12.0.149 const int beta_mat[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1422 of file matrices.h.

C.12.0.150 const int blosum100[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 126 of file matrices.h.

C.12.0.151 const int blosum30[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 150 of file matrices.h.

C.12.0.152 const int blosum35[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 174 of file matrices.h.

C.12.0.153 const int blosum40[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 198 of file matrices.h.

C.12.0.154 const int blosum45[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 222 of file matrices.h.

C.12.0.155 const int blosum50[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 246 of file matrices.h.

C.12.0.156 const int blosum55[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 270 of file matrices.h.

C.12.0.157 const int blosum60[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 294 of file matrices.h.

C.12.0.158 const int blosum62[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 318 of file matrices.h.

C.12.0.159 const int blosum65[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 342 of file matrices.h.

C.12.0.160 const int blosum70[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 366 of file matrices.h.

C.12.0.161 const int blosum75[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 390 of file matrices.h.

C.12.0.162 const int blosum80[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 414 of file matrices.h.

C.12.0.163 const int blosum85[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 438 of file matrices.h.

C.12.0.164 const int blosum90[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 462 of file matrices.h.

C.12.0.165 const int blosumn[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 486 of file matrices.h.

C.12.0.166 const int coil_mat[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1446 of file matrices.h.

C.12.0.167 const int dayhoff[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 510 of file matrices.h.

C.12.0.168 const int dna_idmat[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 50 of file matrices.h.

C.12.0.169 const int identity_aa[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 76 of file matrices.h.

C.12.0.170 const int idmat[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 101 of file matrices.h.

C.12.0.171 const int pam100[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 534 of file matrices.h.

C.12.0.172 const int pam110[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 558 of file matrices.h.

C.12.0.173 const int pam120[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 582 of file matrices.h.

C.12.0.174 const int pam130[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 606 of file matrices.h.

C.12.0.175 const int pam140[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 630 of file matrices.h.

C.12.0.176 const int pam150[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 654 of file matrices.h.

C.12.0.177 const int pam160[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 678 of file matrices.h.

C.12.0.178 const int pam190[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 702 of file matrices.h.

C.12.0.179 const int pam200[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 726 of file matrices.h.

C.12.0.180 const int pam210[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 750 of file matrices.h.

C.12.0.181 const int pam220[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 774 of file matrices.h.

C.12.0.182 const int pam230[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 798 of file matrices.h.

C.12.0.183 const int pam240[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 822 of file matrices.h.

C.12.0.184 const int pam250[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 846 of file matrices.h.

C.12.0.185 const int pam260[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 870 of file matrices.h.

C.12.0.186 const int pam280[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 894 of file matrices.h.

C.12.0.187 const int pam290[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 918 of file matrices.h.

C.12.0.188 const int pam300[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 942 of file matrices.h.

C.12.0.189 const int pam310[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 966 of file matrices.h.

C.12.0.190 const int pam320[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 990 of file matrices.h.

C.12.0.191 const int pam330[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1014 of file matrices.h.

C.12.0.192 const int pam340[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1038 of file matrices.h.

C.12.0.193 const int pam360[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1062 of file matrices.h.

C.12.0.194 const int pam370[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1086 of file matrices.h.

C.12.0.195 const int pam380[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1110 of file matrices.h.

C.12.0.196 const int pam390[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1134 of file matrices.h.

C.12.0.197 const int pam400[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1158 of file matrices.h.

C.12.0.198 const int pam430[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1182 of file matrices.h.

C.12.0.199 const int pam440[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1206 of file matrices.h.

C.12.0.200 const int pam450[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1230 of file matrices.h.

C.12.0.201 const int pam460[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1254 of file matrices.h.

C.12.0.202 const int pam490[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1278 of file matrices.h.

C.12.0.203 const int pam500[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1302 of file matrices.h.

C.12.0.204 const int phat_t75_b73[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1326 of file matrices.h.

C.12.0.205 const int phat_t80_b78[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1350 of file matrices.h.

C.12.0.206 const int phat_t85_b82[MATRIX_SIZE][MATRIX_SIZE]

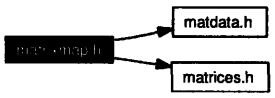
Definition at line 1374 of file matrices.h.

C.13 matrixmap.h File Reference

```
#include "matdata.h"
```

```
#include "matrices.h"
```

Include dependency graph for matrixmap.h:



This graph shows which files directly or indirectly include this file:

Variables

- struct {
  char * name
  const int(* mat)[MATRIX_SIZE]
  } matrix_map []

Detailed Description

This file contains structures and functions for handling scoring matrices.

Definition in file matrixmap.h.

Variable Documentation

C.13.0.207 const int(* mat)[MATRIX_SIZE]

Definition at line 15 of file matrixmap.h.

Referenced by alignMat(), alignWordsMat_bit(), and main().

C.13.0.208 struct { ... } matrix_map[]

This data structure maps the names of common matrices to the names of their variables

Referenced by getMatrixByName().

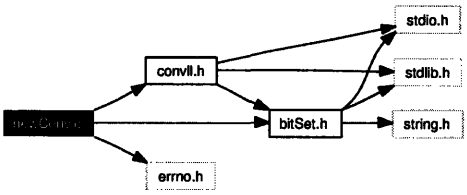
C.13.0.209 char* name

Definition at line 14 of file matrixmap.h.

C.14 newConv.c File Reference

```
#include "bitSet.h"
#include <errno.h>
#include "convll.h"
```

Include dependency graph for newConv.c:



Functions

- int findCliques (bitSet_t *Q, bitSet_t *cand, bitSet_t *mask, bitGraph_t *oG, int support, int qCount, cll_t **elemPats, int *indexToSeq, int p)
- int singleLinkage (bitSet_t *Q, bitSet_t *cand, bitSet_t *mask, bitGraph_t *oG, int support, int qCount, cll_t **elemPats, int *indexToSeq, int p)
- int filterIter (bitGraph_t *graph, int support, bitSet_t *changed, bitSet_t *work)
- int filterGraph (bitGraph_t *graph, int support, int R)
- bitGraph_t * pruneBitGraph (bitGraph_t *bg, int *indexToSeq, int **offsetToIndex, int numOfSeqs, int p)
- cll_t * pruneCll (cll_t *head, int *indexToSeq, int p)
- cll_t * convolve (bitGraph_t *bg, int support, int R, int *indexToSeq, int p, int clusterMethod, int **offsetToIndex, int numberOfSequences, int noConvolve, FILE *OUTPUT_FILE)

Detailed Description

This file contains the core functions that performed the convolution in the Gemoda algorithm. As well, there are two clustering functions defined in this file: one for single linkage clustering, and one for clique based clustering.

Definition in file newConv.c.

Function Documentation

C.14.0.210 cll_t* convolve (bitGraph_t * *bg*, int *support*, int *R*, int * *indexToSeq*, int *p*, int *clusterMethod*, int ** *offsetToIndex*, int *numberOfSequences*, int *noConvolve*, FILE * *OUTPUT_FILE*)

Our outer convolution function. This function will call preliminary functions, cluster the data, and then call the main convolution function. This is the interface between the main gemoda-<x> code and the generic code that gets all of the work done. Input: the bitGraph to be clustered and convolved, the minimum support necessary for a motif to be returned, a flag indicating whether recursive filtering should be used, a pointer to the data structure that dereferences offset indices to sequence numbers, the number of unique source sequences that a motif must be present in, and a number indicating the clustering method that is to be used. Output: the final motif linked list with all motifs that are to be given as output to the user.

Definition at line 625 of file newConv.c.

References bitGraphSetFalseDiagonal(), completeConv(), deleteBitSet(), fillSet(), filterGraph(), findCliques(), newBitSet(), pruneBitGraph(), pruneCll(), singleLinkage(), bitGraph_t::size, and yankCll().

```
629 {
630   bitSet_t * cand = NULL;
631   bitSet_t * mask = NULL;
632   bitSet_t * Q = NULL;
633   int size = bg->size;
634   cll_t * elemPats = NULL;
635   cll_t * allCliques = NULL;
636   cll_t * curr = NULL;
637
638     // contains indices (rows) containing the threshold value.
639     cand = newBitSet (size);
640   mask = newBitSet (size);
641   Q = newBitSet (size);
642   fillSet (cand);
643   fillSet (mask);
644
645     // Note that we prune based on p before setting the diagonal false.
646     if (p > 1)
647     {
648       bg =
649     pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
650     }
651
652     // Now we set the main diagonal false for clustering and filtering.
653     bitGraphSetFalseDiagonal (bg);
654   filterGraph (bg, support, R);
655   fprintf (OUTPUT_FILE, "Graph filtered!  Now clustering...\n");
656   fflush (NULL);
657   if (clusterMethod == 0)
658     {
659       findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
660     }
661   else
662     {
663       singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
```

```
664               p);
665     }
666   fprintf (OUTPUT_FILE,
667         "Clusters found!  Now filtering clusters (if option set)...\n");
668   fflush (NULL);
669   if (p > 1)
670     {
671       elemPats = pruneCll (elemPats, indexToSeq, p);
672     }
673   deleteBitSet (cand);
674   deleteBitSet (mask);
675   deleteBitSet (Q);
676
677     // Now let's convolve what we made.
678     if (noConvolve == 0)
679     {
680       fprintf (OUTPUT_FILE, "Now convolving...\n");
681       fflush (NULL);
682       allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
683     }
684
685   else
686     {
687       curr = elemPats;
688       while (curr != NULL)
689     {
690       yankCll (&elemPats, NULL, &curr, &allCliques, 0);
691     }
692     }
693   return allCliques;
694 }
```

C.14.0.211 int filterGraph (bitGraph_t * *graph*, int *support*, int *R*)

Function to "filter" the initial bitGraph that is being clustered. "Filtering" is the process of removing all nodes from the graph that cannot possibly be in motifs because they are not connected to enough other nodes. This can be done once (if R != 1), or it can be done recursively (if R == 1). When done recursively, it takes the just-filtered graph and checks all of the nodes that the recently removed node used to be connected to; since they have changed in connectivity, they may no longer be connected to enough nodes to be a member of a motif. This is iterated until convergence. Note that the default is to have recursive filtering on, as it ought to decrease the computational complexity of the clustering step and ought not have much of a computational footprint... in cases where it takes a while, it is probably having a good impact in the clustering step, whereas if it is not effective, it probably won't take that long anyway. Input: a bitGraph to be filtered, the minimum support that a motif must have, and the flag indicating recursive filtering or not. Output: Integer success value of 0 (and an altered bitGraph so that all nodes with connections have at least <min support>=""> connections).

Definition at line 359 of file newConv.c.

References copySet(), countSet(), deleteBitSet(), emptySet(), filterIter(), newBitSet(), and bitGraph_t::size.

Referenced by convolve().

```
360 {
361   bitSet_t * changed = newBitSet (graph->size);
362   bitSet_t * work = newBitSet (graph->size);
363   emptySet (changed);
364   emptySet (work);
365
366     // Iteratively call the filtering by copying the previous "work" into
367     // "changed" after each iteration step.
368     if (R == 1)
369     {
370
371       do
372     {
373       filterIter (graph, support, changed, work);
374       copySet (work, changed);
375     }
376       while (countSet (changed) > 0);
377     }
378   else
379     {
380
381     // Otherwise, just do it once.
382     filterIter (graph, support, changed, work);
383     }
384   deleteBitSet (changed);
385   deleteBitSet (work);
386   return 0;
387 }
```

C.14.0.212 int filterIter (bitGraph_t * *graph*, int *support*, bitSet_t * *changed*, bitSet_t * *work*)

The iterator used to "filter" the graph. It takes information in the bitset telling which nodes' rows have changed and only checks them... this should make it pretty efficient time-wise at only a small memory cost. Note the convention that the first time this is called, the changed bitSet is empty... and that the master function is responsible for catching the signal that no changes were made in the last iteration. Input: the bitGraph to be filtered, the minimum support required for a motif to be returned, a bitSet with nodes changed from the previous iteration, and a bitSet to export the nodes changed in this iteration. Output: integer success value of 0 (and also a filtered bitGraph and a bitSet with the nodes changed in this iteration).

Definition at line 228 of file newConv.c.

References countSet(), emptySet(), bitGraph_t::graph, nextBitBitSet(), setFalse(), and setTrue().

Referenced by filterGraph().

```
230 {
231   int i = 0, j = 0;
232   int lastBit = 0, nextBit = 0, lastRow = 0, nextRow = 0;
233   int numNodes = 0;
234   int changedSize = countSet (changed);
235   emptySet (work);
236
237     // Note the convention that the first time the function is called,
238     // it is done with an empty "changed" bitSet as a sentinel.  It is
239     // the responsibility of the master function calling the iterator
```

```
240     // to catch future empty changed sets to know that convergence has
241     // been achieved.
242     //
243     // So, if it's your first time through, go through each node and make
244     // sure that each is connected to at least <support> - 1 others.
245     if (changedSize == 0)
246     {
247       for (i = 0; i < graph->size; i++)
248     {
249       numNodes = countSet (graph->graph[i]);
250       if (numNodes >= support - 1)
251         {
252           continue;
253         }
254       else
255         {
256
257         // Otherwise, zero it out, but going one by
258         // one so that you can also zero out the
259         // symmetric bit.
260         lastBit = 0;
261           for (j = 0; j < numNodes; j++)
262         {
263           nextBit = nextBitBitSet (graph->graph[i], lastBit);
264           if (nextBit == -1)
265             {
266               fprintf (stderr,
267                 "\nEnd of bitSet reached! - initial\n");
268               fflush (stderr);
269               exit (0);
270             }
271           setFalse (graph->graph[i], nextBit);
272           setFalse (graph->graph[nextBit], i);
273
274             // And set that corresponding bit true
275             // in the work bitSet so that we
276             // know we changed it for the next
277             // round.
278             setTrue (work, nextBit);
279           lastBit = nextBit + 1;
280         }
281         }
282     }
283     }
284   else
285     {
286
287     // Otherwise, we've been here before, so just follow what
288     // the changed bitSet says to do... only those bitSets that
289     // were changed could possibly have gone under the minimum
290     // support requirement.
291     lastRow = 0;
292       for (i = 0; i < changedSize; i++)
293     {
294       nextRow = nextBitBitSet (changed, lastRow);
295       if (nextRow == -1)
296         {
297           fprintf (stderr, "\nEnd of bitSet reached! - iter,row\n");
298           fflush (stderr);
299           exit (0);
300         }
301
302         // So now we've found the row that needs to be checked.
303         // We do the same thing we did above... either move
304         // on if it has enough possible support, or zero
305         // it out (with its symmetric locations) one by one.
306         numNodes = countSet (graph->graph[nextRow]);
307       if (numNodes >= support - 1)
```

```
308         {
309           lastRow = nextRow + 1;
310           continue;
311         }
312       else
313         {
314           lastBit = 0;
315           for (j = 0; j < numNodes; j++)
316         {
317           nextBit = nextBitBitSet (graph->graph[nextRow], lastBit);
318           if (nextBit == -1)
319             {
320               fprintf (stderr,
321                 "\nEnd of BitSet reached! = iter,Bit\n");
322               fflush (stderr);
323               exit (0);
324             }
325           setFalse (graph->graph[nextRow], nextBit);
326           setFalse (graph->graph[nextBit], nextRow);
327           setTrue (work, nextBit);
328           lastBit = nextBit + 1;
329         }
330           lastRow = nextRow + 1;
331         }
332     }
333     }
334   return 1;
335 }
```

C.14.0.213 int findCliques (bitSet_t * *Q*, bitSet_t * *cand*, bitSet_t * *mask*, bitGraph_t * *oG*, int *support*, int *qCount*, cll_t ** *elemPats*, int * *indexToSeq*, int *p*)

Recursive algorithm to exhaustively enumerate all of the maximal cliques that exist in the data. This is one of the main workhorses of Gemoda when used in its exhaustive form. This algorithm was originally published by Etsuji Tomita, Akira Tanaka, and Haruhisa Takahasi as a Technical Report of IPSJ (Information Processing Society of Japan): Tomita, E, A Tanaka, & H Takahasi (1989). "An optimal algorithm for finding all of the cliques". SIG Algorithms 12, pp 91-98. Input: a bitset with the nodes currently in the clique, a bitset with the candidates for expanding the clique, a bitset inidcating the current subgraph being searched, the bitGraph to be searched for cliques, the minimum support parameter, a counter variable for keeping track of how many nodes are in the current clique, a linked list of cliques that have been discovered so far, and a pointer to the data structure that dereferences offset indexes into sequence numbers, and the minimum number of unique sequences that must contain the motif. Output: integer success value of 0 (but more importantly, the elemPats clique linked list is expanded to contain all elementary (minimum-length) motif cliques.

Definition at line 37 of file newConv.c.

References bitSetIntersection(), checkBit(), countSet(), deleteBitSet(), bitGraph_t::graph, newBitSet(), nextBitBitSet(), pushClique(), setFalse(), setTrue(), and bitGraph_t::size.

Referenced by convolve().

```
40 {
41   bitSet_t ** gammaOG = NULL;
42   bitSet_t * candQ = newBitSet (oG->size);
43   bitSet_t * newMask = newBitSet (oG->size);
44   int i, q;
45   int graphSize;
46   int max = -1;
47   int numBits;
48   int u = 0;
49   int newMaskCount;
50   int candQCount;
51   graphSize = oG->size;
52
53     //
54     // Find which vertex in subg maximizes |cand intersect gamma(u) |
55     gammaOG = oG->graph;
56   for (i = 0; i < graphSize; i++)
57     {
58
59     // Don't check this vertex if it's masked
60     if (!(checkBit (mask, i)))
61     {
62       continue;
63     }
64
65     // cand is always a subset of mask, so intersecting
66     // with mask is redundant
67     bitSetIntersection (gammaOG[i], cand, candQ);
68       numBits = countSet (candQ);
69       if (numBits > max)
70     {
71       u = i;
72       max = numBits;
73     }
74     }
75
76     // Then do the extension of the q's
77     qCount++;
78
79     // This loop iterates over all possible values of cand - gamma() by
80     // iterating over all possible values of cand but immediately
81     // "continue"ing if the node is also in gamma(u)
82     q = nextBitBitSet (cand, 0);
83   while (q != -1)
84     {
85       if (checkBit (gammaOG[u], q))
86     {
87       q = nextBitBitSet (cand, q + 1);
88       continue;
89     }
90
91     // SUBGq = SUBG i Gamma
92     bitSetIntersection (mask, gammaOG[q], newMask);
93       newMaskCount = countSet (newMask);
94       setTrue (Q, q);
95
96     // Only recurse if there are more candidates to be included,
97     // and they will allow us to reach the minimum support.
98     if (newMaskCount > 0 && qCount + newMaskCount >= support)
99     {
100
101         // CANDq = CAND i Gamma
102         bitSetIntersection (gammaOG[q], cand, candQ);
103       candQCount = countSet (candQ);
104
```

```
105         // only recurse if we can possibly get to a clique
106         // of size with minimum support
107         if (candQCount > 0 && qCount + candQCount >= support)
108         {
109
110         // recursion with
111         // new candidates, new mask, and original graph
112         findCliques (Q, candQ, newMask, oG, support, qCount, elemPats,
113                  indexToSeq, p);
114         }
115     }
116       else if (qCount >= support)
117     {
118
119         // This should be done when:
120         // 1. countSet(newMask) == 0 [connected subgraph is maximal]
121         // 2. Qcount >= minCount [connected subgraph has enough nodes]
122         *elemPats = pushClique (Q, *elemPats, indexToSeq, p);
123     }
124
125     // Remove q from Q, and remove q from cand
126     setFalse (Q, q);
127       setFalse (cand, q);
128       q = nextBitBitSet (cand, q + 1);
129     }
130   qCount--;
131   deleteBitSet (candQ);
132   deleteBitSet (newMask);
133   return 0;
134 }
```

C.14.0.214 bitGraph_t* pruneBitGraph (bitGraph_t * *bg*, int * *indexToSeq*, int ** *offsetToIndex*, int *numOfSeqs*, int *p*)

Simple function (non-recursive) to prune off the first level of motifs that will not meet the "minimum number of unique sequences" criterion. This could have been implemented as above, but it may have gotten a little expensive with less yield, so only the first run through is done here. Input: a bit graph to be pruned, a pointer to the structure that dereferences offset indices to sequence numbers, a pointer to the structure that dereferences seq/position to offsets, the number of unique sequences in the input set, and the minimum number of unique sequences that must contain the motif. Output: a pruned bitGraph.

Definition at line 402 of file newConv.c.

References emptySet(), bitGraph_t::graph, and nextBitBitSet().

```
404 {
405   int i = 0, j = 0, nextBit = 0;
406   int *seqNums = NULL;
407
408     // Since we don't immediately know which node is in which source
409     // sequence, we can't just count them up regularly.  Instead, we'll
410     // need to keep track of which sequences they come from and
411     // increment _something_.  What we chose to do here is just make
412     // an array of integers of length = <p>.  Then, we try to put the
413     // source sequence number of each neighbor (including itself, since
414     // the main diagonal is still true at this time) into the next slot
415     // Since we will monotonically search the bitSet, we can just
416     // move on to the first bit in the next sequence using the
```

```
417     // offsetToIndex structure so that we know the next sequence number
418     // to be put in is always unique.
419     seqNums = (int *) malloc (p * sizeof (int));
420   if (seqNums == NULL)
421     {
422       fprintf (stderr, "Memory error - pruneBitGraph\n%s\n",
423         strerror (errno));
424       fflush (stderr);
425       exit (0);
426     }
427
428     // So, for each row in the bitgraph...
429     for (i = 0; i < bg->size; i++)
430     {
431
432     // Make sure the whole array is -1 sentinels.
433     for (j = 0; j < p; j++)
434     {
435       seqNums[j] = -1;
436     }
437       j = 0;
438
439     // Find the first neighbor of this bit.
440     nextBit = nextBitBitSet (bg->graph[i], 0);
441       if (nextBit == -1)
442     {
443       continue;
444     }
445       else
446     {
447
448         // and put its sequence number in the array of ints.
449         seqNums[0] = indexToSeq[nextBit];
450     }
451
452     // If it's the last sequence, then bail out so that we don't
453     // segfault in the next step.
454     if (seqNums[0] >= numOfSeqs - 1)
455     {
456       emptySet (bg->graph[i]);
457       continue;
458     }
459
460     // Find the next neighbor of this bit, STARTING AT the first
461     // bit in the next sequence.
462     nextBit =
463     nextBitBitSet (bg->graph[i],
464                offsetToIndex[indexToSeq[nextBit] + 1][0]);
465
466     // And iterate this until we run out of neighbors.
467     while (nextBit >= 0)
468     {
469       j++;
470       seqNums[j] = indexToSeq[nextBit];
471
472         // Or until this new neighbor will fill up the array
473         if (j == p - 1)
474         {
475           break;
476         }
477
478         // Or until this new neighbor is in the last sequence.
479         if (seqNums[j] >= numOfSeqs - 1)
480         {
481           break;
482         }
483
484         // Get the next neighbor!
```

```
485         nextBit =
486         nextBitBitSet (bg->graph[i],
487                 offsetToIndex[indexToSeq[nextBit] + 1][0]);
488     }
489
490     // If we didn't have enough unique sequences, and either a) we
491     // were in the nth-to-last sequence and there were no
492     // neighbors after it, or b) we were in the last sequence,
493     // then the last number will still be our sentinel, -1.  If
494     // the last number is not a sentinel, then we have at least
495     // p distinct sequence occurrences, so we're OK.
496     if (seqNums[p - 1] == -1)
497     {
498       emptySet (bg->graph[i]);
499     }
500     }
501   free (seqNums);
502   return (bg);
503 }
```

C.14.0.215 cll_t* pruneCll (cll_t * *head*, int * *indexToSeq*, int *p*)

Prunes a motif linked list of all motifs without support in at least

unique source sequences. Input: head of a motif linked list, pointer to a structure that dereferences offset indices to sequence numbers, minimum number of unique source sequences in which a motif must occur. Output: head of a (potentially altered) motif linked list.

Definition at line 514 of file newConv.c.

References cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by completeConv(), and convolve().

```
515 {
516   int i = 0, j = 0, thisSeq = 0;
517   int *seqNums = NULL;
518   cll_t * curr = head;
519   cll_t * prev = NULL;
520   cll_t * storage = NULL;
521
522     // We'll do this similar to the pruneBitGraph function... we will
523     // keep track of which source sequence each motif occurrence was in.
524     // Again, since the occurrences are listed monotonically, we only
525     // need to compare the last non-sentinel index to the current
526     // sequence number.
527     seqNums = (int *) malloc (p * sizeof (int));
528   if (seqNums == NULL)
529     {
530       fprintf (stderr, "Memory error - pruneCll\n%s\n", strerror (errno));
531       fflush (stderr);
532       exit (0);
533     }
534   while (curr != NULL)
535     {
536
537     // First make sure the set size is at least p.
538     // This is redundant, but extremely simple and not expensive,
539     // so we'll leave it in just as a check.
540     if (curr->set->size < p)
541     {
```

```
542       if (prev != NULL)
543         {
544           prev->next = curr->next;
545         }
546       else
547         {
548           head = curr->next;
549         }
550       storage = curr->next;
551       free (curr->set->members);
552       free (curr->set);
553       free (curr);
554       curr = storage;
555       continue;
556     }
557       for (i = 0; i < p; i++)
558     {
559       seqNums[i] = -1;
560     }
561       j = 0;
562       seqNums[0] = indexToSeq[curr->set->members[0]];
563
564     // Note, we've checked to make sure size > p, and we know
565     // p must be 2 or greater, so we can start at 1 without
566     // worrying about segfaulting
567     for (i = 1; i < curr->set->size; i++)
568     {
569       thisSeq = indexToSeq[curr->set->members[i]];
570       if (thisSeq != seqNums[j])
571         {
572           j++;
573           seqNums[j] = thisSeq;
574           if (j == p - 1)
575         {
576           break;
577         }
578         }
579     }
580
581     // Same story as before... if the last number is -1,
582     // then we didn't have enough to fill up the <p> different
583     // slots, so this doesn't meet our criterion.
584     if (seqNums[p - 1] == -1)
585     {
586       if (prev != NULL)
587         {
588           prev->next = curr->next;
589         }
590       else
591         {
592           head = curr->next;
593         }
594       storage = curr->next;
595       free (curr->set->members);
596       free (curr->set);
597       free (curr);
598       curr = storage;
599     }
600       else
601     {
602       prev = curr;
603       curr = curr->next;
604     }
605     }
606   free (seqNums);
607   return (head);
608 }
```

C.14.0.216 int singleLinkage (bitSet_t * *Q*, bitSet_t * *cand*, bitSet_t * *mask*, bitGraph_t * *oG*, int *support*, int *qCount*, cll_t ** *elemPats*, int * *indexToSeq*, int *p*)

A recursive routine for single linkage clustering. This clustering is much faster than exhaustively enumerating all cliques, but it puts each node in only one cluster and is not guaranteed to give all possible motifs. Input: a bitSet containing the current motif, a bitSet containing candidates to be added to the current motif, a bitSet containing the current subgraph to be clustered, the original bitGraph to be clustered, the minimum support necessary for a motif to be returned, the current number of nodes in the motif, a linked list of elementary motifs (length is the same as the window size), pointer to a structure to derference index values to sequence numbers, and the minimum number of unique sequences that a motif must be in to be returned. Output: integer success value of 0 (but more importantly, the linked list elemPats is updated to contain all of the motifs of length = window size.

Definition at line 154 of file newConv.c.

References bitSetUnion(), checkBit(), copySet(), countSet(), bitGraph_t::graph, nextBitBitSet(), pushClique(), and setFalse().

Referenced by convolve().

```
157 {
158   int i = 0;
159   int j = 0;
160
161     // go to the first vertex that has not been clustered yet
162     i = nextBitBitSet (cand, 0);
163   if (i != -1)
164     {
165
166     // this vertex has been clustered
167     setFalse (cand, i);
168
169     // start a new cluster, Q
170     copySet (oG->graph[i], Q);
171
172     // go over each vertex in the cluster
173     j = nextBitBitSet (Q, 0);
174       while (j != -1)
175     {
176
177         // if this vertex has been clustered already, skip it and go
178         // to the next one
179         if (!checkBit (cand, j))
180         {
181           j = nextBitBitSet (Q, j + 1);
182           continue;
183         }
184
185         // Add this vertex's neighbors to the current cluster
186         bitSetUnion (Q, oG->graph[j], Q);
187
188         // This vertex has now been clustered
189         setFalse (cand, j);
190
191         // go over each vertex in the cluster
192         j = nextBitBitSet (Q, 0);
```

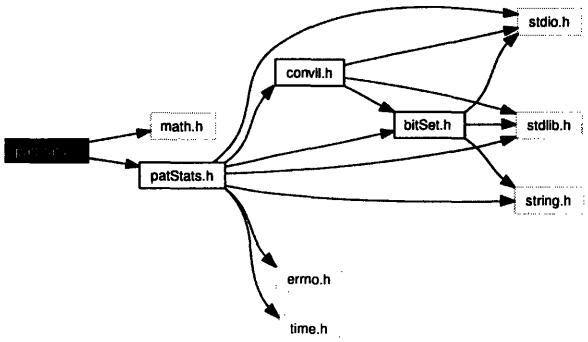
```
193     }
194
195     // Did we make a cluster that was large enough?
196     if (countSet (Q) >= support)
197     {
198       *elemPats = pushClique (Q, *elemPats, indexToSeq, p);
199     }
200
201     // recurse
202     singleLinkage (Q, cand, mask, oG, support, 0, elemPats, indexToSeq,
203                    p);
204     }
205   else
206     {
207       return 0;
208     }
209   return 0;
210 }
```

C.15 patStats.c File Reference

`#include <math.h>`

`#include "patStats.h"`

Include dependency graph for patStats.c:



Functions

- int getLargestSupport (cll_t *cliqs)
- int getLargestLength (cll_t *cliqs)
- int measureDiagonal (const bitGraph_t *bg, const int i, const int j)
- unsigned int ** increaseMem (unsigned int **d, int dimToChange, int currSupport, int currLength, int newVal)
- unsigned int ** oldGetStatMat (bitGraph_t *bg, int support, int length, int *supportDim, int *lengthDim, int numBlanks)
- unsigned int ** getStatMat (bitGraph_t *bg, int support, int length, int *supportDim, int *lengthDim, int numBlanks, int s, FILE *OUTPUT_FILE)
- int cumDMatrix (unsigned int **d, cll_t *cliqs, int currSupport, int currLength, int bgSize, int numSeqs)
- double calcStatCliq (unsigned int **d, cll_t *cliq, int numWindows)
- int calcStatAllCliqs (unsigned int **d, cll_t *allCliqs, int numWindows)
- int freeD (unsigned int **d, int supportDim)
- int statCompare (const cll_t **first, const cll_t **second)
- cll_t * sortByStats (cll_t *allCliqs)

Detailed Description

This file defines functions that are used to compute the statistical significance of motifs for both the sequence based and real value based implementations of Gemoda. The

basic approach we take, is to calculate the probability of establishing a single cluster, and to multiply this probability by the probability that the cluster can be extended an arbitrary number of locations. Essentially, this is the probability of getting and elementary motif during the clustering phase and having that motif convolved multiple times during the convolution phase.

Definition in file patStats.c.

Function Documentation

C.15.0.217 int calcStatAllCliqs (unsigned int ** *d*, cll_t * *allCliqs*, int *numWindows*)

Definition at line 676 of file patStats.c.

References calcStatCliq(), cnode::next, and cnode::stat.

Referenced by main().

```
677 {
678   cll_t * curr = NULL;
679   curr = allCliqs;
680   while (curr != NULL)
681     {
682       curr->stat = calcStatCliq (d, curr, numWindows);
683       curr = curr->next;
684     }
685   return (0);
686 }
```

C.15.0.218 double calcStatCliq (unsigned int ** *d*, cll_t * *cliq*, int *numWindows*)

Definition at line 623 of file patStats.c.

References cnode::length, cnode::set, and cSet_t::size.

Referenced by calcStatAllCliqs().

```
624 {
625   double stat = 0;
626   int i = 0;
627   int supChooseTwo = 0;
628   double interimP = 0;
629   int support = cliq->set->size;
630   int length = cliq->length;
631   double numTrials = 0;
632   if (support < 2)
633     {
634       fprintf (stderr, "Support for cluster less than 2... exiting.\n");
635       fflush (stderr);
636       exit (0);
637     }
638
639     // OK, so support is at least two.  So we make the connections all
```

```
640     // on the first level, knowing that each node being connected has
641     // at least zero in common.  There are [(size of cluster) - 1] of
642     // these connections to be made.
643     // And we know we can call for d[0][1] because if the second index
644     // were out of bounds, then there would be no similarities, and
645     // there would be no reason to call this function.
646     interimP = ((double) d[0][1]) / ((double) d[0][0]);
647   stat = pow (interimP, support - 1);
648   stat *= ((double) numWindows * (numWindows - 1)) / ((double) 2);
649
650     // Now we actually calculate the probability... the first connection
651     // has to be made no matter what, and after that we multiply for
652     // every connection after the first one.  So we descend iteratively
653     // until we have made all connections, terminating after we've made
654     // the single i = (n - 2) connection.  There is no i = (n - 1)
655     // connection.
656     for (i = 1; i < support - 1; i++)
657     {
658       interimP = ((double) d[i][1]) / ((double) d[i][0]);
659       stat *= pow (interimP, support - i - 1);
660       stat *= ((double) (numWindows - (i + 1))) / ((double) (i + 2));
661     } supChooseTwo = (support * (support - 1)) / 2;
662
663     // Remember that length = (numwindows - 1), or alternatively,
664     // the number of extensions... normally we'd want to have the last
665     // p be p[support][numwindows - 1], which corresponds to
666     // alteredD[support][numwindows]/alteredD[support][numwindows-1],
667     // so that means we want our last d to be d[support][numwindows].
668     // Here, we note that the calculation of p's would be continuously
669     // re-normalizing, so multiplying all p's is the same as dividing
670     // the last d by the initial d.
671     interimP = ((double) d[support][length + 1]) / ((double) d[support][1]);
672   stat *= pow (interimP, supChooseTwo);
673   return stat;
674 }
```

C.15.0.219 int cumDMatrix (unsigned int ** *d*, cll_t * *cliqs*, int *currSupport*, int *currLength*, int *bgSize*, int *numSeqs*)

Definition at line 522 of file patStats.c.

References getLargestLength(), and getLargestSupport().

Referenced by main().

```
524 {
525   int maxSup = 0;
526   int maxLen = 0;
527   int i, j;
528   int numWins = 0;
529
530     maxSup = getLargestSupport (cliqs);
531   maxLen = getLargestLength (cliqs);
532
533 /********* COMMENTED OUT
534     // First we note that the number of unique streaks of a given
535     //  support is defined by d[support][1], where as 1 increases,
536     //  the value of d decreases because only unique streaks are
537     //  counted.
538     // We also note that the number of disjoint node-pairs with a given
539     //  number of other nodes in common is defined by d[support][0].
540     // So, in order to properly account for all "unique" comparisons
541     //  (which is equal to (# streaks + # disjoint node-pairs), we must
542     //  add d[support][1] to d[support][0].
```

```
543
544     for (i = 0; i < currSupport + 1; i++) {
545         d[i][0] += d[i][1];
546     }
547     ********************/
548
549     // We no longer need to do that, since now we sum across both
550     // the support and the length dimensions.  Now, d[support][0] will
551     // necessarily include d[support][1] being added to it.  We don't
552     // want to add this anymore, otherwise we would be underestimating
553     // the probability of making that first connection.  For instance,
554     // if there were no nodes with 20 in common that weren't also
555     // connected, and no nodes whatsoever with more than 20 in common,
556     // we'd want the p[20][0] to be 1, which would be
557     // d[20][1]/d[20][0].  When summing across length directions,
558     // this happens naturally, whereas before we needed to do it
559     // artificially as per above.  If we did above, we'd have the
560     // probability of each node being 1/2 instead of 1.
561
562     // Rather than storing doubles and doing lots of multiplications,
563     // we're going to limit the number of operations done in the actual
564     // probability calculation by only storing cumulative sums in d.
565     // Now remember, what we're storing at each location is the
566     // number of nodes with [i] or more nodes in common (including
567     // each other and selves) that can be extended [j] times (with
568     // their initial similarity counting as 1).
569     //
570     // We go up to the last possible index in the length direction, which
571     // means going up to [maxLen].  We know that this is legitimate
572     // because maxLen is less than or equal to the longest possible
573     // diagonal, and the longest possible diagonal will be less
574     // than or equal to currLength.  Since we have allotted
575     // (currLength + 1) integers, we know we're OK to access [currLength].
576     for (j = 0; j < currLength + 1; j++)
577     {
578
579     // We start at currSupport - 1, because currSupport will
580     // clearly not be changed, and this makes it a much easier
581     // loop to read.
582     for (i = currSupport - 1; i >= 0; i--)
583     {
584       d[i][j] += d[i + 1][j];
585     }
586     }
587   for (i = 0; i < currSupport + 1; i++)
588     {
589       for (j = currLength - 1; j >= 0; j--)
590     {
591       d[i][j] += d[i][j + 1];
592     }
593     }
594
595     // Now we need to forcibly set d[0][0] to its correct value... it's
596     // just the total number of comparisons, not including comparisons
597     // to delimiter 0's meant to separate sequences.  The number of
598     // windows is equal to the number of offsets minus the number
599     // of sequences (assuming one delimiter per sequence).  We don't count
600     // the main diagonal, so the first row has one less, and we want to
601     // sum over all the subsequent rows in the upper half of the matrix.
602     // So it's (numWins - 1)*(numWins - 1 + 1)/2 to sum that up.
603     numWins = bgSize - numSeqs;
604   d[0][0] = numWins * (numWins - 1) / 2;
605
606     /*
607         for (i = 0; i <= maxSup; i++) { printf("support = %d:\t",i); for (j = 0; j <=
608        maxLen; j++) { printf("%d\t",d[i][j]); } printf("\n"); }
609      */
610     return 1;
```

```
611 }
```

C.15.0.220 int freeD (unsigned int ** *d*, int *supportDim*)

Definition at line 688 of file patStats.c.

Referenced by main().

```
689 {
690   int i = 0;
691   if (d == 0)
692     {
693       return 0;
694     }
695   else
696     {
697
698     // Still, it's supportDim + 1, because we have an extra
699     // one for the "0" support.
700     for (i = 0; i < supportDim + 1; i++)
701     {
702       free (d[i]);
703     }
704       free (d);
705       return 0;
706     }
707 }
```

C.15.0.221 int getLargestLength (cll_t * *cliqs*)

Given a clique linked list, this function will return an integer which is equal to the length of the member of the linked list with the largest length.

Definition at line 44 of file patStats.c.

References cnode::length, and cnode::next.

Referenced by cumDMatrix().

```
45 {
46   int len = 0;
47   cll_t * curCliq = NULL;
48   curCliq = cliqs;
49   while (curCliq != NULL)
50     {
51       if (curCliq->length > len)
52     {
53       len = curCliq->length;
54     }
55       curCliq = curCliq->next;
56     }
57
58     // We return (len + 1) because the length of the shortest streak
59     // is one, but is stored in the cluster data structure as being
60     // zero (number of extensions that have been made).
61     return (len + 1);
62 }
```

C.15.0.222 int getLargestSupport (cll_t * *cliqs*)

Given a clique linked list, this function will return an integer which is equal to the support of the member of the linked list with the largest support.

Definition at line 22 of file patStats.c.

References cnode::next, cnode::set, and cSet_t::size.

Referenced by cumDMatrix().

```
23 {
24   int size = 0;
25   cll_t * curCliq = NULL;
26   curCliq = cliqs;
27   while (curCliq != NULL)
28     {
29       if (curCliq->set->size > size)
30     {
31       size = curCliq->set->size;
32     }
33       curCliq = curCliq->next;
34     }
35   return size;
36 }
```

C.15.0.223 unsigned int** getStatMat (bitGraph_t * *bg*, int *support*, int *length*, int * *supportDim*, int * *lengthDim*, int *numBlanks*, int *s*, FILE * *OUTPUT_FILE*)

Definition at line 329 of file patStats.c.

References bitGraphRowIntersection(), checkBit(), countSet(), deleteBitSet(), bitGraph_t::graph, increaseMem(), measureDiagonal(), newBitSet(), nextBitBitSet(), and bitGraph_t::size.

Referenced by main().

```
331 {
332   int *Q = NULL;
333   unsigned int **d = NULL;
334   int i, j, k;
335   int x, y;
336   bitSet_t * X = NULL;
337   int currSupport;
338   int currLength;
339   int multiplier = 50;
340   int diagonal = 0;
341   time_t probStart, probEnd;
342   int timeNeeded = 0;
343   int sampleCounter = 1;
344
345     // int visitCounter = 0, uniqCounter = 0;
346     currSupport = support * multiplier;
347   currLength = length * multiplier;
348   X = newBitSet (bg->size);
349
350     // printf("Made bitSet of size %d\n", bg->size);
351     Q = (int *) malloc (bg->size * sizeof (int));
```

```
352   if (Q == NULL)
353     {
354       fprintf (stderr,
355         "\nMemory error --- couldn't allocate array!"  "\n%s\n",
356         strerror (errno));
357       fflush (stderr);
358       exit (0);
359     }
360   for (i = 0; i < bg->size; i++)
361     {
362       Q[i] = 0;
363     }
364   d =
365     (unsigned int **) malloc ((currSupport + 1) * sizeof (unsigned int *));
366   if (d == NULL)
367     {
368       fprintf (stderr,
369         "\nMemory error --- couldn't allocate array!"  "\n%s\n",
370         strerror (errno));
371       fflush (stderr);
372       exit (0);
373     }
374   for (i = 0; i < currSupport + 1; i++)
375     {
376       d[i] =
377     (unsigned int *) malloc ((currLength + 1) * sizeof (unsigned int));
378       if (d[i] == NULL)
379     {
380       fprintf (stderr, "\nMemory error --- couldn't allocate array!"
381             "\n%s\n", strerror (errno));
382       fflush (stderr);
383       exit (0);
384     }
385       for (j = 0; j < currLength + 1; j++)
386     {
387       d[i][j] = 0;
388     }
389     }
390
391     // printf("size=%d\n",bg->size);
392     time (&probStart);
393   for (i = 0; i < bg->size; i++)
394     {
395       if (i == 200)
396     {
397       time (&probEnd);
398       timeNeeded = ((double) (probEnd - probStart)) /
399         ((double) 60) * ((double) bg->size) / ((double) 200);
400       if (timeNeeded > 2)
401         {
402           fprintf (OUTPUT_FILE,
403             "Max total time to calculate probability:\n");
404           fprintf (OUTPUT_FILE, "\t%d minutes\n", timeNeeded);
405           fprintf (OUTPUT_FILE, "Actual time will be less than this, "
406             "but at least half of it.\n");
407           fprintf (OUTPUT_FILE,
408             "To bypass excessive probability calculations,"
409             " cancel and use a different value\n"
410             " for the '-s' flag (samples every "
411             "'s' points).\n");
412           fflush (NULL);
413         }
414     }
415       j = nextBitBitSet (bg->graph[i], 0);
416       while (j >= 0)
417     {
418       k = nextBitBitSet (bg->graph[i], j + 1);
419       while (k >= 0)
```

```
420         {
421           if (checkBit (bg->graph[j], k) == 0)
422         {
423           if (sampleCounter == s)
424             {
425               bitGraphRowIntersection (bg, j, k, X);
426
427             // visitCounter++;
428             if (nextBitBitSet (X, 0) >= i)
429             {
430
431                 // uniqCounter++;
432                 x = countSet (X);
433               while (x > currSupport)
434                 {
435                   d =
436                 increaseMem (d, 1, currSupport, currLength,
437                          currSupport +
438                          support * multiplier);
439                   currSupport += support * multiplier;
440                 }
441               d[x][0] += 1;
442             }
443               sampleCounter = 0;
444             }
445           sampleCounter++;
446         }
447           k = nextBitBitSet (bg->graph[i], k + 1);
448         }
449       if (j <= i)
450         {
451           j = nextBitBitSet (bg->graph[i], j + 1);
452           continue;
453         }
454       bitGraphRowIntersection (bg, i, j, X);
455       x = countSet (X);
456
457         // Note, now we're using "diagonals" rather than
458         // location in a horizontal array.  So you always
459         // start from the main diagonal at 0 and move out.
460         diagonal = j - i;
461
462         // We change this to greater-than-one because
463         // after Q[diagonal] is reduced to one, it isn't
464         // visited again until we reach a new streak, (because
465         // the next bit in the diagonal is a zero), and at
466         // that point we want to start with a new diagonal
467         // measure.
468         if (Q[diagonal] > 1)
469         {
470           y = Q[diagonal] - 1;
471           Q[diagonal]--;
472         }
473       else
474         {
475           y = measureDiagonal (bg, i, j);
476           Q[diagonal] = y;
477         }
478       while (x > currSupport)
479         {
480           d = increaseMem (d, 1, currSupport, currLength,
481                 currSupport + support * multiplier);
482           currSupport += support * multiplier;
483         }
484       while (y > currLength)
485         {
486           d =
487         increaseMem (d, 2, currSupport, currLength,
```

```
488                 currLength + length * multiplier);
489           currLength += length * multiplier;
490         }
491       d[x][y]++;
492       j = nextBitBitSet (bg->graph[i], j + 1);
493
494         /*
495             if(x != 0){ printf("%d:\t%d %d\n", j, x, y); fflush(stdout); }
496          */
497     }
498
499     /*
500         printf("done\n"); fflush(stdout);
501      */
502     }
503
504     // We need to rescale by the sampling factor for all i>0 in d[i][0].
505     //
506     for (i = 1; i < currSupport; i++)
507     {
508       d[i][0] *= s;
509     }
510
511     // Now we only need to assign the correct value for d[0][0]...
512     // but rather than figuring that out, we will just assign it in the
513     // cumulative function, since there it is merely the number of unique
514     // non-self comparisons and is easy to calculate.
515     deleteBitSet (X);
516   free (Q);
517   *supportDim = currSupport;
518   *lengthDim = currLength;
519   return (d);
520 }
```

C.15.0.224 unsigned int** increaseMem (unsigned int ** *d*, int *dimToChange*, int *currSupport*, int *currLength*, int *newVal*)

This function is used to increase the size of an array of pointers to pointers to integers. dimToChange is 1 for the first dimension (support), 2 for the second dimension (length). newVal is the new value for the dimension to be changed, not including the "1" that should be added... so it should just be some integer times the initial support.

Definition at line 91 of file patStats.c.

Referenced by getStatMat(), and oldGetStatMat().

```
93 {
94   int i = 0, j = 0;
95   if (dimToChange == 1)
96     {
97       d =
98     (unsigned int **) realloc (d, (newVal + 1) * sizeof (unsigned int *));
99       if (d == NULL)
100     {
101       fprintf (stderr, "\nMemory error --- couldn't allocate array!"
102             "\n%s\n", strerror (errno));
103       fflush (stderr);
104       exit (0);
105     }
106       for (i = currSupport + 1; i < newVal + 1; i++)
```

```
107     {
108       d[i] =
109         (unsigned int *) malloc ((currLength + 1) *
110                     sizeof (unsigned int));
111       if (d[i] == NULL)
112         {
113           fprintf (stderr,
114             "\nMemory error --- couldn't allocate array!"
115             "\n%s\n", strerror (errno));
116           fflush (stderr);
117           exit (0);
118         }
119       for (j = 0; j < currLength + 1; j++)
120         {
121           d[i][j] = 0;
122         }
123     }
124       return d;
125     }
126   else if (dimToChange == 2)
127     {
128       for (i = 0; i < currSupport + 1; i++)
129     {
130       d[i] =
131         (unsigned int *) realloc (d[i],
132                     (newVal + 1) * sizeof (unsigned int));
133       if (d[i] == NULL)
134         {
135           fprintf (stderr,
136             "\nMemory error --- couldn't allocate array!"
137             "\n%s\n", strerror (errno));
138           fflush (stderr);
139           exit (0);
140         }
141       for (j = currLength + 1; j < newVal + 1; j++)
142         {
143           d[i][j] = 0;
144         }
145     }
146       return d;
147     }
148   else
149     {
150       fprintf (stderr, "Invalid arguments to increaseMem!\n\n");
151       fflush (stderr);
152       exit (0);
153     }
154 }
```

C.15.0.225 int measureDiagonal (const bitGraph_t * *bg*, const int *i*, const int *j*)

Given a bit graph, and two indices within that bit graph, this will return an integer which is equal to the number of values in the bit graph that are true along a diagonal that begins at the two indices. This routine is used to check for streaks in an adjacency matrix and is used during the convolution.

Definition at line 72 of file patStats.c.

References bitGraphCheckBit().

Referenced by getStatMat(), and oldGetStatMat().

```
73 {
74   int len = 0;
75   while (bitGraphCheckBit (bg, i + len, j + len) != 0)
76     {
77       len++;
78     }
79   return len;
80 }
```

C.15.0.226 unsigned int** oldGetStatMat (bitGraph_t * *bg*, int *support*, int *length*, int * *supportDim*, int * *lengthDim*, int *numBlanks*)

OK, here is something that is a little bit "hackish" but that we have to do. Since our initial matrix is being pruned and filtered before being clustered, but we need to calculate stats based on the original matrix, we need to get information from the matrix before pruning, so we're using this function. We could just make a copy of that matrix, but it's far too big, and that would cause an unneccessary constraint on memory, limiting the size of problems we can address. But we need to define just how big our d matrix is before we can use it. We could go through and compute the longest streak beforehand, and then redo everything, but we've already found the first step of finding all of the streaks to be fairly expensive (KLJ). So instead what we'll do is use the user's parameters as a benchmark and expand from there. We'll assume that most of the time, the biggest streak (number of extensions) will be less than 50 times the length given as input by the user, and the biggest support will be less than 50 times the minimum number of support given by the user. This seems perhaps overly conservative, but otherwise is reasonable. We then realize that even on a 64-bit computer, if the user gives L=50 and K=50, we'll still use less than 48 MB of memory... and if L=50 and K=50, it is extremely likely that doubling the adjacency matrix would have been a much worse option. Scaling back to more common values of L$\sim$20 and K$\sim$20, the memory used shoots down to $\sim$9MB, which is definitely acceptable. Now, if for some reason our initial allocation wasn't enough, then we'll have to go through and realloc all of our memory again. Somewhat time-consuming, but hopefully not done too often. Each time we find we try to put something in an index that doesn't exist, we'll reallocate our memory, adding twice as much in the dimension that was violated. It is important to us that we get back the final dimensions of this matrix, since in the support dimension we'll have to sum across all values, and in the length dimension we'll have to be sure we're not at the edge of a matrix during our d manipulations later on.

