Motif discovery in sequential data

by

Kyle L. Jensen

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2006

Motif discovery in sequential data

by

Kyle L. Jensen

Submitted to the Department of Chemical Engineering on April 27, 2006, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

In this thesis, I discuss the application and development of methods for the automated discovery of motifs in sequential data. These data include DNA sequences, protein sequences, and real-valued sequential data such as protein structures and timeseries of arbitrary dimension. As more genomes are sequenced and annotated, the need for automated, computational methods for analyzing biological data is increasing rapidly. In broad terms, the goal of this thesis is to treat sequential data sets as unknown languages and to develop tools for interpreting an understanding these languages.

The first chapter of this thesis is an introduction to the fundamentals of motif discovery, which establishes a common mode of thought and vocabulary for the subsequent chapters. One of the central themes of this work is the use of grammatical models, which are more commonly associated with the field of computational linguistics. In the second chapter, I use grammatical models to design novel antimicrobial peptides (AmPs). AmPs are small proteins used by the innate immune system to combat bacterial infection in multicellular eukaryotes. There is mounting evidence that these peptides are less susceptible to bacterial resistance than traditional antibiotics and may form the basis for a novel class of therapeutics. In this thesis, I described the rational design of novel AmPs that show limited homology to naturally-occurring proteins but have strong bacteriostatic activity against several species of bacteria, including *Staphylococcus aureus* and *Bacillus anthracis*. These peptides were designed using a linguistic model of natural AmPs by treating the amino acid sequences of natural AmPs as a formal language and building a set of regular grammars to describe this language. This set of grammars was used to create novel, unnatural AmP sequences that conform to the formal syntax of natural antimicrobial peptides but populate a previously unexplored region of protein sequence space.

The third chapter describes a novel, GEneric MOtif DIscovery Algorithm (Gemoda) for sequential data. Gemoda can be applied to any dataset with a sequential character, including both categorical and real-valued data. As I show, Gemoda deterministically discovers motifs that are maximal in composition and length. As well, the algorithm allows any choice of similarity metric for finding motifs. These motifs are representation-agnostic: they can be represented using regular expressions, position weight matrices, or any other model for sequential data. I demonstrate a number of applications of the algorithm, including the discovery of motifs in amino acids and DNA sequences, and the discovery of conserved protein sub-structures.

The final chapter is devoted to a series of smaller projects, employing tools and methods
indirectly related to motif discovery in sequential data. I describe the construction of a software tool, Biogrep that is designed to match large pattern sets against large biosequence databases in a parallel fashion. This makes biogrep well-suited to annotating sets of sequences using biologically significant patterns. In addition, I show that the BLOSUM series of amino acid substitution matrices, which are commonly used in motif discovery and sequence alignment problems, have changed drastically over time. The fidelity of amino acid sequence alignment and motif discovery tools depends strongly on the target frequencies implied by these underlying matrices. Thus, these results suggest that further optimization of these matrices is possible.

The final chapter also contains two projects wherein I apply statistical motif discovery tools instead of grammatical tools. In the first of these two, I develop three different physiochemical representations for a set of roughly 700 HIV–I protease substrates and use these representations for sequence classification and annotation. In the second of these two projects, I develop a simple statistical method for parsing out the phenotypic contribution of a single mutation from libraries of functional diversity that contain a multitude of mutations and varied phenotypes. I show that this new method successfully elucidates the effects of single nucleotide polymorphisms on the strength of a promoter placed upstream of a reporter gene.

The central theme, present throughout this work, is the development and application of novel approaches to finding motifs in sequential data. The work on the design of AmPs is very applied and relies heavily on existing literature. In contrast, the work on Gemoda is the greatest contribution of this thesis and contains many new ideas.

Thesis Supervisor: Gregory N. Stephanopoulos
Title: Bayer Professor of Chemical Engineering
Acknowledgments

I am indebted to many people who both directly and indirectly contributed to this thesis. First, I would like to thank those collaborators who directly contributed. Most of all, I’m grateful for the help and friendship of Mark Styczynski. Mark was my collaborator on all matters computational for the past four years — his influence is evident throughout this document. I’m also grateful for my collaboration with Christopher Loose, who performed many of the experiments on antimicrobial peptides described in Chapter 2. Finally, I would like to thank Isidore Rigoutsos, who straddled the line between collaborator and advisor. Isidore taught me an attention to detail and a penchant for the UNIX command line and vi editor.

Most importantly, I am indebted to Greg Stephanopoulos, my advisor, whose