Definition at line 196 of file patStats.c.

References bitGraphRowIntersection(), countSet(), deleteBitSet(), increaseMem(), measureDiagonal(), newBitSet(), and bitGraph_t::size.

```
198 {
199   int *Q = NULL;
200   unsigned int **d = NULL;
201   int i, j;
```

```
202   int x, y;
203   bitSet_t * X = NULL;
204   int currSupport;
205   int currLength;
206   int multiplier = 50;
207   time_t probStart, probEnd;
208   int timeNeeded = 0;
209   currSupport = support * multiplier;
210   currLength = length * multiplier;
211   X = newBitSet (bg->size);
212
213     // printf("Made bitSet of size %d\n", bg->size);
214     Q = (int *) malloc (bg->size * sizeof (int));
215   if (Q == NULL)
216     {
217       fprintf (stderr,
218         "\nMemory error --- couldn't allocate array!"  "\n%s\n",
219         strerror (errno));
220       fflush (stderr);
221       exit (0);
222     }
223   for (i = 0; i < bg->size; i++)
224     {
225       Q[i] = 0;
226     }
227   d =
228     (unsigned int **) malloc ((currSupport + 1) * sizeof (unsigned int *));
229   if (d == NULL)
230     {
231       fprintf (stderr,
232         "\nMemory error --- couldn't allocate array!"  "\n%s\n",
233         strerror (errno));
234       fflush (stderr);
235       exit (0);
236     }
237   for (i = 0; i < currSupport + 1; i++)
238     {
239       d[i] =
240     (unsigned int *) malloc ((currLength + 1) * sizeof (unsigned int));
241       if (d[i] == NULL)
242     {
243       fprintf (stderr, "\nMemory error --- couldn't allocate array!"
244             "\n%s\n", strerror (errno));
245       fflush (stderr);
246       exit (0);
247     }
248       for (j = 0; j < currLength + 1; j++)
249     {
250       d[i][j] = 0;
251     }
252     }
253   time (&probStart);
254   for (i = 0; i < bg->size; i++)
255     {
256       if (i == 200)
257     {
258       time (&probEnd);
259       timeNeeded = ((double) (probEnd - probStart)) /
260         ((double) 60) * ((double) bg->size) / ((double) 200);
261       if (timeNeeded > 2)
262         {
263           printf ("Max total time to calculate probability:\n");
264           printf ("\t%d minutes\n", timeNeeded);
265           printf ("Actual time will be less than this, but at",
266                "least half of it.\n");
267           printf ("To bypass excessive probability calculations,",
268                "cancel and use the '-d' flag.\n");
269           fflush (NULL);
```

```
270         }
271       }
272         for (j = bg->size - 1; j > i; j--)
273       {
274         bitGraphRowIntersection (bg, i, j, X);
275         x = countSet (X);
276         if (Q[j - 1] != 0)
277           {
278             y = Q[j - 1] - 1;
279             Q[j] = Q[j - 1] - 1;
280           }
281         else
282           {
283             y = measureDiagonal (bg, i, j);
284             Q[j] = y;
285           }
286         while (x > currSupport)
287           {
288             d = increaseMem (d, 1, currSupport, currLength,
289                   currSupport + support * multiplier);
290             currSupport += support * multiplier;
291           }
292         while (y > currLength)
293           {
294             d =
295           increaseMem (d, 2, currSupport, currLength,
296                    currLength + length * multiplier);
297             currLength += length * multiplier;
298           }
299         d[x][y]++;
300
301           /*
302               if(x != 0){ printf("%d:\t%d %d\n", j, x, y); fflush(stdout); }
303            */
304       }
305
306       /*
307           printf("done\n"); fflush(stdout);
308        */
309       }
310
311       // We know that the "blanks", inserted to delimit unique sequences
312       // and prevent convolution through them, will skew our statistics,
313       // so we subtract them.  We know that they will never be similar to
314       // any others, so will only add to the d[0][0] number.  Furthermore,
315       // we know how many they add.  Since d never hits the main diagonal
316       // and only does the upper half of the matrix, the first one
317       // contributes bgsize - 1 to d[0][0], the next bgsize - 2, etc.
318       for (i = 0; i < numBlanks; i++)
319       {
320         d[0][0] -= bg->size - 1 - i;
321       }
322     deleteBitSet (X);
323     free (Q);
324     *supportDim = currSupport;
325     *lengthDim = currLength;
326     return (d);
327   }
```

C.15.0.227 cll_t* sortByStats (cll_t * *allCliqs*)

This function is used to sort a link to list of cliques by the statistical significance of the motifs found in that linked list.

Definition at line 732 of file patStats.c.

References cnode::id, cnode::next, and statCompare().

Referenced by main().

```
733 {
734   cll_t * curCliq = NULL;
735   cll_t ** arrayOfCliqs = NULL;
736   int numOfCliqs = 0;
737   int i = 0;
738   curCliq = allCliqs;
739   if (curCliq != NULL)
740     {
741       numOfCliqs = curCliq->id + 1;
742     }
743   else
744     {
745       return (NULL);
746     }
747   arrayOfCliqs = (cll_t **) malloc (numOfCliqs * sizeof (cll_t *));
748   for (i = 0; i < numOfCliqs; i++)
749     {
750       arrayOfCliqs[i] = curCliq;
751       curCliq = curCliq->next;
752     }
753   qsort (arrayOfCliqs, numOfCliqs, sizeof (cll_t *), statCompare);
754   for (i = 0; i < numOfCliqs - 1; i++)
755     {
756       arrayOfCliqs[i]->next = arrayOfCliqs[i + 1];
757     }
758   arrayOfCliqs[numOfCliqs - 1]->next = NULL;
759   return (arrayOfCliqs[0]);
760 }
```

C.15.0.228 int statCompare (const cll_t ** *first*, const cll_t ** *second*)

Definition at line 709 of file patStats.c.

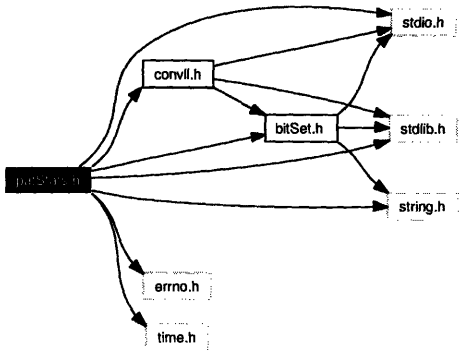
Referenced by sortByStats().

```
710 {
711   double difference = (*first)->stat - (*second)->stat;
712   if (difference < 0)
713     {
714       return (-1);
715     }
716   else if (difference > 0)
717     {
718       return (1);
719     }
720   else
721     {
722       return (0);
723     }
724 }
```

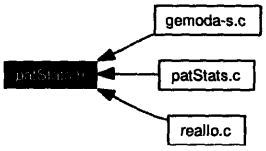
C.16 patStats.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "bitSet.h"
#include "convll.h"
#include <time.h>
```

Include dependency graph for patStats.h:



This graph shows which files directly or indirectly include this file:



Functions

- unsigned int ** getStatMat (bitGraph_t *bg, int support, int length, int *supportDim, int *lengthDim, int numBlanks, int s, FILE *OUTPUT_FILE)
- int cumDMatrix (unsigned int **d, cll_t *cliqs, int currSupport, int currLength, int bgSize, int numSeqs)
- int calcStatAllCliqs (unsigned int **d, cll_t *allCliqs, int numWindows)
- cll_t * sortByStats (cll_t *allCliqs)
- int freeD (unsigned int **d, int supportDim)

Function Documentation

C.16.0.229 int calcStatAllCliqs (unsigned int ** *d*, cll_t * *allCliqs*, int *numWindows*)

Definition at line 623 of file patStats.c.

References calcStatCliq(), cnode::next, and cnode::stat.

Referenced by main().

C.16.0.230 int cumDMatrix (unsigned int ** *d*, cll_t * *cliqs*, int *currSupport*, int *currLength*, int *bgSize*, int *numSeqs*)

Definition at line 460 of file patStats.c.

References getLargestLength(), and getLargestSupport().

Referenced by main().

C.16.0.231 int freeD (unsigned int ** *d*, int *supportDim*)

Definition at line 637 of file patStats.c.

Referenced by main().

C.16.0.232 unsigned int** getStatMat (bitGraph_t * *bg*, int *support*, int *length*, int * *supportDim*, int * *lengthDim*, int *numBlanks*, int *s*, FILE * *OUTPUT_FILE*)

Definition at line 289 of file patStats.c.

References bitGraphRowIntersection(), checkBit(), countSet(), deleteBitSet(), bitGraph_t::graph, increaseMem(), measureDiagonal(), newBitSet(), nextBitBitSet(), and bitGraph_t::size.

Referenced by main().

C.16.0.233 cll_t* sortByStats (cll_t * *allCliqs*)

This function is used to sort a link to list of cliques by the statistical significance of the motifs found in that linked list.

Definition at line 674 of file patStats.c.

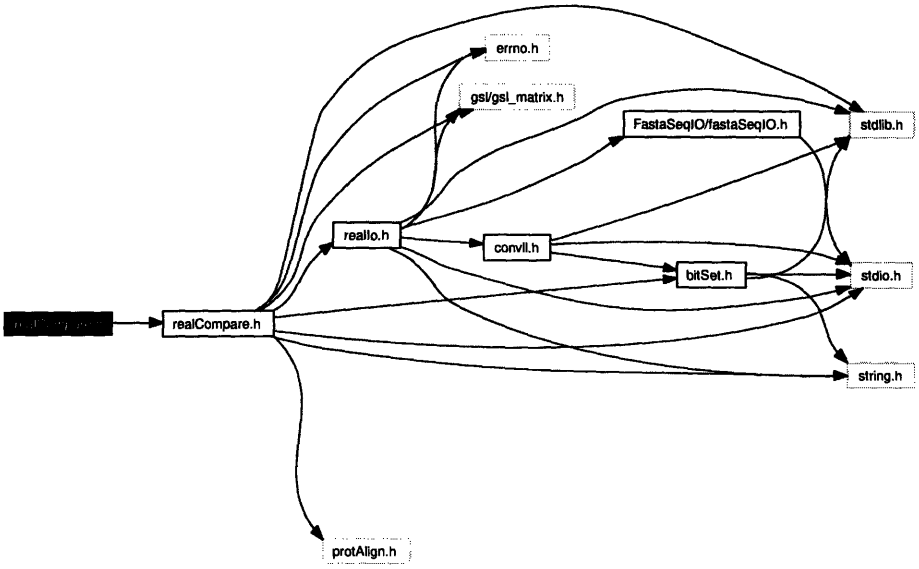
References cnode::id.

Referenced by main().

C.17 realCompare.c File Reference

`#include "realCompare.h"`

Include dependency graph for realCompare.c:



Functions

- double rmsdCompare (rdh_t *data, int win1, int win2, int L, double *extra-Params)
- double generalMatchFactor (rdh_t *data, int win1, int win2, int L, double *extraParams)
- double massSpecCompareWElut (rdh_t *data, int win1, int win2, int L, double *extraParams)
- double(*)(rdh_t *, int, int, int, double *) getCompFunc (int compFunc)
- bitGraph_t * realComparison (rdh_t *data, int L, double g, int compFunc, double *extraParams)

Detailed Description

This file defines a series of functions that are used during the comparison phase of the Gemoda algorithm in the real valued implementation. We define a handful of comparison functions — some that are well suited to protein structure comparison and others that are more suited to the comparison of mass spectrometry spectra.

Definition in file realCompare.c.

Function Documentation

C.17.0.234 double generalMatchFactor (rdh_t * *data*, int *win1*, int *win2*, int *L*, double * *extraParams*)

This function is used to compute a generalized match factor, which is useful for computing the degree of similarity between mass spectrometry spectra.

Definition at line 111 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc().

```
113 {
114   int i, j;
115   double numerator = 0.0;
116
117     /*
118        double denominator=0.0;
119      */
120   double xsum;
121   double ysum;
122   double ldenom = 0.0;
123   double rdenom = 0.0;
124   int dim;
125   int seq1, pos1;
126   int seq2, pos2;
127   gsl_matrix_view view1;
128   gsl_matrix_view view2;
129   gsl_matrix * mat1;
130   gsl_matrix * mat2;
131   dim = getRdhDim (data);
132
133     // Find out which seq,pos pairs these two
134     // windows correspond to
135     getRdhIndexSeqPos (data, win1, &seq1, &pos1);
136   getRdhIndexSeqPos (data, win2, &seq2, &pos2);
137
138     // Get a reference to a submatrix.  That is,
139     // 'chop out' the window.
140     view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
141   view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);
142
143     // Some error checking here would be nice!
144     // Did we get the matrices we wanted?
145
146     // This just makes it easier to handle the views
147     mat1 = &view1.matrix;
148   mat2 = &view2.matrix;
149
150     // Loop over each position
151     for (i = 0; i < mat1->size1; i++)
152     {
153       xsum = 0.0;
154       ysum = 0.0;
155
156     // Loop over each dimension at each position
157     for (j = 0; j < dim; j++)
158     {
159       xsum += gsl_matrix_get (mat1, i, j);
160       ysum += gsl_matrix_get (mat2, i, j);
161     }
162       numerator += (i + 1) * sqrt (xsum * ysum);
```

```
163       ldenom += (i + 1) * xsum;
164       rdenom += (i + 1) * ysum;
165     }
166   return pow (numerator, 2.0) / (ldenom * rdenom);
167 }
```

C.17.0.235 double(*)(rdh_t *, int, int, int, double *) getCompFunc ()

Definition at line 264 of file realCompare.c.

References generalMatchFactor(), massSpecCompareWElut(), and rmsdCompare().

```
265 {
266   double (*comparisonFunc) (rdh_t *, int, int, int, double *) = &rmsdCompare;
267   switch (compFunc)
268     {
269     case 0:
270       comparisonFunc = &rmsdCompare;
271       break;
272     case 1:
273       comparisonFunc = &generalMatchFactor;
274       break;
275     case 2:
276       comparisonFunc = &massSpecCompareWElut;
277       break;
278     default:
279       comparisonFunc = &rmsdCompare;
280       break;
281     }
282   return (comparisonFunc);
283 }
```

C.17.0.236 double massSpecCompareWElut (rdh_t * *data*, int *win1*, int *win2*, int *L*, double * *extraParams*)

This function is used to compute the match factor between to mass spectrometry spectra in a similar manner to the previous function; however, this function imposes a penalty for spectra that are separated by large distances in elution time. This function is commonly used by SpecConnect.

Definition at line 178 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc().

```
180 {
181   int i, j;
182   double numerator = 0.0;
183
184     /*
185        double denominator=0.0;
186      */
187   double xsum;
188   double ysum;
189   double cum;
190   double ldenom = 0.0;
191   double rdenom = 0.0;
```

```
192   int dim;
193   int seq1, pos1;
194   int seq2, pos2;
195   double weight = 2.0;
196   gsl_matrix_view view1;
197   gsl_matrix_view view2;
198   gsl_matrix * mat1;
199   gsl_matrix * mat2;
200   double maxElut = -1;
201   if (extraParams != NULL)
202     {
203       maxElut = extraParams[0];
204     }
205   dim = getRdhDim (data);
206
207     // Find out which seq,pos pairs these two
208     // windows correspond to
209     getRdhIndexSeqPos (data, win1, &seq1, &pos1);
210   getRdhIndexSeqPos (data, win2, &seq2, &pos2);
211
212     // Get a reference to a submatrix.  That is,
213     // 'chop out' the window.
214     view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
215   view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);
216
217     // Some error checking here would be nice!
218     // Did we get the matrices we wanted?
219
220     // This just makes it easier to handle the views
221     mat1 = &view1.matrix;
222   mat2 = &view2.matrix;
223   cum = 1.0;
224
225     // Loop over each position
226     for (i = 0; i < mat1->size1; i++)
227     {
228       xsum = 0.0;
229       ysum = 0.0;
230
231     // First take the first dimension for elution time
232     if (maxElut >= 0)
233     {
234       if (fabs
235           (gsl_matrix_get (mat1, i, 0) - gsl_matrix_get (mat2, i, 0)) >
236           maxElut)
237         {
238           cum = 0;
239           break;
240         }
241     }
242
243     // printf("\n");
244     //
245     // Loop over each subsequent dimension at each position
246     for (j = 1; j < dim; j++)
247     {
248
249         // printf("mat1val=%lf,mat2val=%lf\n",gsl_matrix_get(mat1,i,j),
250         // gsl_matrix_get(mat2,i,j));
251         numerator += pow (j, weight) * sqrt (gsl_matrix_get (mat1, i, j)
252                         *gsl_matrix_get (mat2, i,
253                                          j));
254       ldenom += pow (j, weight) * gsl_matrix_get (mat1, i, j);
255       rdenom += pow (j, weight) * gsl_matrix_get (mat2, i, j);
256
257         // printf("numer=%lf,ldenom=%lf,rdenom=%lf\n",numerator,
258         // ldenom,rdenom);
259     }
```

```
260       cum *= pow (numerator, 2.0) / (ldenom * rdenom);
261     }
262   return pow (cum, 1.0 / L);
263 }
```

C.17.0.237 bitGraph_t* realComparison (rdh_t * *data*, int *L*, double *g*, int *compFunc*, double * *extraParams*)

Definition at line 285 of file realCompare.c.

References bitGraphSetTrueSym(), getCompFunc, getRdhIndexSeqPos(), rdh_t::indexSize, initRdhIndex(), newBitGraph(), and rmsdCompare().

Referenced by main().

```
287 {
288   int i, j;
289   int seq1, pos1;
290   int seq2, pos2;
291   bitGraph_t * bg = NULL;
292   double score;
293   double (*comparisonFunc) (rdh_t *, int, int, int, double *) = &rmsdCompare;
294
295     // Initialize the rdh's index
296     initRdhIndex (data, L, 1);
297
298     // Allocate a new bit graph
299     bg = newBitGraph (data->indexSize);
300
301     // Choose the comparison function, pass a reference to it
302     comparisonFunc = getCompFunc (compFunc);
303   for (i = 0; i < data->indexSize; i++)
304     {
305
306     // Skip seperators
307     getRdhIndexSeqPos (data, i, &seq1, &pos1);
308       if (seq1 == -1 || pos1 == -1)
309     {
310       continue;
311     }
312       for (j = i; j < data->indexSize; j++)
313     {
314       getRdhIndexSeqPos (data, j, &seq2, &pos2);
315       if (seq2 == -1 || pos2 == -1)
316         {
317           continue;
318         }
319
320         // This is the comparison function
321         score = comparisonFunc (data, i, j, L, extraParams);
322
323         // printf("score (%2d,%2d) vs. (%2d, %2d) =\t%lf\n",seq1, pos1, seq2, pos2,
324         // score);
325         if (compFunc == 0)
326         {
327           if (score <= g)
328         {
329           bitGraphSetTrueSym (bg, i, j);
330         }
331         }
332       else if ((compFunc == 1) || (compFunc == 2))
333         {
334           if (score >= g)
```

```
335         {
336           bitGraphSetTrueSym (bg, i, j);
337         }
338         }
339       else
340         {
341           fprintf (stderr, "Comparison function undefined in "
342             "realComparison function,\n located in "
343             "realCompare.c.  Exiting.\n\n");
344           fflush (stderr);
345           exit (0);
346         }
347     }
348     }
349   return bg;
350 }
```

C.17.0.238 double rmsdCompare (rdh_t * *data*, int *win1*, int *win2*, int *L*, double * *extraParams*)

Calculate the rmsd between two windows, with optional translation and rotation. The input to this function is a real data handler object, two integers that point to the windows within the real data that are to be compared, an integer that specifies the length of the windows, and a pointer to a double precision floating point that can be used to store other parameters as needed. This last parameter is most useful for implementing other comparison functions, without having to make, too many changes to other parts of the code.

This function operates in three stages. First, we compute the centroid of each window and move the second window such that its centroid overlaps with that of the first window. Second, we use rigid body rotation to find the rotational matrix that minimizes the root mean squared deviation between the two windows. Finally, this function returns that minimized RMSD.

Definition at line 31 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc(), and realComparison().

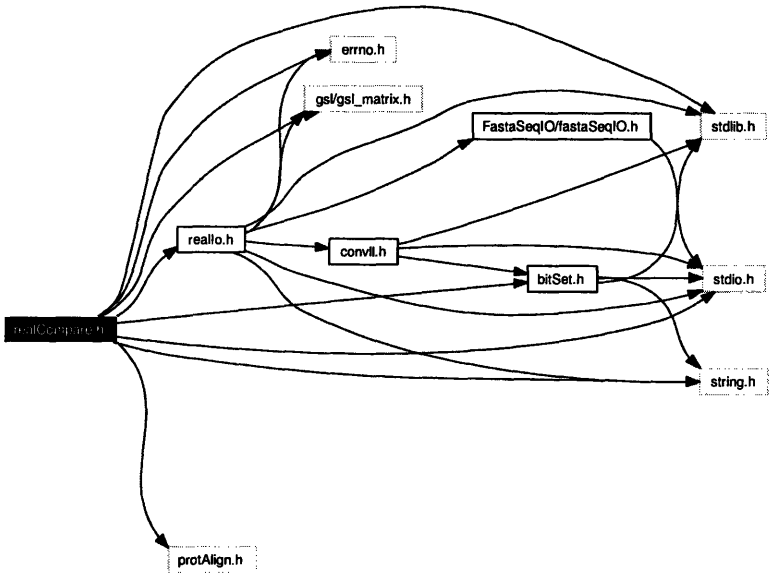
```
32 {
33   int trans = 1;
34   int rot = 1;
35   int dim;
36   double result = 0;
37   int seq1, pos1;
38   int seq2, pos2;
39   gsl_matrix_view view1;
40   gsl_matrix_view view2;
41   gsl_matrix * mat1;
42   gsl_matrix * mat2;
43   gsl_matrix * mat1copy;
44   gsl_matrix * mat2copy;
45
46     // The "rint" function is in math.h and rounds a number to the
47     // nearest integer.  It raises an "inexact exception" if the
48     // number initially wasn't an integer.
49     if (extraParams != NULL)
```

```
50      {
51        trans = rint (extraParams[0]);
52        rot = rint (extraParams[1]);
53      }
54    dim = getRdhDim (data);
55
56      // Find out which seq,pos pairs these two
57      // windows correspond to
58      getRdhIndexSeqPos (data, win1, &seq1, &pos1);
59    getRdhIndexSeqPos (data, win2, &seq2, &pos2);
60
61      // Get a reference to a submatrix.  That is,
62      // 'chop out' the window.
63      view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
64    view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);
65
66      // This just makes it easier to handle the views
67      mat1 = &view1.matrix;
68    mat2 = &view2.matrix;
69
70      // Create copies of the windows, because our comparison
71      // will require altering the matrices
72      mat1copy = gsl_matrix_alloc (mat1->size1, mat1->size2);
73    mat2copy = gsl_matrix_alloc (mat2->size1, mat2->size2);
74    gsl_matrix_memcpy (mat1copy, mat1);
75    gsl_matrix_memcpy (mat2copy, mat2);
76
77      /*
78          printf("matrix1:\n"); gsl_matrix_pretty_fprintf(stdout, mat1copy, "%f ");
79         printf("\nmatrix2:\n"); gsl_matrix_pretty_fprintf(stdout, mat2copy, "%f ");
80       */
81
82      // Are we going to do a translation?
83      if (trans == 1)
84      {
85        moveToCentroid (mat1copy);
86        moveToCentroid (mat2copy);
87      }
88
89      // Are we going to do a rotation?
90      if (rot == 1)
91      {
92
93      // Rotate mat2copy to have a minimal
94      // rmsd with mat1copy
95      rotateMats (mat1copy, mat2copy);
96      }
97
98      // Compute the rmsd between mat2copy and mat2copy
99      result = gsl_matrix_rmsd (mat1copy, mat2copy);
100   gsl_matrix_free (mat1copy);
101   gsl_matrix_free (mat2copy);
102   return result;
103 }
```

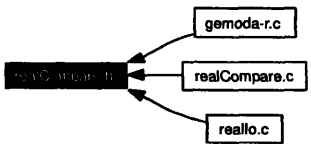
C.18 realCompare.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <gsl/gsl_matrix.h>
#include "realIo.h"
#include "bitSet.h"
#include "protAlign.h"
```

Include dependency graph for realCompare.h:



This graph shows which files directly or indirectly include this file:



Functions

- double rmsdCompare (rdh_t *data, int win1, int win2, int L, double *extra-Params)
- double generalMatchFactor (rdh_t *data, int win1, int win2, int L, double *extraParams)
- double massSpecCompareWElut (rdh_t *data, int win1, int win2, int L, double *extraParams)
- bitGraph_t * realComparison (rdh_t *data, int l, double g, int compFunc, double *extraParams)

Variables

- double(*)(rdh_t *, int, int, int, double *) getCompFunc (int compFunc)

Detailed Description

This file contains declarations and definitions used for the comparison of real valued data during the comparison phase of Gemoda. The functions declared here are defined in realCompare.c.

Definition in file realCompare.h.

Function Documentation

C.18.0.239 double generalMatchFactor (rdh_t * *data*, int *win1*, int *win2*, int *L*, double * *extraParams*)

This function is used to compute a generalized match factor, which is useful for computing the degree of similarity between mass spectrometry spectra.

Definition at line 111 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc().

C.18.0.240 double massSpecCompareWElut (rdh_t * *data*, int *win1*, int *win2*, int *L*, double * *extraParams*)

This function is used to compute the match factor between to mass spectrometry spectra in a similar manner to the previous function; however, this function imposes a penalty for spectra that are separated by large distances in elution time. This function is commonly used by SpecConnect.

Definition at line 174 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc().

C.18.0.241 bitGraph_t* realComparison (rdh_t * *data*, int *l*, double *g*, int *compFunc*, double * *extraParams*)

Definition at line 272 of file realCompare.c.

References bitGraphSetTrueSym(), getCompFunc, getRdhIndexSeqPos(), rdh_t::indexSize, initRdhIndex(), newBitGraph(), and rmsdCompare().

Referenced by main().

C.18.0.242 double rmsdCompare (rdh_t * *data*, int *win1*, int *win2*, int *L*, double * *extraParams*)

Calculate the rmsd between two windows, with optional translation and rotation. The input to this function is a real data handler object, two integers that point to the windows within the real data that are to be compared, an integer that specifies the length of the windows, and a pointer to a double precision floating point that can be used to store other parameters as needed. This last parameter is most useful for implementing other comparison functions, without having to make, too many changes to other parts of the code.

This function operates in three stages. First, we compute the centroid of each window and move the second window such that its centroid overlaps with that of the first window. Second, we use rigid body rotation to find the rotational matrix that minimizes the root mean squared deviation between the two windows. Finally, this function returns that minimized RMSD.

Definition at line 31 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc(), and realComparison().

Variable Documentation

C.18.0.243 double(*)(rdh_t*, int, int, int, double*) getCompFunc(int compFunc)

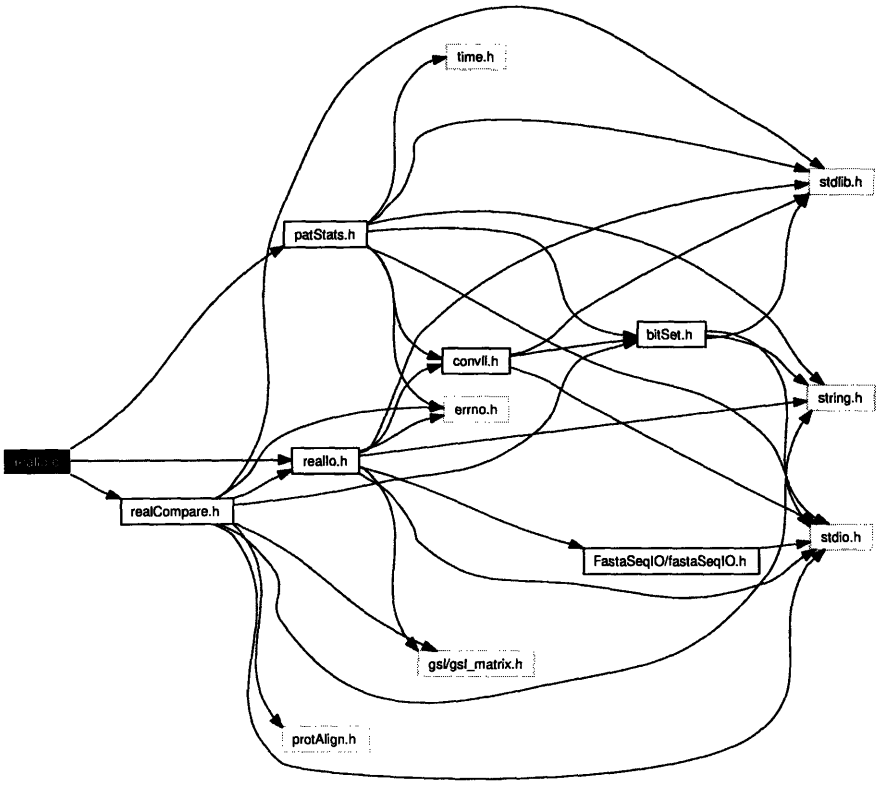
Definition at line 36 of file realCompare.h.

Referenced by findCliqueCentroid(), outputRealPatsWCentroid(), and realComparison().

C.19 realIo.c File Reference

```
#include "realIo.h"
#include "realCompare.h"
#include "patStats.h"
```

Include dependency graph for realIo.c:



Functions

- wordToDouble (char *s, int begin, int end)
- int countFields (char *s, char sep)
- int checkRealDataFormat (char **buf, int nl, char sep, int *numSeq_p, int *dim_p)
- int countTotalFields (char **buf, int nl, char sep)
- rdh_t * initRdh (int x)
- int getRdhSeqLength (rdh_t *data, int seqNo)
- int initRdhIndex (rdh_t *data, int wordSize, int seqGap)

- rdh_t * freeRdh (rdh_t *data)
- int getRdhDim (rdh_t *data)
- int setRdhLabel (rdh_t *data, int seqNo, char *s)
- int setRdhValue (rdh_t *data, int seqNo, int posNo, int dimNo, double val)
- int setRdhIndex (rdh_t *data, int seqNo, int posNo, int index)
- int getRdhIndexSeqPos (rdh_t *data, int index, int *seq, int *pos)
- double getRdhValue (rdh_t *data, int seqNo, int posNo, int dimNo)
- char * getRdhLabel (rdh_t *data, int seqNo)
- int printRdhSeq (rdh_t *data, int seqNo, FILE *FH)
- int setRdhColFromString (rdh_t *data, int seqNo, int colNo, char *s, char sep)
- int initRdhGslMat (rdh_t *data, int seqNo, int x, int y)
- int pushOnRdhSeq (rdh_t *data, char **buf, int startLine, int dim, char sep)
- rdh_t * parseRealData (char **buf, int nl, char sep, int numSeq, int dim)
- rdh_t * readRealData (FILE *INPUT)
- int outputRealPats (rdh_t *data, cll_t *allPats, int L, FILE *OUTPUT_FILE, int **d)
- int findCliqueCentroid (rdh_t *data, cll_t *curCliq, int L, int compFunc, double *extraParams, int *candidates)
- int makeAlternateCentroid (rdh_t *data, cll_t *curCliq, int *candidates)
- int outputRealPatsWCentroid (rdh_t *data, cll_t *allPats, int L, FILE *OUTPUT_FILE, double *extraParams, int compFunc)

Detailed Description

This file defines functions that are used for the parsing of user supplied data in the real valued implementation of Gemoda.

Definition in file realIo.c.

Function Documentation

C.19.0.244 int checkRealDataFormat (char ** *buf*, int *nl*, char *sep*, int * *numSeq_p*, int * *dim_p*)

Check that each sequence has the same dimensionality and that, within a sequence, each dimension has the same number of entries. Note: this routine alters *nunSeq_p and *dim_p! Also, you must call this routine before calling parseRealData. Otherwise, parseRealData is garunteed to die if the data turn out to be ill-formatted.

Definition at line 163 of file realIo.c.

References countFields().

Referenced by readRealData().

```
{
  int i;
  int thisDim = 0;
  int status = 1;
  int width;
  int fieldCount = 0;       // number of positions in a single sequence
  int numSeq = 0;       // number of sequences
  int dim = 0;          // The dimensionality of the sequences

  // NOTE this is not checking the dimensionality of the last sequence...
  // that's bad.  We can fix that though.
  // Check the dimensionality of each sequence
  for (i = 0; i < nl; i++)
    {
      if (buf[i][0] == '>')
    {

      // If this is only the second sequence we've seen,
      // record the dimensionality of the first sequence
      // as the dim to insist upon from here on out
      if (numSeq == 1)
        {
          dim = thisDim;

          // For other sequences, we need to check to make sure
          // that they've got the same dimensions as previous
          // sequences
        }
      else if (numSeq > 1)
        {

          // If the dimensions are wrong, quit with status=0
          if (thisDim != dim)
        {
          status = 0;
          break;
        }
        }
      numSeq++;
      width = 0;
      thisDim = 0;
    }
      else
    {

      // Field count can be different for each sequence but
      // must be the same for each dimension in a single sequence
      fieldCount = countFields (buf[i], sep);

      // If this is the first row of this sequence,
      // then store the number of fields
      if (thisDim == 0)
        {
          width = fieldCount;

          // If it's not the first row, make sure it has the
          // same number of fields as previous rows in this
          // sequence
        }
      else
        {
          if (fieldCount != width)
        {
          status = 0;
          break;
        }
        }
      thisDim++;
```

```
232     }
233     }
234
235   // Pass back the numSeq and dim
236   *numSeq_p = numSeq;
237   *dim_p = thisDim;
238   return status;
239 }
```

C.19.0.245 int countFields (char * *s*, char *sep*)

Count the number of fields (delimited by 'sep') in a single string. I was going to use strsep in string.h for this; however, I don't like that it changes the input string, which makes free-ing the string later more tricky. Ignores consecutive seperators.

Definition at line 90 of file realIo.c.

References wordToDouble().

Referenced by checkRealDataFormat(), countTotalFields(), and pushOnRdhSeq().

```
91 {
92   int i;
93   int begin = 0;
94   int end = 0;
95   int status = 0;       // 0 = in sep, 1 = in word
96   int fieldCount = 0;
97   double val;
98   if (s == NULL)
99     {
100       fprintf (stderr, "Passed NULL string to countFields -- error!");
101       fflush (stderr);
102       exit (0);
103     }
104
105   // Loop over the length of the string
106   for (i = 0; i < strlen (s); i++)
107     {
108
109       // The previous state was space
110       if (status == 0)
111     {
112
113       // We hit a word
114       if (s[i] != sep)
115         {
116           begin = i;
117           status = 1;
118         }
119       else
120         {           // We hit more space
121           continue;
122         }
123     }
124       else
125     {           // The previous state was word
126       if (s[i] != sep)
127         {
128           continue;
129         }
130       else
131         {           // We hit a space
132           end = i - 1;
```

```
133           status = 0;
134
135           // being and end now delimit a word,
136           // turn that word into a double
137           val = wordToDouble (s, begin, end);
138           fieldCount++;
139         }
140     }
141     }
142
143   // At the end, if we were in a word, we have
144   // one more field
145   if (status == 1)
146     {               // We're in a word
147       val = wordToDouble (s, begin, strlen (s));
148       fieldCount++;
149     }
150   return fieldCount;
151 }
```

C.19.0.246 int countTotalFields (char ** *buf*, int *nl*, char *sep*)

Count the number of fields in each sequence and return the sum of these.

Definition at line 246 of file realIo.c.

References countFields().

Referenced by parseRealData().

```
247 {
248   int i = 0;
249   int totalFields = 0;
250   int seqNo = 0;
251   while (i < nl)
252     {
253
254       // Hit a new sequence
255       if (buf[i][0] == '>')
256     {
257       seqNo++;
258
259       // Assume that the sequence has at least
260       // one row (should have called checkRealDataFormat!
261       // and that each row has the same number of fields
262       totalFields += countFields (buf[i + 1], sep);
263     }
264       i++;
265     }
266   return totalFields;
267 }
```

C.19.0.247 int findCliqueCentroid (rdh_t * *data*, cll_t * *curCliq*, int *L*, int *compFunc*, double * *extraParams*, int * *candidates*)

This function is used to find the centroid of a clique. That is, to find the center of mass.

Definition at line 1096 of file realIo.c.

References getCompFunc, cSet_t::members, cnode::set, and cSet_t::size.

Referenced by outputRealPatsWCentroid().

```
1098 {
1099   double (*comparisonFunc) (rdh_t *, int, int, int, double *) = NULL;
1100   int i = 0, j = 0, indmin = -1, counter = 0;
1101   double sim = 0, min = 0, flagmin = 0;
1102   double *cliqueAdjMat = NULL;
1103   cliqueAdjMat = (double *) malloc (curCliq->set->size * sizeof (double));
1104   if (cliqueAdjMat == NULL)
1105     {
1106       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
1107       fflush (stderr);
1108       exit (0);
1109     }
1110   for (i = 0; i < curCliq->set->size; i++)
1111     {
1112       cliqueAdjMat[i] = 0;
1113     }
1114
1115   // We'll accumulate our comparison function values... except here
1116   // we're really assuming that we're using a match factor, with
1117   // value less than one, so that we can subtract it from one to
1118   // get a distance, and then find the centroid by identifying the
1119   // node with the smallest cumulative Euclidean distance to all
1120   // nodes.
1121   // Note that we only need to compare each unique pair, and can apply
1122   // the results from each comparison to each member of the pair,
1123   // hence the somewhat odd indices of initiation for the for loops.
1124   comparisonFunc = getCompFunc (compFunc);
1125   for (i = 0; i < curCliq->set->size; i++)
1126     {
1127       for (j = i + 1; j < curCliq->set->size; j++)
1128     {
1129       sim =
1130         comparisonFunc (data, curCliq->set->members[i],
1131                 curCliq->set->members[j], L, extraParams);
1132
1133       // printf("i = %d, j = %d, L = %d, extra = %lf, sim =
1134       // %lf\n",i,j,L,extraParams[0],sim);
1135       cliqueAdjMat[i] += pow (1 - sim, 2);
1136       cliqueAdjMat[j] += pow (1 - sim, 2);
1137     }
1138     }
1139
1140   // Now we find the minimum Euclidean distance.
1141   min = cliqueAdjMat[0];
1142   indmin = 0;
1143   for (i = 1; i < curCliq->set->size; i++)
1144     {
1145
1146       // printf("index %d product = %lf\n",i,cliqueAdjMat[i]);
1147       if (cliqueAdjMat[i] < min)
1148     {
1149       indmin = i;
1150       min = cliqueAdjMat[i];
1151       flagmin = 0;
1152     }
1153       else if (cliqueAdjMat[i] == min)
1154     {
1155       flagmin = 1;
1156     }
1157     }
1158
1159   // If we had a duplicate on the minimum, we locate all duplicates.
1160   if (flagmin == 1)
```

```
1161     {
1162       counter = 0;
1163       for (i = 0; i < curCliq->set->size; i++)
1164     {
1165       if (cliqueAdjMat[i] == min)
1166         {
1167           counter++;
1168           candidates[counter] = i;
1169         }
1170     }
1171
1172       // Store the number of candidates at the array's beginning
1173       candidates[0] = counter;
1174       free (cliqueAdjMat);
1175       return (-1);
1176     }
1177   else
1178     {
1179       free (cliqueAdjMat);
1180       return (indmin);
1181     }
1182 }
```

C.19.0.248 rdh_t* freeRdh (rdh_t * *data*)

This function returns a null pointer after freeing the memory associated with a real data holder object. The function takes one parameter: a pointer to the real data holder, *data*.

Definition at line 462 of file realIo.c.

References rdh_t::indexToPos, rdh_t::indexToSeq, rdh_t::label, rdh_t::offsetToIndex, and rdh_t::seq.

Referenced by main().

```
463 {
464   int i;
465   if (data != NULL)
466     {
467       if (data->indexToPos != NULL)
468     {
469       free (data->indexToPos);
470       data->indexToPos = NULL;
471     }
472       if (data->indexToSeq != NULL)
473     {
474       free (data->indexToSeq);
475       data->indexToSeq = NULL;
476     }
477       if (data->offsetToIndex != NULL)
478     {
479       for (i = 0; i < data->size; i++)
480         {
481           free (data->offsetToIndex[i]);
482           data->offsetToIndex[i] = NULL;
483         }
484       free (data->offsetToIndex);
485       data->offsetToIndex = NULL;
486     }
487       for (i = 0; i < data->size; i++)
488     {
```

```
489       if (data->seq[i] != NULL)
490         {
491           gsl_matrix_free (data->seq[i]);
492           data->seq[i] = NULL;
493         }
494       if (data->label[i] != NULL)
495         {
496           free (data->label[i]);
497           data->label[i] = NULL;
498         }
499     }
500       if (data->seq != NULL)
501     {
502       free (data->seq);
503       data->seq = NULL;
504     }
505       if (data->label != NULL)
506     {
507       free (data->label);
508       data->label = NULL;
509     }
510       free (data);
511       data = NULL;
512     }
513   return data;
514 }
```

C.19.0.249 int getRdhDim (rdh_t * *data*)

This function returns an integer equal to the dimensions of the data stored in a real data holder object. The function takes one parameter: a pointer to the real data holder, *data*.

Definition at line 524 of file realIo.c.

References rdh_t::seq.

Referenced by generalMatchFactor(), getRdhValue(), massSpecCompareWElut(), printRdhSeq(), rmsdCompare(), and setRdhValue().

```
525 {
526   if (data == NULL || data->seq == NULL || data->seq[0] == NULL)
527     {
528       fprintf (stderr, "Passed bad data to getRdhSeqLength -- error!");
529       fflush (stderr);
530       exit (0);
531     }
532   return data->seq[0]->size2;
533 }
```

C.19.0.250 int getRdhIndexSeqPos (rdh_t * *data*, int *index*, int * *seq*, int * *pos*)

This function is used to access and change the sequence and position values, given an index. The function takes four parameters: a pointer to the real data holder, *data*, an integer *index*, a pointer integer *seq*, and a pointer integer *pos*.

Definition at line 633 of file realIo.c.

References rdh_t::indexSize, rdh_t::indexToPos, and rdh_t::indexToSeq.

Referenced by generalMatchFactor(), makeAlternateCentroid(), massSpecCompareWElut(), outputRealPats(), outputRealPatsWCentroid(), realComparison(), and rmsdCompare().

```
634 {
635   if (data == NULL || data->indexToSeq == NULL || data->indexToPos == NULL
636       || index > data->indexSize)
637     {
638       fprintf (stderr, "Passed bad data to getRdhIndexSeqPos -- error!");
639       fflush (stderr);
640       exit (0);
641     }
642
643   /*
644      printf("Setting index %d -> %d, %d\n", index, seqNo, posNo);
645    */
646   /*
647      fflush(stdout);
648    */
649   *seq = data->indexToSeq[index];
650   *pos = data->indexToPos[index];
651   return 0;
652 }
```

C.19.0.251 char* getRdhLabel (rdh_t * *data*, int *seqNo*)

This function is used to retrieve the label of a particular sequence in a real data holder object. The function takes two parameters: a pointer to the real data holder *data*; and an integer which is the sequence number to be accessed *seqNo*. The function returns a pointer to a string, which is the label for that sequence.

Definition at line 689 of file realIo.c.

References rdh_t::label.

Referenced by printRdhSeq().

```
690 {
691   if (data == NULL || data->label == NULL || data->label[seqNo] == NULL)
692     {
693       fprintf (stderr, "Passed bad data to getRdhLabel -- error!");
694       fflush (stderr);
695       exit (0);
696     }
697   return data->label[seqNo];
698 }
```

C.19.0.252 int getRdhSeqLength (rdh_t * *data*, int *seqNo*)

This function returns an integer that is equal to the sequence length of a particular sequence within the real data holder object. The function takes two parameters: a pointer to the real data holder, *data*, and the index of the sequence for which we need to know the length, *seqNo*.

Definition at line 331 of file realIo.c.

References rdh_t::seq.

Referenced by getRdhValue(), initRdhIndex(), printRdhSeq(), and setRdhValue().

```
332 {
333   if (data == NULL || data->seq == NULL || data->seq[seqNo] == NULL)
334     {
335       fprintf (stderr, "Passed bad data to getRdhSeqLength -- error!");
336       fflush (stderr);
337       exit (0);
338     }
339   return data->seq[seqNo]->size1;
340 }
```

C.19.0.253 double getRdhValue (rdh_t * *data*, int *seqNo*, int *posNo*, int *dimNo*)

This function is used to retrieve the value of a particular dimension, position, and sequence. The function takes four parameters: a pointer to the real data holder *data*; an integer which is the sequence number to be accessed *seqNo*; an integer that is the position number to be accessed *posNo*; and an integer that is the dimension to be accessed *dimNo*.

Definition at line 666 of file realIo.c.

References getRdhDim(), getRdhSeqLength(), and rdh_t::seq.

Referenced by printRdhSeq().

```
667 {
668   if (data == NULL || data->seq == NULL || data->seq[seqNo] == NULL
669       || posNo > getRdhSeqLength (data, seqNo) || dimNo > getRdhDim (data))
670     {
671       fprintf (stderr, "Passed bad data to getRdhValue -- error!");
672       fflush (stderr);
673       exit (0);
674     }
675   return gsl_matrix_get (data->seq[seqNo], posNo, dimNo);
676 }
```

C.19.0.254 rdh_t* initRdh (int *x*)

This function initializes a real data holder object. The function takes as its input a size x which is the number of sequences that will be stored in the object. The function returns a pointer to the object, which has been allocated the correct amount of memory.

Definition at line 277 of file realIo.c.

References rdh_t::indexSize, rdh_t::indexToPos, rdh_t::indexToSeq, rdh_t::label, rdh_t::seq, and rdh_t::size.

Referenced by parseRealData().

```
278 {
279   int i;
280   rdh_t *data = NULL;
281
282   // Allocate space for our structure
283   data = (rdh_t *) malloc (sizeof (rdh_t));
284   if (data == NULL)
285     {
286       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
287       fflush (stderr);
288       exit (0);
289     }
290   data->size = x;
291
292   // Index has to be initialized later, once
293   // we know the word size.
294   data->indexSize = 0;
295   data->indexToSeq = NULL;
296   data->indexToPos = NULL;
297
298   /*
299      data->indexSize = y;
300    */
301   data->label = (char **) malloc (data->size * sizeof (char *));
302   if (data->label == NULL)
303     {
304       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
305       fflush (stderr);
306       exit (0);
307     }
308   data->seq = (gsl_matrix **) malloc (data->size * sizeof (gsl_matrix *));
309   if (data->seq == NULL)
310     {
311       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
312       fflush (stderr);
313       exit (0);
314     }
315   for (i = 0; i < data->size; i++)
316     {
317       data->label[i] = NULL;
318       data->seq[i] = NULL;
319     }
320   return data;
321 }
```

C.19.0.255 int initRdhGslMat (rdh_t * *data*, int *seqNo*, int *x*, int *y*)

This function is used to initialize the memory for the matrix in which the real value to data are stored. To store these data, we use the GNU scientific library. The function takes four parameters: a pointer to the real data holder *data*; an integer, which is the sequence number to be set *seqNo*; an integer, which is the first dimension of the matrix size x; and an integer, which is the second dimension of the matrix size y;

Definition at line 829 of file realIo.c.

References rdh_t::seq.

Referenced by pushOnRdhSeq().

```
830 {
831   data->seq[seqNo] = gsl_matrix_alloc (x, y);
832   if (data->seq[seqNo] == NULL)
```

```
833     {
834       return 0;
835     }
836   else
837     {
838       return 1;
839     }
840 }
```

C.19.0.256 int initRdhIndex (rdh_t * *data*, int *wordSize*, int *seqGap*)

This function is used to initialize the two indices inside a real data holder. The function takes as its input three parameters a pointer to the real data holder, *data*, the size of the words to be compared during the comparison stage *wordSize*, and an integer *seqGap*, which is used to place empty data between unique sequences, such that we do not convolve from one sequence into another during the convolution stage.

Definition at line 358 of file realIo.c.

References getRdhSeqLength(), rdh_t::indexSize, rdh_t::indexToPos, rdh_t::indexToSeq, rdh_t::offsetToIndex, and rdh_t::size.

Referenced by realComparison().

```
359 {
360   int i, j, k;
361   int numWindows = 0;
362   int thisNumWindows;
363   int numSeq;
364   int seqLen = 0;
365
366   // The number of sequences
367   numSeq = data->size;
368
369   // Allocate offsetToIndex's outer structure
370   data->offsetToIndex = (int **) malloc (numSeq * sizeof (int *));
371   if (data->offsetToIndex == NULL)
372     {
373       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
374       fflush (stderr);
375       exit (0);
376     }
377
378   // For each sequence
379   for (i = 0; i < numSeq; i++)
380     {
381
382       // How many windows are in this sequence
383       seqLen = getRdhSeqLength (data, i);
384       numWindows += seqLen - wordSize + 1;
385
386       // And also use this to further allocate offsetToIndex
387       data->offsetToIndex[i] =
388     (int *) malloc ((seqLen - wordSize + 1) * sizeof (int));
389       if (data->offsetToIndex[i] == NULL)
390     {
391       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
392       fflush (stderr);
393       exit (0);
394     }
395     }
```

```
396
397   // One index for each word plus seqGap between each sequence
398   // and a gap at the end
399   data->indexSize = numWindows + numSeq * seqGap;
400
401   // Allocate indexToSeq
402   // NOTE that it should be size of int, not int *... I think we got
403   // fortunate in the previous revision because they are the same
404   // size
405   data->indexToSeq = (int *) malloc (data->indexSize * sizeof (int));
406   if (data->indexToSeq == NULL)
407     {
408       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
409       fflush (stderr);
410       exit (0);
411     }
412
413   // Allocate indexToPos
414   // See above for int vs. int* argument.
415   data->indexToPos = (int *) malloc (data->indexSize * sizeof (int));
416   if (data->indexToPos == NULL)
417     {
418       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
419       fflush (stderr);
420       exit (0);
421     }
422
423   // Fill in the values
424   k = 0;
425   for (i = 0; i < numSeq; i++)
426     {
427
428       // How many windows are in this sequence?
429       thisNumWindows = getRdhSeqLength (data, i) - wordSize + 1;
430
431       // For each window, make an entry in the indexToSeq
432       // and indexToPos and offsetToIndex
433       for (j = 0; j < thisNumWindows; j++)
434     {
435       data->indexToSeq[k] = i;
436       data->indexToPos[k] = j;
437       data->offsetToIndex[i][j] = k;
438       k++;
439     }
440
441       // Add gaps between sequences in the index.
442       // Usually seqGap is just 1;
443       for (j = 0; j < seqGap; j++)
444     {
445
446       // -1 means no sequence and no position
447       data->indexToSeq[k] = -1;
448       data->indexToPos[k] = -1;
449       k++;
450     }
451     }
452   return 0;
453 }
```

C.19.0.257 int makeAlternateCentroid (rdh_t * *data*, cll_t * *curCliq*, int * *candidates*)

This function is used to choose an alternate centroid for a given clique. In order to make the centroid decision slightly less dependent on input order, we decide to choose

from the tied candidates the one whose relative position in the sequence is highest. There is no basis in theory for this, it is done so that a consistent choice is made. Only rarely will two spectra be tied for being a centroid and have the same sequence number. In that case, we pretty much have to default to the sequence number, which is what would be done without this function. Note that now though we are less sensitive to the order of input of the sequences, we are now more sensitive to the context surrounding a given spectrum. That is, if it is put in the beginning of the sequence, it is more likely to be chosen. This choice can only be justified insofar as if multiple choices are tied, then they are the same cumulative distance to the clique, and so *any* should be allowed to be chosen equally. There should be little difference in terms of tangible results. This just makes the semantics consistent.

Definition at line 1202 of file realIo.c.

References getRdhIndexSeqPos(), cSet_t::members, and cnode::set.

Referenced by outputRealPatsWCentroid().

```
1203 {
1204   int indmin, min, i;
1205   int curSeq, curPos;
1206   int numCandidates = candidates[0];
1207   indmin = candidates[1];
1208   getRdhIndexSeqPos (data, curCliq->set->members[indmin], &curSeq, &curPos);
1209   min = curPos;
1210
1211   // We use less-than-or-equal here because we're starting at 1,
1212   // so we want 1 to end.  The length of candidates is one more than
1213   // the maxSup, so we know we can reach candidates[maxSup] without
1214   // a segfault.
1215   for (i = 2; i <= numCandidates; i++)
1216     {
1217       getRdhIndexSeqPos (data, curCliq->set->members[candidates[i]], &curSeq,
1218             &curPos);
1219       if (curPos < min)
1220     {
1221       indmin = candidates[i];
1222       min = curPos;
1223     }
1224     }
1225   return (indmin);
1226 }
```

C.19.0.258 int outputRealPats (rdh_t * *data*, cll_t * *allPats*, int *L*, FILE * *OUTPUT_FILE*, int ** *d*)

This function is used to print out motifs discovered by Gemoda in an attractive fashion. The function takes five parameters: a pointer to a real data holder object *data*; a pointer to a linked list of motifs *allPats*; an integer which is Gemoda's input parameter *L*; and a pointer to a file handle to which output is printed *OUTPUT_FILE*.

Definition at line 1046 of file realIo.c.

References getRdhIndexSeqPos(), cnode::length, cSet_t::members, cnode::next, rdh_t::seq, cnode::set, cSet_t::size, and cnode::stat.

Referenced by main().

```
1048 {
1049   int i, j, pos1;
1050   int curSeq, curPos;
1051   cll_t *curCliq = NULL;
1052   curCliq = allPats;
1053   i = 0;
1054   while (curCliq != NULL)
1055     {
1056       fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d\t", i,
1057            curCliq->length + L, curCliq->set->size);
1058       if (d != NULL)
1059     {
1060       fprintf (OUTPUT_FILE, "\tsignif=%le\n", curCliq->stat);
1061     }
1062       else
1063     {
1064       fprintf (OUTPUT_FILE, "\n");
1065     }
1066       for (j = 0; j < curCliq->set->size; j++)
1067     {
1068       pos1 = curCliq->set->members[j];
1069       getRdhIndexSeqPos (data, pos1, &curSeq, &curPos);
1070       fprintf (OUTPUT_FILE, "   %d\t%d\t", curSeq, curPos);
1071       fprintf (OUTPUT_FILE, "%lf\t",
1072            gsl_matrix_get (data->seq[curSeq], curPos, 0));
1073
1074       /*
1075          for(k=curPos ; k<curPos+curCliq->length+L ; k++){ fprintf(OUTPUT_FILE, "%c",
1076          mySequences[curSeq].seq[k]); }
1077        */
1078       fprintf (OUTPUT_FILE, "\n");
1079     }
1080       fprintf (OUTPUT_FILE, "\n\n");
1081       curCliq = curCliq->next;
1082       i++;
1083     }
1084   return 0;
1085 }
```

C.19.0.259 int outputRealPatsWCentroid (rdh_t * *data*, cll_t * *allPats*, int *L*, FILE * *OUTPUT_FILE*, double * *extraParams*, int *compFunc*)

This function is used to output real valued patterns in a format such that they are centered on a particular centroid.

Definition at line 1233 of file realIo.c.

References findCliqueCentroid(), getCompFunc, getRdhIndexSeqPos(), cnode::length, makeAlternateCentroid(), cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by main().

```
1236 {
1237   int i, j, k, pos1, centroid;
1238   int curSeq, curPos;
1239   int maxSup = 0;
```

```
1240   cll_t *curCliq = NULL;
1241   double mfToCentroid = 0;
1242   double (*comparisonFunc) (rdh_t *, int, int, int, double *) = NULL;
1243   int *candidates = NULL;
1244   curCliq = allPats;
1245   while (curCliq != NULL)
1246     {
1247       if (curCliq->set->size > maxSup)
1248     {
1249       maxSup = curCliq->set->size;
1250     }
1251       curCliq = curCliq->next;
1252     }
1253   candidates = (int *) malloc ((maxSup + 1) * sizeof (int));
1254   if (candidates == NULL)
1255     {
1256       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
1257       fflush (stderr);
1258       exit (0);
1259     }
1260   for (i = 0; i <= maxSup; i++)
1261     {
1262       candidates[i] = 0;
1263     }
1264   comparisonFunc = getCompFunc (compFunc);
1265   curCliq = allPats;
1266   i = 0;
1267   while (curCliq != NULL)
1268     {
1269       fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d\n", i,
1270            curCliq->length + L, curCliq->set->size);
1271       centroid =
1272     findCliqueCentroid (data, curCliq, L, compFunc, extraParams,
1273                 candidates);
1274       if (centroid < 0)
1275     {
1276       centroid = makeAlternateCentroid (data, curCliq, candidates);
1277
1278       // fprintf(OUTPUT_FILE, "WARNING: No single node in"
1279       // " cluster has non-zero similarity to all other\n nodes"
1280       // " in cluster; centroid set to first node.\n");
1281       // centroid = 0;
1282     }
1283       for (j = 0; j < curCliq->set->size; j++)
1284     {
1285       pos1 = curCliq->set->members[j];
1286       getRdhIndexSeqPos (data, pos1, &curSeq, &curPos);
1287       fprintf (OUTPUT_FILE, "   %d\t%d\t", curSeq, curPos);
1288
1289       // fprintf(OUTPUT_FILE, "%lf\t",
1290       // gsl_matrix_get(data->seq[curSeq],curPos,0));
1291       mfToCentroid =
1292         comparisonFunc (data, curCliq->set->members[j],
1293                 curCliq->set->members[centroid], L, extraParams);
1294       fprintf (OUTPUT_FILE, "%lf\t", mfToCentroid);
1295
1296       /*
1297          for(k=curPos ; k<curPos+curCliq->length+L ; k++){ fprintf(OUTPUT_FILE, "%c",
1298          mySequences[curSeq].seq[k]); }
1299        */
1300       fprintf (OUTPUT_FILE, "\n");
1301     }
1302       fprintf (OUTPUT_FILE, "\n\n");
1303       curCliq = curCliq->next;
1304       i++;
1305       for (k = 0; k <= maxSup; k++)
1306     {
1307       candidates[k] = 0;
```

```
1308     }
1309     }
1310   free (candidates);
1311   return 0;
1312 }
```

C.19.0.260 rdh_t* parseRealData (char ** *buf*, int *nl*, char *sep*, int *numSeq*, int *dim*)

This function is used to parse a single line of a fastA formatted input buffer containing real valued data. The function takes

parameters: a pointer to an array of pointers to characters, which stores the sequences that we will read from *buf*; an integer, which is the line in the buffer on which we should start *nl*; a single character, which is used to delimit the input data *sep*; an integer which is the number of the sequence that we are currently reading in *numSeq*; an integer that is the dimensionality of the input data *dim*;

Definition at line 933 of file realIo.c.

References countTotalFields(), initRdh(), and pushOnRdhSeq().

Referenced by readRealData().

```
934 {
935   int i;
936   int seqNo = -1;
937   int totalNumFields;
938   rdh_t *data = NULL;
939   totalNumFields = countTotalFields (buf, nl, sep);
940
941   /*
942      data = initRdh(numSeq, totalNumFields + numSeq - 1);
943    */
944   data = initRdh (numSeq);
945
946   // We're going to add an empty index between
947   // windows that correspond to different
948   // sequences
949
950   // Fast forward to the first sequence
951   i = 0;
952   while (i < nl)
953     {
954
955       // Hit a new sequence
956       if (buf[i][0] == '>')
957     {
958       seqNo++;      // Note that seqNo started at -1!
959       pushOnRdhSeq (data, buf, i, dim, sep);
960       i += dim + 1;
961     }
962       else
963     {
964       i++;
965     }
966     }
967
968   /*
969      printRdhSeq(data, 0, stdout);
970    */
```

```
971   return data;
972 }
```

C.19.0.261 int printRdhSeq (rdh_t * *data*, int *seqNo*, FILE * *FH*)

This function is used to print out a real valued data sequence in a pretty manner. The function takes three parameters: a pointer to the real data holder *data*; an integer which is the sequence to be printed out *seqNo*; and a pointer to a file handle which is where the output will be printed *FH*.

Definition at line 710 of file realIo.c.

References getRdhDim(), getRdhLabel(), getRdhSeqLength(), and getRdhValue().

```
711 {
712   int i, j;
713   int len;
714   int dim;
715   len = getRdhSeqLength (data, seqNo);
716   dim = getRdhDim (data);
717   fprintf (FH, "%s\n", getRdhLabel (data, seqNo));
718   for (i = 0; i < len; i++)
719     {
720       for (j = 0; j < dim; j++)
721     {
722       fprintf (FH, "%3.1f ", getRdhValue (data, seqNo, i, j));
723     }
724       fprintf (FH, "\n");
725     }
726   return 0;
727 }
```

C.19.0.262 int pushOnRdhSeq (rdh_t * *data*, char ** *buf*, int *startLine*, int *dim*, char *sep*)

This function is used to fill in a real data holder structure as we are reading in the sequences. Notably, this routine uses a few static variables, so it can only be called once and should not be used to alter the real data holder structure later. The function takes five parameters: a pointer to the real data holder *data*; a pointer to an array of pointers to characters, which stores the sequences that we will read from *buf*; an integer, which is the line in the buffer on which we should start *startLine*; an integer that is the dimensionality of the input data *dim*; a single character, which is used to delimit the input data *sep*;

Definition at line 863 of file realIo.c.

References countFields(), initRdhGslMat(), setRdhColFromString(), and setRdhLabel().

Referenced by parseRealData().

```
864 {
865   int i, j, k;
866   int numFields;
```

```
867
868   // NOTE THAT THESE ARE STATIC VARIABLES!!!!!
869   // That is, they retain their last value on
870   // each call to this function!
871   static int seqNo = 0;
872
873   /*
874      static int indexNo=0;
875    */
876   i = startLine;
877
878   // Assume that the sequence has at least
879   // one row (should have called checkRealDataFormat!
880   numFields = countFields (buf[i + 1], sep);
881
882   // Initialize the gsl_matrix object for this
883   // sequence in 'data'
884   //
885   // NOTE THAT WE STORE THE TRANSPOSE OF WHAT'S IN
886   // THE INPUT FILE -- x,y = position x, dimension y
887   initRdhGslMat (data, seqNo, numFields, dim);
888
889   // Set the sequence label
890   setRdhLabel (data, seqNo, buf[i]);
891
892   // Read in 'dim' rows
893   for (j = i + 1, k = 0; j < i + 1 + dim; j++, k++)
894     {
895
896       /*
897          printf("%d\n", countFields(buf[j], sep));
898        */
899
900       // Set the k-th dimension of this sequence
901       // STILL NOTE THE TRANSPOSE!
902       setRdhColFromString (data, seqNo, k, buf[j], sep);
903     }
904
905   /*
906      for ( l=0 ; l<numFields ; l++ ){ setRdhIndex(data, seqNo, l, indexNo); indexNo++;
907      }
908    */
909   seqNo++;
910
911   // Augment indexNo once more to have a -1 between each sequence!
912   /*
913      indexNo++;
914    */
915   return 0;
916 }
```

C.19.0.263 rdh_t* readRealData (FILE * *INPUT*)

This function is used to read in a fasta formatted file containing real value data and store the entire thing and a real data holder object. The function takes one parameter: a pointer to a file handle, which is where the data are read from *INPUT*;

Definition at line 983 of file realIo.c.

References checkRealDataFormat(), parseRealData(), and ReadFile().

Referenced by main().

```
984 {
985   char **buf = NULL;
986   int nl;
987   int i;
988   char sep = ' ';
989   int numSeq = 0;
990   int dimensions = 0;
991   int status = 1;
992   rdh_t *data = NULL;
993
994   // Read the entire INPUT file and put it's
995   // contents into 'buf'.  This function also
996   // alters the contents of the location pointed
997   // to by &nl.  Now nl is the number of lines
998   // in the file (or the size of the buff array.
999   buf = ReadFile (INPUT, &nl);
1000   if (buf == NULL)
1001     {
1002       return NULL;
1003     }
1004   status = checkRealDataFormat (buf, nl, sep, &numSeq, &dimensions);
1005   if (numSeq <= 0 || dimensions <= 0 || status == 0)
1006     {
1007       fprintf (stderr,
1008            "Data file is poorly formatted or no sequences read!\n");
1009       fprintf (stderr,
1010            "Each sequence needs to be the same dimensionality!  QUITTING!\n");
1011       fprintf (stderr, "numSeq = %d, dimensions = %d, status = %d\n", numSeq,
1012            dimensions, status);
1013       exit (EXIT_FAILURE);
1014     }
1015
1016   // From here on, we assume that the sequence file is well-formatted
1017   // to make the code more simple.
1018   data = parseRealData (buf, nl, sep, numSeq, dimensions);
1019
1020   // Free up our buffer
1021   for (i = 0; i < nl; i++)
1022     {
1023       if (buf[i] != NULL)
1024     {
1025       free (buf[i]);
1026     }
1027     }
1028   if (buf != NULL)
1029     {
1030       free (buf);
1031     }
1032   return data;
1033 }
```

C.19.0.264 int setRdhColFromString (rdh_t * *data*, int *seqNo*, int *colNo*, char * *s*, char *sep*)

This function is used to fill in the values of a sequence in a real data holder object by reading them straight from a string, which is assumed to be a series of floating-point values separated by some particular character. The function takes five parameters: a pointer to the real data holder *data*; an integer, which is the sequence number to be set *seqNo*; an integer representing the dimension of the sequence which is to be set *colNo*; a pointer to the string holding the floating-point values *s*; a character, which separates the floating-point values in the string *sep*;

Definition at line 744 of file realIo.c.

References rdh_t::seq, setRdhValue(), and wordToDouble().

Referenced by pushOnRdhSeq().

```
745 {
746   int i;
747   int begin = 0;
748   int end = 0;
749   int status = 0;       // 0 = in sep, 1 = in word
750   int fieldCount = 0;
751   double val;
752
753   // Make sure the string is not null and
754   // the rdh_t gsl_matrix array is not null
755   // and the selected gsl_matrix is not null
756   if (s == NULL || data->seq == NULL || data->seq[seqNo] == NULL)
757     {
758       fprintf (stderr, "Passed bad data to setRdhColFromString -- error!");
759       fflush (stderr);
760       exit (0);
761     }
762
763   // Loop over the length of the string
764   for (i = 0; i < strlen (s); i++)
765     {
766
767       // The previous state was space
768       if (status == 0)
769     {
770
771       // We hit a word
772       if (s[i] != sep)
773         {
774           begin = i;
775           status = 1;
776         }
777       else
778         {           // We hit more space
779           continue;
780         }
781     }
782       else
783     {           // The previous state was word
784       if (s[i] != sep)
785         {
786           continue;
787         }
788       else
789         {           // We hit a space
790           end = i - 1;
791           status = 0;
792           val = wordToDouble (s, begin, end);
793
794           // Go to the gsl_matrix object data->seq[seqNo]
795           // and set the (fieldCount, colNo) = val;
796           setRdhValue (data, seqNo, fieldCount, colNo, val);
797           fieldCount++;
798         }
799     }
800     }
801
802   // At the end, if we were in a word, we have
803   // one more field
804   if (status == 1)
805     {               // We're in a word
```

```
806       val = wordToDouble (s, begin, strlen (s));
807
808       // Added in, MPS 5/3/05 ---
809       // And don't forget to set the RdhValue!
810       setRdhValue (data, seqNo, fieldCount, colNo, val);
811       fieldCount++;
812     }
813   return fieldCount;
814 }
```

C.19.0.265 int setRdhIndex (rdh_t * *data*, int *seqNo*, int *posNo*, int *index*)

This function is used to fill in entries in the indices of the real data holder. The function takes four parameters: a pointer to the real data holder, *data*, an integer specifying the sequence number *seqNo*, an integer specifying the position number within the sequence *posNo*, and an integer specifying what the index for this sequence number and position number should be *index*.

Definition at line 600 of file realIo.c.

References rdh_t::indexSize, rdh_t::indexToPos, and rdh_t::indexToSeq.

```
601 {
602   if (data == NULL || data->indexToSeq == NULL || data->indexToPos == NULL
603       || index > data->indexSize)
604     {
605       fprintf (stderr, "Passed bad data to getRdhValue -- error!");
606       fflush (stderr);
607       exit (0);
608     }
609
610   /*
611      printf("Setting index %d -> %d, %d\n", index, seqNo, posNo);
612    */
613   /*
614      fflush(stdout);
615    */
616   data->indexToSeq[index] = seqNo;
617   data->indexToPos[index] = posNo;
618   return 0;
619 }
```

C.19.0.266 int setRdhLabel (rdh_t * *data*, int *seqNo*, char * *s*)

This function will label a sequence within a real data holder object with a particular string. The function takes two parameters: a pointer to the real data holder, *data*, an integer *seqNo*, and a pointer to a string *s*.

Definition at line 543 of file realIo.c.

References rdh_t::label, and rdh_t::seq.

Referenced by pushOnRdhSeq().

```
544 {
```

```
545   if (data->seq == NULL || data->label == NULL)
546     {
547       fprintf (stderr, "Passed bad data to setRdhLabel -- error!");
548       fflush (stderr);
549       exit (0);
550     }
551   data->label[seqNo] = strdup (s);
552   if (data->label[seqNo] == NULL)
553     {
554       fprintf (stderr, "\nMemory Error allocating label!\n%s\n",
555           strerror (errno));
556       fflush (stderr);
557       exit (0);
558     }
559   return 0;
560 }
```

C.19.0.267 int setRdhValue (rdh_t * *data*, int *seqNo*, int *posNo*, int *dimNo*, double *val*)

This function will set a particular dimension at a particular position within a specified sequence to a user supplied value. The function takes five parameters: a pointer to the real data holder, *data*, an integer *seqNo* which is the sequence which needs its value set, two integers that specify the position number and the dimension number that needs to be set, and finally a double precision floating point number which is the value to which the the data should be set.

Definition at line 575 of file realIo.c.

References getRdhDim(), getRdhSeqLength(), and rdh_t::seq.

Referenced by setRdhColFromString().

```
576 {
577   if (data == NULL || data->seq == NULL || data->seq[seqNo] == NULL
578       || posNo > getRdhSeqLength (data, seqNo) || dimNo > getRdhDim (data))
579     {
580       fprintf (stderr, "Passed bad data to setRdhValue -- error!");
581       fflush (stderr);
582       exit (0);
583     }
584   gsl_matrix_set (data->seq[seqNo], posNo, dimNo, val);
585   return 0;
586 }
```

C.19.0.268 wordToDouble (char * *s*, int *begin*, int *end*)

Turn the substring of s starting at char s[begin] and ending at s[end] int a double. INPUT: a string s, integer begin, and integer end. OUTPUT: a double. NOTE: Throws an error and dies if there's a problem making the double from the substring. No room for ill-formated data files. double

Definition at line 30 of file realIo.c.

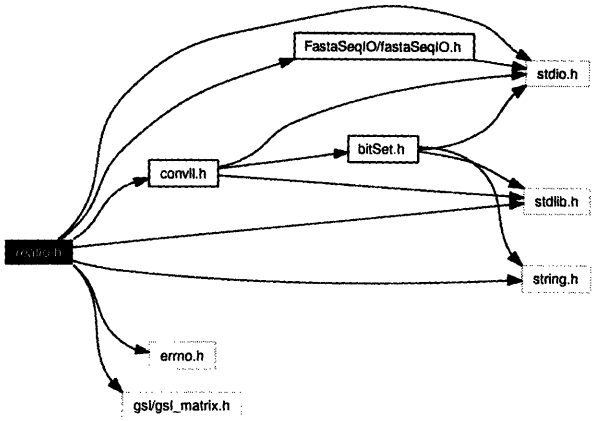
Referenced by countFields(), and setRdhColFromString().

```
31 {
32   char *str = NULL;
33   char *endptr;
34   double val;
35   int size;
36   int memsize;
37
38   // Check for a sane substring
39   if (end - begin <= 0)
40     {
41       fprintf (stderr, "\nInvalid argument to wordToDouble!\n");
42       fflush (stderr);
43       exit (0);
44     }
45
46   // Get the required string size
47   memsize = end - begin + 2;    // An extra space in mem for null-termination
48   size = end - begin + 1;
49
50   // Get memory for a temporary string
51   str = (char *) malloc (memsize * sizeof (char));
52   if (str == NULL)
53     {
54       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
55       fflush (stderr);
56       exit (0);
57     }
58
59   // Make sure the string ends with a null char
60   str[size] = '\0';
61
62   // Copy the word into str
63   str = strncpy (str, s + begin, size);
64
65   // Set endptr to str as initial value
66   endptr = str;
67   val = strtod (str, &endptr);
68
69   // endptr should point to the last char
70   // used in the conversion if strtod worked
71   if (val == 0 && endptr == str)
72     {
73       fprintf (stderr, "\nError making double from string: %s\n", str);
74       fflush (stderr);
75       exit (0);
76     }
77   free (str);
78   return val;
79 }
```

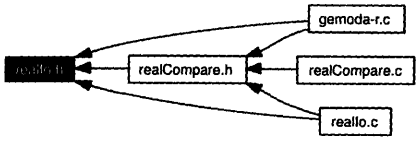
C.20 realIo.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <gsl/gsl_matrix.h>
#include "FastaSeqIO/fastaSeqIO.h"
#include "convll.h"
```

Include dependency graph for realIo.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct rdh_t

Functions

- rdh_t * readRealData (FILE *INPUT)

- rdh_t * freeRdh (rdh_t *data)
- int initRdhIndex (rdh_t *data, int wordSize, int seqGap)
- int getRdhIndexSeqPos (rdh_t *data, int index, int *seq, int *pos)
- int getRdhDim (rdh_t *data)
- int outputRealPats (rdh_t *data, cll_t *allPats, int L, FILE *OUTPUT_FILE, int **d)
- int outputRealPatsWCentroid (rdh_t *data, cll_t *allPats, int L, FILE *OUTPUT_FILE, double *extraParams, int compFunc)

Function Documentation

C.20.0.269 rdh_t* freeRdh (rdh_t * *data*)

This function returns a null pointer after freeing the memory associated with a real data holder object. The function takes one parameter: a pointer to the real data holder, *data*.

Definition at line 396 of file realIo.c.

References rdh_t::indexToPos, rdh_t::indexToSeq, rdh_t::label, rdh_t::offsetToIndex, rdh_t::seq, and rdh_t::size.

Referenced by main().

C.20.0.270 int getRdhDim (rdh_t * *data*)

This function returns an integer equal to the dimensions of the data stored in a real data holder object. The function takes one parameter: a pointer to the real data holder, *data*.

Definition at line 447 of file realIo.c.

References rdh_t::seq.

Referenced by generalMatchFactor(), getRdhValue(), massSpecCompareWElut(), printRdhSeq(), rmsdCompare(), and setRdhValue().

C.20.0.271 int getRdhIndexSeqPos (rdh_t * *data*, int *index*, int * *seq*, int * *pos*)

This function is used to access and change the sequence and position values, given an index. The function takes four parameters: a pointer to the real data holder, *data*, an integer *index*, a pointer integer *seq*, and a pointer integer *pos*.

Definition at line 544 of file realIo.c.

References rdh_t::indexSize, rdh_t::indexToPos, and rdh_t::indexToSeq.

Referenced by generalMatchFactor(), makeAlternateCentroid(), massSpecCompareWElut(), outputRealPats(), outputRealPatsWCentroid(), realComparison(), and rmsdCompare().

C.20.0.272 int initRdhIndex (rdh_t * *data*, int *wordSize*, int *seqGap*)

This function is used to initialize the two indices inside a real data holder. The function takes as its input three parameters a pointer to the real data holder, *data*, the size of the words to be compared during the comparison stage *wordSize*, and an integer *seqGap*, which is used to place empty data between unique sequences, such that we do not convolve from one sequence into another during the convolution stage.

Definition at line 307 of file realIo.c.

References getRdhSeqLength(), rdh_t::indexSize, rdh_t::indexToPos, rdh_t::indexToSeq, rdh_t::offsetToIndex, and rdh_t::size.

Referenced by realComparison().

C.20.0.273 int outputRealPats (rdh_t * *data*, cll_t * *allPats*, int *L*, FILE * *OUTPUT_FILE*, int ** *d*)

This function is used to print out motifs discovered by Gemoda in an attractive fashion. The function takes five parameters: a pointer to a real data holder object *data*; a pointer to a linked list of motifs *allPats*; an integer which is Gemoda's input parameter *L*; and a pointer to a file handle to which output is printed *OUTPUT_FILE*.

Definition at line 904 of file realIo.c.

References getRdhIndexSeqPos(), cnode::length, cSet_t::members, cnode::next, rdh_t::seq, cnode::set, cSet_t::size, and cnode::stat.

Referenced by main().

C.20.0.274 int outputRealPatsWCentroid (rdh_t * *data*, cll_t * *allPats*, int *L*, FILE * *OUTPUT_FILE*, double * *extraParams*, int *compFunc*)

This function is used to output real valued patterns in a format such that they are centered on a particular centroid.

Definition at line 1068 of file realIo.c.

References findCliqueCentroid(), getCompFunc, getRdhIndexSeqPos(), cnode::length, makeAlternateCentroid(), cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by main().

C.20.0.275 rdh_t* readRealData (FILE * *INPUT*)

This function is used to read in a fasta formatted file containing real value data and store the entire thing and a real data holder object. The function takes one parameter: a pointer to a file handle, which is where the data are read from *INPUT*;

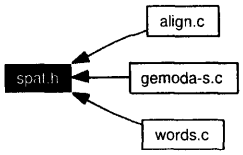
Definition at line 850 of file realIo.c.

References checkRealDataFormat(), parseRealData(), and ReadFile().

Referenced by main().

C.21 spat.h File Reference

This graph shows which files directly or indirectly include this file:



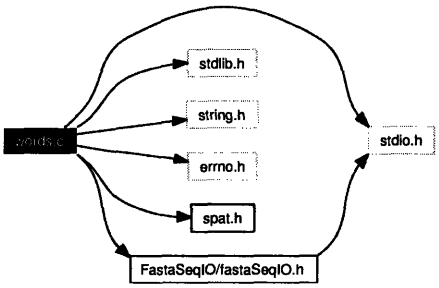
Data Structures

- struct sOffset_t
- struct sPat_t

C.22 words.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "spat.h"
#include "FastaSeqIO/fastaSeqIO.h"
```

Include dependency graph for words.c:



Data Structures

- struct sHashEntry_t
- struct sHash_t

Defines

- #define SHASH_MAX_KEY_SIZE 1000

Functions

- int sieve3 (long n)
- unsigned long hash1 (unsigned char *str)
- int hashpjw (char *s)
- sHash_t initSHash (int n)
- sHashEntry_t * searchSHash (sHashEntry_t *newEntry, sHash_t *thisHash, int create)
- int destroySHash (sHash_t *thisHash)
- int printSHash (sHash_t *thisHash, FILE *FH)

- int printSPats (sPat_t *a, int n)
- int destroySPatA (sPat_t *words, int wc)
- sPat_t * countWords2 (fSeq_t *seq, int numSeq, int L, int *numWords)

Detailed Description

This file defines functions that are used in the processing of string based sequences. There are a number of functions defined in this file better used for hashing strings so that the comparison phase can be sped up by only comparing unique words. Heuristically, we have noticed that for sequences in which there is a large degree of redundancy these hashing functions can significantly speed up the comparison phase.

Definition in file words.c.

Define Documentation

C.22.0.276 #define SHASH_MAX_KEY_SIZE 1000

Definition at line 192 of file words.c.

Referenced by printSHash(), and searchSHash().

Function Documentation

C.22.0.277 sPat_t* countWords2 (fSeq_t * *seq*, int *numSeq*, int *L*, int * *numWords*)

Counts words of size L in the input FastA sequences, hashes all of the words, and returns an array of sPat_t objects.

Definition at line 373 of file words.c.

References sHashEntry_t::data, destroySHash(), sHashEntry_t::idx, initSHash(), sHashEntry_t::key, sHashEntry_t::L, sPat_t::length, sOffset_t::next, sPat_t::offset, sOffset_t::pos, sOffset_t::prev, searchSHash(), sOffset_t::seq, sieve3(), sPat_t::string, and sPat_t::support.

Referenced by main().

```
374 {
375   int i, j;
376   int totalChars = 0;
377   int hashSize;
378   sHashEntry_t newEntry;
379   sHashEntry_t *ep;
380   sHash_t wordHash;
381   sPat_t *words = NULL;
382   int wc = 0;
383   int prev = -1;
```

```
384   int l;
385
386
387   // Count the total number of characters.  This
388   // is the upper limit on how many words we can have
389   for (i = 0; i < numSeq; i++)
390     {
391       totalChars += strlen (seq[i].seq);
392     }
393
394   // Get a prime number for the size of the hash table
395   hashSize = sieve3 ((long) (2 * totalChars));
396   wordHash = initSHash (hashSize);
397
398   // Chop up each sequence and hash out the words of size L
399   for (i = 0; i < numSeq; i++)
400     {
401       prev = -1;
402
403       // skip sequences that are too short to have
404       // a pattern
405       if (strlen (seq[i].seq) < L)
406     {
407       continue;
408     }
409       for (j = 0; j < strlen (seq[i].seq) - L + 1; j++)
410     {
411
412       // Make a hash table entry for this word
413       newEntry.key = &(seq[i].seq[j]);
414       newEntry.data = 1;
415       newEntry.idx = wc;
416       newEntry.L = L;
417
418       // Check to see if it's already in the hash table
419       ep = searchSHash (&newEntry, &wordHash, 0);
420       if (ep == NULL)
421         {
422
423           // If it's not, create an entry for it
424           ep = searchSHash (&newEntry, &wordHash, 1);
425
426           // Increase the size of our word array
427           words = (sPat_t *) realloc (words, (wc + 1) * sizeof (sPat_t));
428           if (words == NULL)
429         {
430           fprintf (stderr, "Error!\n");
431           fflush (stderr);
432         }
433           // Add the new word
434           words[wc].string = &(seq[i].seq[j]);
435           words[wc].length = L;
436           words[wc].support = 1;
437           words[wc].offset =
438         (sOffset_t *) malloc (1 * sizeof (sOffset_t));
439           if (words[wc].offset == NULL)
440         {
441           fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
442           fflush (stderr);
443           exit (0);
444         }
445           words[wc].offset[0].seq = i;
446           words[wc].offset[0].pos = j;
447           words[wc].offset[0].prev = prev;
448           words[wc].offset[0].next = -1;
449
450           if (prev != -1)
451         {
```

```
452           words[prev].offset[words[prev].support - 1].next = wc;
453         }
454           prev = wc;
455           wc++;
456
457         }
458       else
459         {
460
461           // If it is, increase the count for this word
462           ep->data++;
463
464           // add a new offset to the word array
465           l = words[ep->idx].support;
466           words[ep->idx].offset =
467         (sOffset_t *) realloc (words[ep->idx].offset,
468                                (l + 1) * sizeof (sOffset_t));
469           words[ep->idx].offset[l].seq = i;
470           words[ep->idx].offset[l].pos = j;
471           words[ep->idx].offset[l].prev = prev;
472           words[ep->idx].offset[l].next = -1;
473
474           // Update the next/prev
475           if (prev != -1)
476         {
477           words[prev].offset[words[prev].support - 1].next = ep->idx;
478         }
479           prev = ep->idx;
480
481           // Have to put this down here for cases when we create
482           // a word and it is immeadiately followed by itself!!
483           words[ep->idx].support += 1;
484         }
485     }
486     }
487
488
489   destroySHash (&wordHash);
490   *numWords = wc;
491   return words;
492 }
```

C.22.0.278 int destroySHash (sHash_t * *thisHash*)

Destroy a hash table, freeing the memory.

Definition at line 272 of file words.c.

References sHash_t::hash, sHash_t::hashSize, and sHash_t::iHashSize.

Referenced by countWords2().

```
273 {
274   int i;
275   free (thisHash->iHashSize);
276   free (thisHash->hashSize);
277   for (i = 0; i < thisHash->totalSize; i++)
278     {
279       if (thisHash->hash[i] != NULL)
280     {
281       free (thisHash->hash[i]);
282       thisHash->hash[i] = NULL;
283     }
284     }
```

```
285   if (thisHash->hash != NULL)
286     {
287       free (thisHash->hash);
288       thisHash->hash = NULL;
289     }
290   return 0;
291 }
```

C.22.0.279 int destroySPatA (sPat_t * *words*, int *wc*)

This function is used to free up the memory allocated in an array of sPat_t space objects. The function returns a null pointer.

Definition at line 352 of file words.c.

References sPat_t::offset.

```
353 {
354   int i;
355   for (i = 0; i < wc; i++)
356     {
357       if (words[i].offset != NULL)
358     {
359       free (words[i].offset);
360       words[i].offset = NULL;
361     }
362     }
363   free (words);
364   words = NULL;
365   return 0;
366 }
```

C.22.0.280 unsigned long hash1 (unsigned char * *str*)

A hashing function that returns an integer, given a pointer to a null characterterminated string.

Definition at line 73 of file words.c.

Referenced by searchSHash().

```
74 {
75   unsigned long hash = 5381;
76   int c;
77
78   while ((c = *str++))
79     hash = ((hash << 5) + hash) + c;    /* hash * 33 + c */
80
81   return hash;
82 }
```

C.22.0.281 int hashpjw (char * *s*)

A hashing function that returns an integer, given a pointer to a null characterterminated string.

Definition at line 89 of file words.c.

```
90 {
91   char *p;
92   unsigned int h, g;
93
94   h = 0;
95   for (p = s; *p != '\0'; p++)
96     {
97       h = (h << 4) + *p;
98       if ((g = h & 0xF0000000))
99     {
100       h ^= g >> 24;
101       h ^= g;
102     }
103     }
104   return h;
105 }
```

C.22.0.282 sHash_t initSHash (int *n*)

Allocates the memory for a sHash table and initializes some of the elements.

Definition at line 155 of file words.c.

References sHash_t::totalSize.

Referenced by countWords2().

```
156 {
157   int i = 0;
158   int step = 0;
159   sHash_t this;
160
161   this.totalSize = n;
162   this.hashSize = (int *) malloc (n * sizeof (int));
163   if (this.hashSize == NULL)
164     {
165       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
166       fflush (stderr);
167       exit (0);
168     }
169   this.iHashSize = (int *) malloc (n * sizeof (int));
170   if (this.iHashSize == NULL)
171     {
172       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
173       fflush (stderr);
174       exit (0);
175     }
176   this.hash = (sHashEntry_t **) malloc (n * sizeof (sHashEntry_t *));
177   if (this.hash == NULL)
178     {
179       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
180       fflush (stderr);
181       exit (0);
182     }
183   for (i = 0; i < n; i++)
184     {
185       this.hash[i] = NULL;
186       this.hashSize[i] = 0;
187       this.iHashSize[i] = step;
188     }
189   return this;
190 }
```

C.22.0.283 int printSHash (sHash_t * *thisHash*, FILE * *FH*)

This function is used to print the hash out and is generally only used for error checking.

Definition at line 298 of file words.c.

References sHashEntry_t::data, sHash_t::hash, sHashEntry_t::key, sHashEntry_t::L, and SHASH_MAX_KEY_SIZE.

```
299 {
300   int i, j;
301   char string[SHASH_MAX_KEY_SIZE];
302
303   for (i = 0; i < thisHash->totalSize; i++)
304     {
305       for (j = 0; j < thisHash->hashSize[i]; j++)
306     {
307
308       strncpy (string, thisHash->hash[i][j].key, thisHash->hash[i][j].L);
309       string[thisHash->hash[i][j].L] = '\0';
310       fprintf (FH, "%s %d\n", string, thisHash->hash[i][j].data);
311
312     }
313     }
314   return 0;
315 }
```

C.22.0.284 int printSPats (sPat_t * *a*, int *n*)

This function is used to print out an array of sPat_t objects and is generally only used for error checking.

Definition at line 321 of file words.c.

References sPat_t::length.

```
322 {
323   char *s = NULL;
324   int i, j;
325   int size = 0;
326   for (i = 0; i < n; i++)
327     {
328       if (a[i].length > size)
329     {
330       s = (char *) realloc (s, a[i].length * sizeof (char));
331     }
332       strncpy (s, a[i].string, a[i].length);
333       s[a[i].length] = '\0';
334       printf ("%d:  %s\n", i, s);
335       for (j = 0; j < a[i].support; j++)
336     {
337       printf ("\t%d %d -> (%d, %d)\n", a[i].offset[j].seq,
338           a[i].offset[j].pos, a[i].offset[j].prev,
339           a[i].offset[j].next);
340     }
341       printf ("\n");
342     }
343   free (s);
344   return 0;
345 }
```

C.22.0.285 sHashEntry_t* searchSHash (sHashEntry_t * *newEntry*, sHash_t * *thisHash*, int *create*)

This function has two purposes. It searches for entries in the hash table and it puts new entries in.

Definition at line 198 of file words.c.

References sHash_t::hash, hash1(), sHash_t::hashSize, sHash_t::iHashSize, sHashEntry_t::key, sHashEntry_t::L, SHASH_MAX_KEY_SIZE, and sHash_t::totalSize.

Referenced by countWords2().

```
199 {
200   char string[SHASH_MAX_KEY_SIZE];
201   unsigned long (*hashFunction) () = &hash1;
202   int i, thisIndex;
203   int status = 0;
204
205   // A string to store the key
206   strncpy (string, newEntry->key, newEntry->L);
207   string[newEntry->L] = '\0';
208
209   // The index that this key hashes to
210   thisIndex = hashFunction ((unsigned char *) string) % thisHash->totalSize;
211
212   // For each member that has this index, check to see
213   // if the key is the same
214   for (i = 0; i < thisHash->hashSize[thisIndex]; i++)
215     {
216       if (strncmp (thisHash->hash[thisIndex][i].key, string, newEntry->L) ==
217       0)
218     {
219
220       // We found a match
221       /*
222          printf("\t%s already in hash table!\n");
223        */
224       status = 1;
225       return &(thisHash->hash[thisIndex][i]);
226       break;
227
228     }
229     }
230
231   // If we didn't find the key and we're told to create it,
232   // then allocate new memory for the hashEntry and put it in
233   if (status == 0 && create != 0)
234     {
235
236       // Allocate space for the new entry at this index
237       if (thisHash->iHashSize[thisIndex] == 0)
238     {
239       thisHash->hash[thisIndex] =
240         (sHashEntry_t *) malloc (sizeof (sHashEntry_t));
241     }
242       else
243     {
244       thisHash->hash[thisIndex] =
245         (sHashEntry_t *) realloc (thisHash->hash[thisIndex],
246                       (thisHash->iHashSize[thisIndex] +
247                        1) * sizeof (sHashEntry_t));
248     }
249       if (thisHash->hash[thisIndex] == NULL)
```

```
250     {
251       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
252       fflush (stderr);
253       exit (0);
254     }
255       // Increase our record of the size
256       i = thisHash->hashSize[thisIndex];
257       thisHash->hash[thisIndex][i] = *newEntry;
258       thisHash->iHashSize[thisIndex]++;
259       thisHash->hashSize[thisIndex]++;
260
261
262       // Return a pointer to this entry
263       return &(thisHash->hash[thisIndex][i]);
264     }
265   return NULL;
266 }
```

C.22.0.286 int sieve3 (long *n*)

Prime number generator: returns first prime number equal or less than

Parameters:

n.

Definition at line 27 of file words.c.

Referenced by countWords2().

```
28 {
29   int i, p, j;
30   int *a;
31   a = (int *) malloc ((n + 1) * sizeof (int));
32   if (a == NULL)
33     {
34       fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
35       fflush (stderr);
36       exit (0);
37     }
38   a[0] = 0;
39   a[1] = 0;
40   for (i = 2; i < n; i++)
41     {
42       a[i] = 1;
43     }
44   p = 2;
45   do
46     {
47       j = 2 * p;
48       do
49     {
50       a[j] = 0;
51       j = j + p;
52     }
53       while (j <= n);
54       p = p + 1;
55     }
56   while (p * p < 2 * n);
57   for (i = n; i > 2; i--)
58     {
59       if (a[i])
60     {
```

```
61       free (a);
62       return i;
63     }
64     }
65   free (a);
66   return 0;
67 }
```

Appendix D

Gemoda data structure documentation

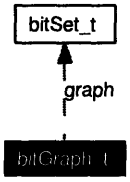
D.1 Introduction

This appendix describes in detail the data structures used in our software implementation of Gemoda, which is described in Chapter 3. We have implemented an object–oriented approach in our program; in this way, we tried to capture some of the benefits of C++ while mainting the easy portability of C. More details and documentation of the files that constitute our implementation of Gemoda can be found in Appendix C.

D.2 bitGraph_t Struct Reference

`#include <bitSet.h>`

Collaboration diagram for bitGraph_t:



Data Fields

- int size
- bitSet_t ** graph

Detailed Description

A bit graph is an array of bit sets. The graph must be of size size x size. This data structure is used to store adjacency matrices. In particular, a bit graph is used in the clustering step. It can easily be considered a set of sets.

Definition at line 48 of file bitSet.h.

Field Documentation

D.2.0.287 bitSet_t** bitGraph_t::graph

A pointer used to store an array of bitSet_t space objects.

Definition at line 56 of file bitSet.h.

Referenced by bitGraphCheckBit(), bitGraphRowIntersection(), bitGraphRowUnion(), bitGraphSetFalse(), bitGraphSetFalseDiagonal(), bitGraphSetFalseSym(), bitGraphSetTrue(), bitGraphSetTrueDiagonal(), bitGraphSetTrueSym(), copyBitGraph(), countBitGraphNonZero(), deleteBitGraph(), emptyBitGraph(), emptyBitGraphRow(), fillBitGraph(), filterIter(), findCliques(), getStatMat(), maskBitGraph(), newBitGraph(), printBitGraph(), pruneBitGraph(), and singleLinkage().

D.2.0.288 int bitGraph_t::size

The total size of a bit graph, which is assumed to be symmetric. There are *size* bit sets in a bit graph, each of size *size*.

Definition at line 53 of file bitSet.h.

Referenced by convolve(), copyBitGraph(), filterGraph(), findCliques(), getStatMat(), main(), newBitGraph(), and oldGetStatMat().

The documentation for this struct was generated from the following file:

- bitSet.h

D.3 bitSet_t Struct Reference

`#include <bitSet.h>`

Data Fields

- int max
- int slots
- int bytes
- bit_t * tf

Detailed Description

A bit set is a data structure for storing set objects that allows for quick set operations such as intersections, unions, differences, and so forth. On a standard 32-bit architecture, 32 operations can be performed at the same time, greatly speeding the clique finding stage of the algorithm.

Definition at line 24 of file bitSet.h.

Field Documentation

D.3.0.289 int bitSet_t::bytes

This variable actually holds the total number of bits, rather than the number of bytes. However, we chose to keep this name rather than make a variety of changes.

Definition at line 37 of file bitSet.h.

Referenced by emptySet(), fillSet(), and newBitSet().

D.3.0.290 int bitSet_t::max

The maximum integer that can be set to true or false.

Definition at line 28 of file bitSet.h.

Referenced by newBitSet(), nextBitBitSet(), setFalse(), and setTrue().

D.3.0.291 int bitSet_t::slots

The total number of slots, where a slot holds a number of bits equal to the size of a bit_t space object.

Definition at line 32 of file bitSet.h.

Referenced by bitSet3WayIntersection(), bitSetDifference(), bitSetIntersection(), bitSetSum(), bitSetUnion(), copySet(), and newBitSet().

D.3.0.292 bit_t* bitSet_t::tf

A pointer to a bit_t, which is used to store an array of these objects.

Definition at line 40 of file bitSet.h.

Referenced by bitSet3WayIntersection(), bitSetDifference(), bitSetIntersection(), bitSetSum(), bitSetUnion(), checkBit(), copySet(), countSet(), deleteBitSet(), emptySet(), fillSet(), flipBits(), newBitSet(), nextBitBitSet(), printBinaryBitSet(), setFalse(), and setTrue().

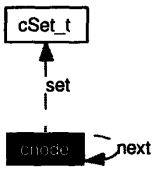
The documentation for this struct was generated from the following file:

- bitSet.h

D.4 cnode Struct Reference

`#include <convll.h>`

Collaboration diagram for cnode:



Data Fields

- cSet_t * set
- int id
- int length
- cnode * next
- double stat

Detailed Description

This data structure is a linked list for storing cliques. Each member of the linked list has a set, an ID number, a length (which gives the number of characters in the motif), a pointer to the next member of the linked list, and a floating-point number for storing statistical information.

Definition at line 35 of file convll.h.

Field Documentation

D.4.0.293 int cnode::id

Identification number for this member.

Definition at line 38 of file convll.h.

Referenced by addToStacks(), printCll(), printCllPattern(), pushCll(), removeSupers(), singleCliqueConv(), sortByStats(), swapNodecSet(), uniqClique(), wholeCliqueConv(), wholeRoundConv(), and yankCll().

D.4.0.294 int cnode::length

Length of this motif.

Definition at line 41 of file convll.h.

Referenced by calcStatCliq(), getLargestLength(), main(), outputRealPats(), outputRealPatsWCentroid(), printCll(), and pushCll().

D.4.0.295 struct cnode* cnode::next

A pointer to the next member, or the next motif.

Definition at line 42 of file convll.h.

Referenced by calcStatAllCliqs(), fillMemberStacks(), getLargestLength(), getLargestSupport(), main(), outputRealPats(), outputRealPatsWCentroid(), popCll(), printCll(), pruneCll(), pushCll(), removeSupers(), singleCliqueConv(), sortByStats(), swapNodecSet(), uniqClique(), wholeRoundConv(), and yankCll().

D.4.0.296 cSet_t* cnode::set

The set for this member of the linked list.

Definition at line 37 of file convll.h.

Referenced by addToStacks(), calcStatCliq(), findCliqueCentroid(), getLargestSupport(), initheadCll(), main(), makeAlternateCentroid(), mergeIntersect(), outputRealPats(), outputRealPatsWCentroid(), popCll(), printCll(), printCllPattern(), pruneCll(), pushCll(), removeSupers(), singleCliqueConv(), swapNodecSet(), uniqClique(), and wholeCliqueConv().

D.4.0.297 double cnode::stat

Used to store the statistical store of a motif.

Definition at line 43 of file convll.h.

Referenced by calcStatAllCliqs(), main(), outputRealPats(), and pushCll().

The documentation for this struct was generated from the following file:

- convll.h

D.5 cSet_t Struct Reference

`#include <convll.h>`

Data Fields

- int size
- int * members

Detailed Description

A cSet_t is used to hold a set of integers, in cases where the upper limit of integers size is unknown. Or, in cases where using a bit set would be impractical. This data structure is used throughout the convolution, where we have found heuristically that intersections of this data type are much faster than those for bitSet_t's, which would require a bit shift.

Definition at line 21 of file convll.h.

Field Documentation

D.5.0.298 int* cSet_t::members

Array of pointers to ints that holds the members of this set.

Definition at line 26 of file convll.h.

Referenced by addToStacks(), bitSetToCSet(), checkCliquecSet(), findCliqueCentroid(), main(), makeAlternateCentroid(), mergeIntersect(), mllToCSet(), outputRealPats(), outputRealPatsWCentroid(), popCll(), printCll(), printCllPattern(), printCSet(), pruneCll(), pushConvClique(), removeSupers(), swapNodecSet(), uniqClique(), and wholeCliqueConv().

D.5.0.299 int cSet_t::size

Number of members in this set.

Definition at line 24 of file convll.h.

Referenced by bitSetToCSet(), calcStatCliq(), checkCliquecSet(), findCliqueCentroid(), getLargestSupport(), main(), mllToCSet(), outputRealPats(), outputRealPatsWCentroid(), printCll(), printCllPattern(), printCSet(), pruneCll(), removeSupers(), singleCliqueConv(), uniqClique(), and wholeCliqueConv().

The documentation for this struct was generated from the following file:

- convll.h

D.6 fSeq_t Struct Reference

`#include <fastaSeqIO.h>`

Data Fields

- char * seq
- char * label

Detailed Description

Definition at line 12 of file fastaSeqIO.h.

Field Documentation

D.6.0.300 char* fSeq_t::label

Definition at line 14 of file fastaSeqIO.h.

Referenced by FreeFSeqs(), initAofFSeqs(), and ReadFSeqs().

D.6.0.301 char* fSeq_t::seq

Definition at line 13 of file fastaSeqIO.h.

Referenced by FreeFSeqs(), initAofFSeqs(), printFSeqSubSeq(), ReadFSeqs(), and ReadTxtSeqs().

The documentation for this struct was generated from the following file:

- FastaSeqIO/fastaSeqIO.h

D.7 mnode Struct Reference

`#include <convll.h>`

Collaboration diagram for mnode:

Data Fields

- int cliqueMembership
- mnode * next

Detailed Description

This data structure is just a link to list of integers used for bookkeeping during the convolution stage.

Definition at line 49 of file convll.h.

Field Documentation

D.7.0.302 int mnode::cliqueMembership

Clique to which this belongs.

Definition at line 52 of file convll.h.

Referenced by mllToCSet(), printMemberStacks(), pushMemStack(), and setStackTrue().

D.7.0.303 struct mnode* mnode::next

A pointer to the next member in the linked list of mll_t space objects.

Definition at line 55 of file convll.h.

Referenced by mllToCSet(), popMemStack(), printMemberStacks(), pushMemStack(), and setStackTrue().

The documentation for this struct was generated from the following file:

- convll.h

D.8 rdh_t Struct Reference

`#include <realIo.h>`

Data Fields

- int size
- int indexSize
- char ** label
- gsl_matrix ** seq
- int * indexToSeq
- int * indexToPos
- int ** offsetToIndex

Detailed Description

This is a data structure, which is used to store real valued data. Basically, this is an array of gsl_matrix objects, where each matrix represents a single, multidimensional array that was read in from a FastA formatted file.

Definition at line 24 of file realIo.h.

Field Documentation

D.8.0.304 int rdh_t::indexSize

The size of the index, where the index is used to store pointers to the different sequences in this object.

Definition at line 30 of file realIo.h.

Referenced by getRdhIndexSeqPos(), initRdh(), initRdhIndex(), realComparison(), and setRdhIndex().

D.8.0.305 int* rdh_t::indexToPos

The array of integers that tell us to which position in a sequence each index in the gsl_matrix array corresponds.

Definition at line 40 of file realIo.h.

Referenced by freeRdh(), getRdhIndexSeqPos(), initRdh(), initRdhIndex(), and setRdhIndex().

D.8.0.306 int* rdh_t::indexToSeq

The array of integers that will tell us to which sequence each index and the gsl_matrix array corresponds.

Definition at line 37 of file realIo.h.

Referenced by freeRdh(), getRdhIndexSeqPos(), initRdh(), initRdhIndex(), main(), and setRdhIndex().

D.8.0.307 char** rdh_t::label

The array of labels that store the names of each sequence.

Definition at line 32 of file realIo.h.

Referenced by freeRdh(), getRdhLabel(), initRdh(), and setRdhLabel().

D.8.0.308 int** rdh_t::offsetToIndex

The array that points from a particular offset to its index.

Definition at line 42 of file realIo.h.

Referenced by freeRdh(), initRdhIndex(), and main().

D.8.0.309 gsl_matrix** rdh_t::seq

The array of matrices that store the data we read in.

Definition at line 34 of file realIo.h.

Referenced by freeRdh(), generalMatchFactor(), getRdhDim(), getRdhSeqLength(), getRdhValue(), initRdh(), initRdhGslMat(), massSpecCompareWElut(), outputRealPats(), rmsdCompare(), setRdhColFromString(), setRdhLabel(), and setRdhValue().

D.8.0.310 int rdh_t::size

The number of sequences stored in this data structure.

Definition at line 27 of file realIo.h.

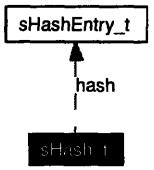
Referenced by initRdh(), initRdhIndex(), and main().

The documentation for this struct was generated from the following file:

- realIo.h

D.9 sHash_t Struct Reference

Collaboration diagram for sHash_t:



Data Fields

- int * hashSize
- int * iHashSize
- int totalSize
- sHashEntry_t ** hash

Detailed Description

A data structure for a hash table. At its root, this structure is just an array of hash entry objects. As well, there are members used to track the size of the hash table.

Definition at line 132 of file words.c.

Field Documentation

D.9.0.311 sHashEntry_t** sHash_t::hash

An array sHashEntry_t space objects.

Definition at line 148 of file words.c.

Referenced by destroySHash(), printSHash(), and searchSHash().

D.9.0.312 int* sHash_t::hashSize

A pointer to an integer that is used to store an array of integers that keep track of the number of sHashEntry_t objects that are hashed to a particular integer.

Definition at line 138 of file words.c.

Referenced by destroySHash(), and searchSHash().

D.9.0.313 int* sHash_t::iHashSize

A pointer to an integer that is used to store an array of integers that keep track of the number of sHashEntry_t objects that are hashed to a particular integer.

Definition at line 143 of file words.c.

Referenced by destroySHash(), and searchSHash().

D.9.0.314 int sHash_t::totalSize

An integer that stores the total number of slots available in our hash.

Definition at line 146 of file words.c.

Referenced by initSHash(), and searchSHash().

The documentation for this struct was generated from the following file:

- words.c

D.10 sHashEntry_t Struct Reference

Data Fields

- char * key
- int L
- int data
- int idx

Detailed Description

Type for a hash table entry. This datatype is used to populate a hash table. The most important members of this data structure are the string, or the key, and the index to which that key hashes.

Definition at line 114 of file words.c.

Field Documentation

D.10.0.315 int sHashEntry_t::data

A throw away variable, used to store any necessary data

Definition at line 121 of file words.c.

Referenced by countWords2(), and printSHash().

D.10.0.316 int sHashEntry_t::idx

The integer to which the *key* of length L hashes

Definition at line 123 of file words.c.

Referenced by countWords2().

D.10.0.317 char* sHashEntry_t::key

A pointer to a string

Definition at line 117 of file words.c.

Referenced by countWords2(), printSHash(), and searchSHash().

D.10.0.318 int sHashEntry_t::L

The length of the string that should be used to compute the hash

Definition at line 119 of file words.c.

Referenced by countWords2(), printSHash(), and searchSHash().

The documentation for this struct was generated from the following file:

- words.c

D.11 sOffset_t Struct Reference

`#include <spat.h>`

Data Fields

- int seq
- int pos
- int next
- int prev

Detailed Description

This object is used to store the location of a particular word and a set of sequences. That is if we hash a word, we would like to know where it came from. This data structure provides that information.

Definition at line 13 of file spat.h.

Field Documentation

D.11.0.319 int sOffset_t::next

The index of the word that follows this word at *pos* plus 1.

Definition at line 23 of file spat.h.

Referenced by countWords2().

D.11.0.320 int sOffset_t::pos

The position in the sequence where the word is located.

Definition at line 20 of file spat.h.

Referenced by countWords2(), and main().

D.11.0.321 int sOffset_t::prev

The index of the word that precedes this word at *pos* minus 1.

Definition at line 26 of file spat.h.

Referenced by countWords2().

D.11.0.322 int sOffset_t::seq

The sequence from which the word came.

Definition at line 17 of file spat.h.

Referenced by countWords2(), and main().

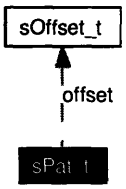
The documentation for this struct was generated from the following file:

- spat.h

D.12 sPat_t Struct Reference

`#include <spat.h>`

Collaboration diagram for sPat_t:



Data Fields

- char * string
- int length
- int support
- sOffset_t * offset

Detailed Description

This data structure is used to store the locations of all the instances of a particular word of length *length* in a set of sequences. This data structure is used principally by the string based version of Gemoda and is used to store words that are hashed before the comparison phase.

Definition at line 36 of file spat.h.

Field Documentation

D.12.0.323 int sPat_t::length

The length of this word.

Definition at line 43 of file spat.h.

Referenced by countWords2(), and printSPats().

D.12.0.324 sOffset_t* sPat_t::offset

An array of sOffset_t objects storing the loci, or offsets where this word occurs.

Definition at line 50 of file spat.h.

Referenced by countWords2(), destroySPatA(), and main().

D.12.0.325 char* sPat_t::string

The pointer to the string for this word.

Definition at line 40 of file spat.h.

Referenced by countWords2().

D.12.0.326 int sPat_t::support

The number of times this word occurs in the sequence set.

Definition at line 46 of file spat.h.

Referenced by countWords2().

The documentation for this struct was generated from the following file:

- spat.h

D.13 sSize_t Struct Reference

Data Fields

- int start
- int stop
- int size

Detailed Description

Definition at line 165 of file fastaSeqIO.c.

Field Documentation

D.13.0.327 int sSize_t::size

Definition at line 168 of file fastaSeqIO.c.

Referenced by ReadFSeqs().

D.13.0.328 int sSize_t::start

Definition at line 166 of file fastaSeqIO.c.

Referenced by ReadFSeqs().

D.13.0.329 int sSize_t::stop

Definition at line 167 of file fastaSeqIO.c.

Referenced by ReadFSeqs().

The documentation for this struct was generated from the following file:

- FastaSeqIO/fastaSeqIO.c

Bibliography

[1] S Aerts, G Thijs, B Coessens, M Staes, Y Moreau, and B De Moor. Toucan: deciphering the cis-regulatory logic of coregulated genes. *Nucleic Acids Res*, 31(6):1753–1764, Mar 2003. 45

[2] N N Alexandrov. SARFing the PDB. *Protein Eng*, 9(9):727–732, Oct 1996. 88

[3] N N Alexandrov and D Fischer. Analysis of topological and nontopological structural similarities in the PDB: new examples with old structures. *Proteins*, 25(3):354–365, Aug 1996. 88

[4] N N Alexandrov and N Go. Biological meaning, statistical significance, and classification of local spatial similarities in nonhomologous proteins. *Protein Sci*, 3(6):866–875, Jul 1994. 88

[5] H Alper, K Miyaoku, and G Stephanopoulos. Construction of lycopene-overproducing e. coli strains by combining systematic and combinatorial gene knockout targets. *Nat Biotechnol*, 23(5):612–616, May 2005. 120, 130, 132, 210

[6] S. F. Altschul. Amino acid substitution matrices from an information theoretic perspective. *J Mol Biol*, 219:555–65, 1991. 166, 186

[7] S F Altschul. National Center for Biotechnology Information. Personal communication, June 2005. 166

[8] S F Altschul. National Center for Biotechnology Information. Personal communication, May 2006. 177

[9] S F Altschul, R Bundschuh, R Olsen, and T Hwa. The estimation of statistical parameters for local alignment score distributions. *Nucleic Acids Res*, 29(2):351–361, Jan 2001. 165

[10] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215:403–10, 1990. 40, 163, 165, 179, 184

[11] S. F. Altschul, T. L. Madden, A. A. Schaffer Zhang, J., Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res*, 25:3389–402, 1997. 41, 47, 62, 91

[12] S Anderson. Shotgun DNA sequencing using cloned DNase I-generated fragments. *Nucleic Acids Res*, 9(13):3015–27, Jul 1981. 23

[13] M R Antoniewicz, J K Kelleher, and G Stephanopoulos. Elementary metabolite units (emu): a novel framework for modeling isotopic distributions. *Metab Eng*, 9(1):68–86, Jan 2007. 206

[14] L Aravind and E V Koonin. The HD domain defines a new superfamily of metal-dependent phosphohydrolases. *Trends Biochem Sci*, 23(12):469–472, 1998. 84

[15] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-d point sets. *IEEE Trans. Pattern Anal. Mach. Intell.*, 9(5):698–700, 1987. 89

[16] P Ausloos, C L Clifton, S G Lias, A I Mikaya, S E Stein, D V Tchekhovskoi, O D Sparkman, V Zaikin, and D Zhu. The critical evaluation of a comprehensive mass spectral library. *J Am Soc Mass Spectrom*, 10(4):287–299, Apr 1999. 115

[17] T L Bailey and C Elkan. Fitting a mixture model by expectation maximization to discover motifs in biopolymers. *Proc Int Conf Intell Syst Mol Biol*, 2:28–36, 1994. 38, 44, 62, 99

[18] A Bairoch. Prosite: a dictionary of sites and patterns in proteins. *Nucleic Acids Res*, 19 Suppl:2241–2245, Apr 1991. 183

[19] A Bairoch. The ENZYME database in 2000. *Nucleic Acids Res*, 28(1):304–305, Feb 2000. 84

[20] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000. *Nucleic Acids Res*, 28(1):45–48, Jan 2000. 85

[21] A Bairoch and B Boeckmann. The swiss-prot protein sequence data bank. *Nucleic Acids Res*, 20 Suppl:2019–2022, May 1992. 183

[22] A Barsch, T Patschkowski, and K Niehaus. Comprehensive metabolite profiling of sinorhizobium meliloti using gas chromatography-mass spectrometry. *Funct Integr Genomics*, 4(4):219–230, Oct 2004. 117

[23] Alex Bateman, Lachlan Coin, Richard Durbin, Robert D Finn, Volker Hollich, Sam Griffiths-Jones, Ajay Khanna, Mhairi Marshall, Simon Moxon, Erik L L Sonnhammer, David J Studholme, Corin Yeats, and Sean R Eddy. The Pfam protein families database. *Nucleic Acids Res*, 32 Database issue:138–141, Feb 2004. 84

[24] H Berman, K Henrick, and H Nakamura. Announcing the worldwide protein data bank. *Nat Struct Biol*, 10(12):980–980, Dec 2003. 23

[25] H M Berman, J Westbrook, Z Feng, G Gilliland, T N Bhat, H Weissig, I N Shindyalov, and P E Bourne. The Protein Data Bank. *Nucleic Acids Res*, 28(1):235–242, Feb 2000. 23, 89

[26] R J Bino, C H Ric de Vos, M Lieberman, R D Hall, A Bovy, H H Jonker, Y Tikunov, A Lommen, S Moco, and I Levin. The light-hyperresponsive high pigment-2dg mutation of tomato: alterations in the fruit metabolome. *New Phytol*, 166(2):427–438, May 2005. 117

[27] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. An up-

dated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, Jun 2002. 217

[28] M Blanchette and M Tompa. Discovery of regulatory elements by a computational method for phylogenetic footprinting. *Genome Res*, 12(5):739–748, May 2002. 47, 48

[29] N Bray, I Dubchak, and L Pachter. Avid: A global alignment program. *Genome Res*, 13(1):97–102, Jan 2003. 47

[30] S E Brenner, P Koehl, and M Levitt. The ASTRAL compendium for protein structure and sequence analysis. *Nucleic Acids Res*, 28(1):254–6, Jan 2000. 164, 183

[31] M Brosché, B Vinocur, E R Alatalo, A Lamminmäki, T Teichmann, E A Ottow, D Djilianov, D Afif, M B Bogeat-Triboulot, A Altman, A Polle, E Dreyer, S Rudd, L Paulin, P Auvinen, and J Kangasjärvi. Gene expression and metabolite profiling of populus euphratica growing in the negev desert. *Genome Biol*, 6(12), 2005. 117

[32] H Brunengraber. private communication, 2005. 147, 149

[33] J Buhler and M Tompa. Finding motifs using random projections. *Journal of Computational Biology*, 9(2):225–242, Apr 2002. 96, 102

[34] Jeremy Buhler and Martin Tompa. Finding motifs using random projections. In *Proceedings of the fifth annual international conference on Computational biology*, pages 69–76. ACM Press, 2001. 62, 95, 96

[35] C Burge and S Karlin. Prediction of complete gene structures in human genomic dna. *J Mol Biol*, 268(1):78–94, Apr 1997. 35

[36] J A Chemler, Y Yan, and M A Koffas. Biosynthesis of isoprenoids, polyunsaturated fatty acids and flavonoids in saccharomyces cerevisiae. *Microb Cell Fact*, 5:20–20, 2006. 54

[37] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. 32

[38] W H Day and F R McMorris. Critical comparison of consensus methods for molecular sequences. *Nucleic Acids Res*, 20(5):1093–1099, Mar 1992. 32

[39] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. In M. O. Dayhoff, editor, *Atlas of Protein Structure*, volume 5(Suppl. 3), pages 345–352. National Biomedical Reasearch Foundataion, Silver Spring, Md., 1978. 179

[40] G G Desbrosses, J Kopka, and M K Udvardi. Lotus japonicus metabolic profiling. development of gas chromatography-mass spectrometry resources for the study of plant-microbe interactions. *Plant Physiol*, 137(4):1302–1318, Apr 2005. 117

[41] R Devantier, B Scheithauer, S G Villas-Bôas, S Pedersen, and L Olsson. Metabolite profiling for analysis of yeast stress response during very high gravity ethanol fermentations. *Biotechnol Bioeng*, 90(6):703–714, Jun 2005. 117

[42] S Dietmann and L Holm. Identification of homology in protein structure classification. *Nat Struct Biol*, 8(11):953–957, Nov 2001. 88

[43] Jack Dongarra. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard I. *The International Journal of High Performance Computing Applications*, 16(1):1–111, Spring 2002. 217

[44] Jack Dongarra. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard II. *The International Journal of High Performance Computing Applications*, 16(2):115–199, Summer 2002. 217

[45] I Dubchak, M Brudno, G G Loots, L Pachter, C Mayor, E M Rubin, and K A Frazer. Active conservation of noncoding sequences revealed by three-way species comparisons. *Genome Res*, 10(9):1304–1306, Sep 2000. 47

[46] W B Dunn, N J Bailey, and H E Johnson. Measuring the metabolome: current analytical technologies. *Analyst*, 130(5):606–625, May 2005. 115, 145

[47] A L Duran, J Yang, L Wang, and L W Sumner. Metabolomics spectral formatting, alignment and conversion tools (msfacts). *Bioinformatics*, 19(17):2283–2293, Nov 2003. 116, 117

[48] S R Eddy. Profile hidden Markov models. *Bioinformatics*, 14(9):755–763, 1998. 76

[49] S R Eddy. Where did the BLOSUM62 alignment score matrix come from? *Nat Biotechnol*, 22(8):1035–1036, Aug 2004. 163

[50] I Eidhammer, I Jonassen, and W R Taylor. Structure comparison and structure patterns. *J Comput Biol*, 7(5):685–716, 2000. 88

[51] K Ellrott, C Yang, F M Sladek, and T Jiang. Identifying transcription factor binding sites through markov chain optimization. *Bioinformatics*, 18 Suppl 2:100–109, 2002. 35

[52] Eleazar Eskin and Pavel A Pevzner. Finding composite regulatory patterns in DNA sequences. *Bioinformatics*, 18 Suppl 1:354–363, 2002. Evaluation Studies. 62, 198, 201

[53] O Fiehn. Metabolomics–the link between genotypes and phenotypes. *Plant Mol Biol*, 48(1-2):155–171, Jan 2002. 112

[54] O Fiehn. Metabolic networks of cucurbita maxima phloem. *Phytochemistry*, 62(6):875–886, Mar 2003. 115

[55] O Fiehn, J Kopka, P Dörmann, T Altmann, R N Trethewey, and L Willmitzer. Metabolite profiling for plant functional genomics. *Nat Biotechnol*, 18(11):1157–1161, Nov 2000. 58, 117

[56] O Fiehn, J Kopka, R N Trethewey, and L Willmitzer. Identification of uncommon plant metabolites based on calculation of elemental compositions using gas

chromatography and quadrupole mass spectrometry. *Anal Chem*, 72(15):3573–3580, Aug 2000. 115, 138

[57] J Forster, I Famili, P Fu, B O Palsson, and J Nielsen. Genome-scale reconstruction of the saccharomyces cerevisiae metabolic network. *Genome Res*, 13(2):244–253, Feb 2003. 112

[58] M Galassi, J Davies, J Theiler, B Gough, G Jungman, M Booth, and F Rossi. *GNU Scientific Library Reference Manual (2nd Ed.)*. 2003. 217

[59] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP–Completeness*. W.H. Freeman and Company, New York, 1979. 62, 69

[60] M. Gribskov and N. L. Robinson. Use of receiver operating characteristic (ROC) analysis to evaluate sequence matching. *Computers in Chemistry*, 20(1):25–33, 1996. 165, 184

[61] D. Guhathakurta, L. A. Schriefer, M. C. Hresko, R. H. Waterson, and G. D. Stormo. Indentifying muscle regulartory elements and genes in the nematode *caenorhabditis elegans*. Department of Genetics, Washington University School of Medicine. 45

[62] M Gupta and J S Liu. Discovery of conserved sequence patterns using a stochastic dictionary model. *J Am Statist Assoc*, 98:55–66, 2003. 44

[63] P Hegde, R Qi, R Gaspard, K Abernathy, S Dharap, J Earle-Hughes, C Gay, N U Nwokekeh, T Chen, A I Saeed, V Sharov, N H Lee, T J Yeatman, and J Quackenbush. Identification of tumor markers in models of human colorectal cancer using a 19,200-element complementary DNA microarray. *Cancer Res*, 61(21):7792–7797, Nov 2001. 62

[64] M K Hellerstein. New stable isotope-mass spectrometric techniques for measuring fluxes through intact metabolic pathways in mammalian systems: introduction of moving pictures into functional genomics and biochemical phenotyping. *Metab Eng*, 6(1):85–100, Jan 2004. 112

[65] J G Henikoff. Fred Hutchinson Cancer Research Center. Personal communication, October 2005. 195

[66] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc Natl Acad Sci U S A*, 89(22):10915–10919, Nov 1992. 83, 163, 164, 179, 180, 183, 186, 188, 189, 191, 195

[67] S. Henikoff and J. G. Henikoff. Performance evaluation of amino acid substitution matrices. *Proteins*, 17:49–61, 1993. 163, 166, 179, 183, 185, 189, 191

[68] S Henikoff and J G Henikoff. Protein family classification based on searching a database of blocks. *Genomics*, 19(1):97–107, Jan 1994. 179

[69] S. Henikoff and J. G. Henikoff. *Amino Acid Substitution Matrices*, volume 54 of *Advances in Protein Chemistry*, pages 73–98. Academic Press, San Diego, 2000. 84, 163, 179

[70] S Henikoff, J G Henikoff, W J Alford, and S Pietrokovski. Automated construction and graphical presentation of protein blocks from unaligned sequences. *Gene*, 163(2):GC17–26, Oct 1995. 62, 163, 169

[71] D Herebian, B Hanisch, and F J Marner. Strategies for gathering structural information on unknown peaks in the gc/ms analysis of corynebacterium glutamicum cell extracts. *Metabolomics*, 1(4):317–324, 2005. 117

[72] G Z Hertz and G D Stormo. Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics*, 15(7-8):563–577, Jul 1999. 62, 99

[73] M Y Hirai, M Yano, D B Goodenowe, S Kanaya, T Kimura, M Awazuhara, M Arita, T Fujiwara, and K Saito. Integration of transcriptomics and metabolomics for understanding of global responses to nutritional stresses in arabidopsis thaliana. *Proc Natl Acad Sci U S A*, 101(27):10205–10210, Jul 2004. 112, 116

[74] K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch. The PROSITE database, its status in 1999. *Nucleic Acids Res*, 27:215–9, 1999. 76, 163, 179

[75] L Holm, C Ouzounis, C Sander, G Tuparev, and G Vriend. A database of protein structure families with common folding motifs. *Protein Sci*, 1(12):1691–1698, 1992. 62

[76] L Holm and C Sander. Protein structure comparison by alignment of distance matrices. *J Mol Biol*, 233(1):123–138, Oct 1993. 88, 91

[77] L Holm and C Sander. Enzyme HIT. *Trends Biochem Sci*, 22(4):116–117, May 1997. Letter. 89, 90

[78] B K P Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America A*, 4(4):629–642, Apr 1987. 89

[79] J. D. Hughes, P. W. Estep, S. Tavazoie, and G. M. Church. Computational identification of cis-regulatory elements associated with groups of functionally related genes in Saccharomyces cerevisiae. *J Mol Biol*, 296:1205–14, 2000. 45

[80] Cornelius G Hunter and Shankar Subramaniam. Protein fragment clustering and canonical local shapes. *Proteins*, 50(4):580–588, Apr 2003. Evaluation Studies. 88

[81] H Idborg, P O Edlund, and S P Jacobsson. Multivariate approaches for efficient detection of potential metabolites from liquid chromatography/mass spectrometry data. *Rapid Commun Mass Spectrom*, 18(9):944–954, 2004. 119

[82] H Idborg-Björkman, P O Edlund, O M Kvalheim, I Schuppe-Koistinen, and S P Jacobsson. Screening of biomarkers in rat urine using lc/electrospray ionization-ms and two-way data analysis. *Anal Chem*, 75(18):4784–4792, Sep 2003. 119

[83] F Jacob and J Monod. Genetic regulatory mechanisms in the synthesis of proteins. *Journal of Molecular Biology*, 3:318–356, 1961. 54

[84] K L Jensen, M P Styczynski, I Rigoutsos, and G N Stephanopoulos. A generic motif discovery algorithm for sequential data. *Bioinformatics*, 22(1):21–28, Jan 2006. 120, 123

[85] I Jonassen, J F Collins, and D G Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Sci*, 4(8):1587–1595, Aug 1995. 49, 62

[86] Inge Jonassen, Ingvar Eidhammer, Darrell Conklin, and William R Taylor. Structure motif discovery and mining the PDB. *Bioinformatics*, 18(2):362–367, Feb 2002. 88

[87] P Jonsson, J Gullberg, A Nordström, M Kusano, M Kowalczyk, M Sjöström, and T Moritz. A strategy for identifying differences in large series of metabolomic samples analyzed by gc/ms. *Anal Chem*, 76(6):1738–1745, Mar 2004. 116

[88] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.* Prentice Hall, Upper Saddle River, New Jersey, 2000. 32

[89] H H Kanani and M I Klapa. Data correction strategy for metabolomics analysis using gas chromatography-mass spectrometry. *Metab Eng*, 9(1):39–51, Jan 2007. 113

[90] M Kanehisa and S Goto. Kegg: kyoto encyclopedia of genes and genomes. *Nucleic Acids Res*, 28(1):27–30, Jan 2000. 112

[91] F Kaplan, J Kopka, D W Haskell, W Zhao, K C Schiller, N Gatzke, D Y Sung, and C L Guy. Exploring the temperature-stress metabolome of arabidopsis. *Plant Physiol*, 136(4):4159–4168, Dec 2004. 117

[92] M Katajamaa and M Oresic. Processing methods for differential analysis of lc/ms profile data. *BMC Bioinformatics*, 6:179–179, 2005. 115, 119

[93] J E Katz, D S Dumlao, S Clarke, and J Hau. A new technique (comspari) to facilitate the identification of minor compounds in complex mixtures by gc/ms and lc/ms: tools for the visualization of matched datasets. *J Am Soc Mass Spectrom*, 15(4):580–584, Apr 2004. 119

[94] U Keich and P A Pevzner. Finding motifs in the twilight zone. *Bioinformatics*, 18(10):1374–1381, Oct 2002. Evaluation Studies. 50, 62

[95] J K Kelleher. Estimating gluconeogenesis with [u-13c]glucose: molecular condensation requires a molecular approach. *Am J Physiol*, 277(3 Pt 1):395–400, Sep 1999. 206

[96] T Kind and O Fiehn. Metabolomic database annotations via query of elemental compositions: mass accuracy is insufficient even at less than 1 ppm. *BMC Bioinformatics*, 7:234–234, 2006. 138

[97] Rachel Kolodny, Patrice Koehl, and Michael Levitt. Comprehensive evaluation of protein structure alignment methods: scoring by geometric measures. *J Mol Biol*, 346(4):1173–88, Mar 2005. 88

[98] N Krasnogor and D A Pelta. Measuring the similarity of protein structures by means of the universal similarity metric. *Bioinformatics*, 20(7):1015–1021, Jun 2004. Evaluation Studies. 89

[99] A Krogh, B Larsson, G von Heijne, and E L Sonnhammer. Predicting transmembrane protein topology with a hidden markov model: application to complete genomes. *J Mol Biol*, 305(3):567–580, Jan 2001. 35

[100] C E Lawrence, S F Altschul, M S Boguski, J S Liu, A F Neuwald, and J C Wootton. Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment. *Science*, 262(5131):208–214, Oct 1993. 45, 46, 62, 99

[101] C D Lima, K L D'Amico, I Naday, G Rosenbaum, E M Westbrook, and W A Hendrickson. MAD analysis of FHIT, a putative human tumor suppressor from the HIT protein family. *Structure*, 5(6):763–774, Jul 1997. 89

[102] J S Liu. The collapsed gibbs sampler with applications to a gene regulation problem. *J Amer Statist Assoc*, 89:958–966, 1994. 46

[103] J S Liu, A F Neuwald, and C E Lawrence. Bayesian models for multiple local sequence alignment and its gibbs sampling strategies. *J Amer Statist Assoc*, 90:1156–1170, 1995. 45, 46

[104] X Liu, D L Brutlag, and J S Liu. Bioprospector: discovering conserved dna motifs in upstream regulatory regions of co-expressed genes. *Pac Symp Biocomput*, pages 127–138, 2001. 45

[105] X S Liu, D L Brutlag, and J S Liu. An algorithm for finding protein-dna binding sites with applications to chromatin-immunoprecipitation microarray experiments. *Nat Biotechnol*, 20(8):835–839, Aug 2002. 44

[106] G G Loots, I Ovcharenko, L Pachter, I Dubchak, and E M Rubin. rvista for comparative sequence-based discovery of functional transcription factor binding sites. *Genome Res*, 12(5):832–839, May 2002. 47, 48

[107] T Madej, J F Gibrat, and S H Bryant. Threading a database of protein cores. *Proteins*, 23(3):356–369, Nov 1995. 88

[108] A Mancheron and I Rusu. Pattern discovery allowing wild-cards, substitution matrices, and multiple score functions. In *Algorithms in Bioinformatics, Proceedings Lecture notes in Bioinformatics*, pages 124–138. Springer–Verlag, 2003. 63

[109] Aron Marchler-Bauer, John B Anderson, Carol DeWeese-Scott, Natalie D Fedorova, Lewis Y Geer, Siqian He, David I Hurwitz, John D Jackson, Aviva R Jacobs, Christopher J Lanczycki, Cynthia A Liebert, Chunlei Liu, Thomas Madej, Gabriele H Marchler, Raja Mazumder, Anastasia N Nikolskaya, Anna R Panchenko, Bachoti S Rao, Benjamin A Shoemaker, Vahan Simonyan, James S Song, Paul A Thiessen, Sona Vasudevan, Yanli Wang, Roxanne A Yamashita,

Jodie J Yin, and Stephen H Bryant. CDD: a curated Entrez database of conserved domain alignments. *Nucleic Acids Res*, 31(1):383–387, Feb 2003. 84

[110] M Margulies, M Egholm, W E Altman, S Attiya, J S Bader, L A Bemben, J Berka, M S Braverman, Y J Chen, Z Chen, S B Dewell, L Du, J M Fierro, X V Gomes, B C Godwin, W He, S Helgesen, C H Ho, C H Ho, G P Irzyk, S C Jando, M L Alenquer, T P Jarvie, K B Jirage, J B Kim, J R Knight, J R Lanza, J H Leamon, S M Lefkowitz, M Lei, J Li, K L Lohman, H Lu, V B Makhijani, K E McDade, M P McKenna, E W Myers, E Nickerson, J R Nobile, R Plant, B P Puc, M T Ronan, G T Roth, G J Sarkis, J F Simons, J W Simpson, M Srinivasan, K R Tartaro, A Tomasz, K A Vogt, G A Volkmer, S H Wang, Y Wang, M P Weiner, P Yu, R F Begley, and J M Rothberg. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, Sep 2005. 23

[111] C Mayor, M Brudno, J R Schwartz, A Poliakov, E M Rubin, K A Frazer, L S Pachter, and I Dubchak. Vista : visualizing global dna sequence alignments of arbitrary length. *Bioinformatics*, 16(11):1046–1047, Nov 2000. 47

[112] J Messing, R Crea, and P H Seeburg. A system for shotgun dna sequencing. *Nucleic Acids Res*, 9(2):309–321, Jan 1981. 23

[113] H W Mewes, K Albermann, M Bähr, D Frishman, A Gleissner, J Hani, K Heumann, K Kleine, A Maierl, S G Oliver, F Pfeiffer, and A Zollner. Overview of the yeast genome. *Nature*, 387(6632 Suppl):7–65, May 1997. 112

[114] Y. Moreau, F. De Smet, G. Thijs, K. Marchal, and B. De Moor. Functional bioinformatics of microarray data: From expression to regulation. *Proceedings of the IEEE*, 90(11):1722–1743, 2002. 45

[115] C R Morris, J T Scott, H-M Chang, R R Sederoff, D O'Malley, and J F Kadla. Metabolic profiling: A new tool in the study of wood formation. *Journal of Agricultural and Food Chemistry*, 52(6):1427–1434, Mar 2004. 117

[116] Venkatesh L Murthy and George D Rose. RNABase: an annotated database of RNA structures. *Nucleic Acids Res*, 31(1):502–504, Jan 2003. 62

[117] A G Murzin, S E Brenner, T Hubbard, and C Chothia. SCOP: a structural classification of proteins database for the investigation of sequences and structures. *J Mol Biol*, 247(4):536–40, Apr 1995. 164, 183

[118] A Nanchen, T Fuhrer, and U Sauer. Determination of metabolic flux ratios from 13c-experiments and gas chromatography-mass spectrometry data: protocol and principles. *Methods Mol Biol*, 358:177–197, 2007. 206

[119] S B Needleman and C D Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3):443–453, Mar 1970. 39

[120] P C Ng and S Henikoff. Predicting deleterious amino acid substitutions. *Genome Res*, 11(5):863–874, May 2001. 179

[121] J K Nicholson, J Connelly, J C Lindon, and E Holmes. Metabonomics: a platform for studying drug toxicity and gene function. *Nat Rev Drug Discov*, 1(2):153–161, Feb 2002. 119

[122] K F Nielsen and J Smedsgaard. Fungal metabolite screening: database of 474 mycotoxins and fungal metabolites for dereplication by standardised liquid chromatography-uv-mass spectrometry methodology. *J Chromatogr A*, 1002(1-2):111–136, Jun 2003. 115

[123] N P Nielsen, J Smedsgaard, and J C Frisvad. Full second-order chromatographic/spectrometric data matrices for automated sample identification and component analysis by non-data-reducing image analysis. *Anal Chem*, 71(3):727–735, Feb 1999. 119

[124] C Notredame, D G Higgins, and J Heringa. T-coffee: A novel method for fast and accurate multiple sequence alignment. *J Mol Biol*, 302(1):205–217, Sep 2000. 38, 163, 179

[125] S O'Hagan, W B Dunn, M Brown, J D Knowles, and D B Kell. Closed-loop, multiobjective optimization of analytical instrumentation: gas chromatography/time-of-flight mass spectrometry of the metabolomes of human serum and of yeast fermentations. *Anal Chem*, 77(1):290–303, Jan 2005. 58, 113, 138

[126] R Olsen, R Bundschuh, and T Hwa. Rapid assessment of extremal statistics for gapped local alignment. *Proc Int Conf Intell Syst Mol Biol*, pages 211–222, 1999. 165

[127] C A Orengo and W R Taylor. SSAP: sequential structure alignment program for protein structure comparison. *Methods Enzymol*, 266:617–635, 1996. 88

[128] Angel R Ortiz, Charlie E M Strauss, and Osvaldo Olmea. MAMMOTH (matching molecular models obtained from theory): an automated method for model comparison. *Protein Sci*, 11(11):2606–2621, Nov 2002. Evaluation Studies. 88

[129] W R Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods Enzymol*, 183:63–98, 1990. 163, 165, 184

[130] W R Pearson. Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. *Genomics*, 11(3):635–50, Nov 1991. 163, 165, 184, 186

[131] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85:2444–2448, 1988. 62, 163, 179

[132] P. A. Pevzner and S.H. Sze. Combinatorial approaches to finding subtle signals in DNA sequences. In *Proceedings International Conference on Intelligent Systems for Molecular Biology*, pages 269–278. AAAI Press, 2000. 63, 93, 94, 95, 96, 99, 101, 106, 108

[133] S Pietrokovski. Searching databases of conserved sequence regions by aligning protein multiple-alignments. *Nucleic Acids Res*, 24(19):3836–3845, Oct 1996. 179

[134] D Pribnow. Nucleotide sequence of an rna polymerase binding site at an early t7 promoter. *Proc Natl Acad Sci U S A*, 72(3):784–788, Mar 1975. 31, 33

[135] Alkes Price, Sriram Ramabhadran, and Pavel A Pevzner. Finding subtle motifs by branching from sample strings. *Bioinformatics*, 19 Suppl 2:II149–II155, Oct 2003. 62

[136] G A Price, G E Crooks, R E Green, and S E Brenner. Statistical evaluation of pairwise protein sequence comparison with the bayesian bootstrap. *Bioinformatics*, 21(20):3824–3831, Oct 2005. 164, 166, 183, 186, 190

[137] B Prithiviraj, A Vikram, A C Kushalappa, and V Yaylayan. Volatile metabolite profiling for the discrimination of onion bulbs infected by erwinia carotovora ssp. carotovora, fusariumoxysporum and botrytis allii. *European Journal of Plant Pathology*, 110(4):371–377, Apr 2004. 117

[138] L M Raamsdonk, B Teusink, D Broadhurst, N Zhang, A Hayes, M C Walsh, J A Berden, K M Brindle, D B Kell, J J Rowland, H V Westerhoff, K van Dam, and S G Oliver. A functional genomics strategy that uses metabolome data to reveal the phenotype of silent mutations. *Nat Biotechnol*, 19(1):45–50, Jan 2001. 58, 112, 116

[139] Sridhar Ramaswamy and Todd R Golub. DNA microarrays in clinical oncology. *J Clin Oncol*, 20(7):1932–1941, Apr 2002. 62

[140] I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in biological sequences: The TEIRESIAS algorithm. *Bioinformatics*, 14:55–67, 1998. 38, 42, 62, 63, 68

[141] I. Rigoutsos, A. Floratos, C. Ouzounis, Y. Gao, and L. Parida. Dictionary building via unsupervised hierarchical motif discovery in the sequence space of natural proteins. *Proteins*, 37:264–77, 1999. 63

[142] U Roessner, L Willmitzer, and A R Fernie. High-resolution metabolic phenotyping of genetically and environmentally diverse potato tuber systems. identification of phenocopies. *Plant Physiol*, 127(3):749–764, Nov 2001. 117

[143] U Roessner-Tunali, B Hegemann, A Lytovchenko, F Carrari, C Bruedigam, D Granot, and A R Fernie. Metabolic profiling of transgenic tomato plants overexpressing hexokinase reveals that the influence of hexose phosphorylation diminishes during fruit development. *Plant Physiol*, 133(1):84–99, Sep 2003. 117

[144] T M Rose, E R Schultz, J G Henikoff, S Pietrokovski, C M McCallum, and S Henikoff. Consensus-degenerate hybrid oligonucleotide primers for amplification of distantly related sequences. *Nucleic Acids Res*, 26(7):1628–1635, Apr 1998. 179

[145] Heladia Salgado, Socorro Gama-Castro, Agustino Martinez-Antonio, Edgar Diaz-Peredo, Fabiola Sanchez-Solano, Martin Peralta-Gil, Delfino Garcia-Alonso, Veronica Jimenez-Jacinto, Alberto Santos-Zavaleta, Cesar Bonavides-Martinez, and Julio Collado-Vides. RegulonDB (version 4.0): transcriptional regulation, operon organization and growth conditions in Escherichia coli K-12. *Nucleic Acids Res*, 32(Database issue):303–306, Jan 2004. 86

[146] A A Schäffer, L Aravind, T L Madden, S Shavirin, J L Spouge, Y I Wolf, E V Koonin, and S F Altschul. Improving the accuracy of psi-blast protein database searches with composition-based statistics and other refinements. *Nucleic Acids Res*, 29(14):2994–3005, Jul 2001. 166, 185

[147] N Schauer, D Steinhauser, S Strelkov, D Schomburg, G Allison, T Moritz, K Lundgren, U Roessner-Tunali, M G Forbes, L Willmitzer, A R Fernie, and

J Kopka. Gc-ms libraries for the rapid identification of metabolites in complex biological samples. *FEBS Lett*, 579(6):1332–1337, Feb 2005. 138, 210

[148] S Schwartz, Z Zhang, K A Frazer, A Smit, C Riemer, J Bouck, R Gibbs, R Hardison, and W Miller. Pipmaker–a web server for aligning two genomic dna sequences. *Genome Res*, 10(4):577–586, Apr 2000. 47

[149] Jessica Shapiro and Douglas Brutlag. FoldMiner and LOCK 2: protein structure comparison and motif discovery on the web. *Nucleic Acids Res*, 32(Web Server issue):W536–41, Jul 2004. 88

[150] A K Smilde, J J Jansen, H C Hoefsloot, R J Lamers, J van der Greef, and M E Timmerman. Anova-simultaneous component analysis (asca): a new tool for analyzing designed metabolomics data. *Bioinformatics*, 21(13):3043–3048, Jul 2005. 119

[151] M G Smith, T A Gianoulis, S Pukatzki, J J Mekalanos, L N Ornston, M Gerstein, and M Snyder. New insights into acinetobacter baumannii pathogenesis revealed by high-density pyrosequencing and transposon mutagenesis. *Genes Dev*, 21(5):601–614, Mar 2007. 23

[152] T F Smith and M S Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–7, Mar 1981. 38, 163, 165, 184

[153] R Staden. Computer methods to locate signals in nucleic acid sequences. *Nucleic Acids Res*, 12(1 Pt 2):505–519, Jan 1984. 35

[154] S E Stein. An integrated method for spectrum extraction and compound identification from gas chromatography/mass spectrometry data. *J Am Soc Mass Spectrom*, 10(8):770–781, Aug 1999. 113, 121, 123

[155] R Z Stolzenberg-Solomon, B I Graubard, S Chari, P Limburg, P R Taylor, J Virtamo, and D Albanes. Insulin, glucose, insulin resistance, and pancreatic cancer in male smokers. *JAMA*, 294(22):2872–2878, Dec 2005. 55

[156] G D Stormo. DNA binding sites: representation and discovery. *Bioinformatics*, 16(1):16–23, Jan 2000. Historical Article. 32

[157] G D Stormo, T D Schneider, L Gold, and A Ehrenfeucht. Use of the 'Perceptron' algorithm to distinguish translational initiation sites in E. coli. *Nucleic Acids Res*, 10(9):2997–3011, May 1982. 35

[158] S Strelkov, M von Elstermann, and D Schomburg. Comprehensive analysis of metabolites in corynebacterium glutamicum by gas chromatography/mass spectrometry. *Biol Chem*, 385(9):853–861, Sep 2004. 117

[159] M P Styczynski, K L Jensen, I R Rigoutsos, and G N Stephanopoulos. The evolution of updated BLOSUM matrices and the Blocks database. *Bioinformatics*, In preparation, 2006. 163

[160] L W Sumner, P Mendes, and R A Dixon. Plant metabolomics: large-scale phytochemistry in the functional genomics era. *Phytochemistry*, 62(6):817–836, Mar 2003. 112

[161] L Tarpley, A L Duran, T H Kebrom, and L W Sumner. Biomarker metabolites capturing the metabolite variance present in a rice plant developmental period. *BMC Plant Biol*, 5:8–8, 2005. 117

[162] J Taylor, R D King, T Altmann, and O Fiehn. Application of metabolomics to plant genotype discrimination using statistics and machine learning. *Bioinformatics*, 18 Suppl 2:241–248, 2002. 117

[163] J D Thompson, D G Higgins, and T J Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res*, 22(22):4673–4680, Nov 1994. 38, 163, 179

[164] Y Tikunov, A Lommen, C H de Vos, H A Verhoeven, R J Bino, R D Hall, and A G Bovy. A novel approach for nontargeted data analysis for metabolomics.

large-scale profiling of tomato fruit volatiles. *Plant Physiol*, 139(3):1125–1137, Nov 2005. 117

[165] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahasi. An optimal algorithm for finding all the cliques. *SIG Algorithms*, 12:91–98, 1989. 69

[166] Martin Tompa, Nan Li, Timothy L Bailey, George M Church, Bart De Moor, Eleazar Eskin, Alexander V Favorov, Martin C Frith, Yutao Fu, W James Kent, Vsevolod J Makeev, Andrei A Mironov, William Stafford Noble, Giulio Pavesi, Graziano Pesole, Mireille Regnier, Nicolas Simonis, Saurabh Sinha, Gert Thijs, Jacques van Helden, Mathias Vandenbogaert, Zhiping Weng, Christopher Workman, Chun Ye, and Zhou Zhu. Assessing computational tools for the discovery of transcription factor binding sites. *Nat Biotechnol*, 23(1):137–144, Jan 2005. 63

[167] A Traven, B Jelicic, and M Sopta. Yeast gal4: a transcriptional paradigm revisited. *EMBO Rep*, 7(5):496–499, May 2006. 54

[168] J C Venter, H O Smith, and L Hood. A new strategy for genome sequencing. *Nature*, 381(6581):364–366, May 1996. 23

[169] J C Verdonk, C H Ric de Vos, H A Verhoeven, M A Haring, A J van Tunen, and R C Schuurink. Regulation of floral scent production in petunia revealed by targeted metabolomics. *Phytochemistry*, 62(6):997–1008, Mar 2003. 117

[170] A Vikram, B Prithiviraj, and A C Kushalappa. Use of volatile metabolite profiles to discriminate fungal diseases of cortland and empire apples. *Journal of Plant Pathology*, 86:215–225, 2004. 117

[171] S G Villas-Bôas, D G Delicado, M Akesson, and J Nielsen. Simultaneous analysis of amino and nonamino organic acids as methyl chloroformate derivatives using gas chromatography-mass spectrometry. *Anal Biochem*, 322(1):134–138, Nov 2003. 120, 209

[172] S G Villas-Bôas, S Mas, M Akesson, J Smedsgaard, and J Nielsen. Mass spectrometry in metabolome analysis. *Mass Spectrom Rev*, 24(5):613–646, Sep-Oct 2005. 116

[173] S G Villas-Bôas, J F Moxley, M Akesson, G Stephanopoulos, and J Nielsen. High-throughput metabolic state analysis: the missing link in integrated functional genomics of yeasts. *Biochem J*, 388(Pt 2):669–677, Jun 2005. 117, 138, 211

[174] G. Vogt, T. Etzold, and P. Argos. An assessment of amino acid exchange matrices in aligning protein sequences: the twilight zone revisited. *J Mol Biol*, 249:816–31, 1995. 163, 179

[175] W W Wasserman, M Palumbo, W Thompson, J W Fickett, and C E Lawrence. Human-mouse genome comparisons to locate regulatory sites. *Nat Genet*, 26(2):225–228, Oct 2000. 45, 47

[176] W Weckwerth, M E Loureiro, K Wenzel, and O Fiehn. Differential metabolic networks unravel the effects of silent plant phenotypes. *Proc Natl Acad Sci U S A*, 101(20):7809–7814, May 2004. 112, 116

[177] J E Wedekind, P A Frey, and I Rayment. The structure of nucleotidylated histidine-166 of galactose-1-phosphate uridylyltransferase provides insight into phosphoryl group transfer. *Biochemistry*, 35(36):11560–11569, Oct 1996. 89

[178] E. Wingender, X. Chen, R. Hehl, H. Karas, I. Liebich, V. Matys, T. Meinhardt, M. Prüss, I. Reuter, and F. Schacherer. TRANSFAC: an integrated system for gene expression regulation. *Nucleic Acids Res*, 28(1):316–319, Jan 2000. 35

[179] D S Wishart, D Tzur, C Knox, R Eisner, A C Guo, N Young, D Cheng, K Jewell, D Arndt, S Sawhney, C Fung, L Nikolai, M Lewis, M A Coutouly, I Forsythe, P Tang, S Shrivastava, K Jeroncic, P Stothard, G Amegbey, D Block, D D Hau, J Wagner, J Miniaci, M Clements, M Gebremedhin, N Guo, Y Zhang, G E Duggan, G D Macinnis, A M Weljie, R Dowlatabadi, F Bamforth, D Clive,

R Greiner, L Li, T Marrie, B D Sykes, H J Vogel, and L Querengesser. Hmdb: the human metabolome database. *Nucleic Acids Res*, 35(Database issue):521–526, Jan 2007. 142

[180] J W Wong, G Cagney, and H M Cartwright. Specalign–processing and alignment of mass spectra datasets. *Bioinformatics*, 21(9):2088–2090, May 2005. 115

[181] J. C. Wootton and S. Federhen. Statistics of local complexity in amino acid sequences and sequence databases. *Computers in Chemistry*, 17:149–163, 1993. 165, 184

[182] Mohammed J. Zaki and Mitsunori Ogihara. Theoretical foundations of association rules. In *In Proceedings of 3 rd SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD'98)*, Seattle, Washington, 1998. 63

[183] Mohammed Javeed Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000. 63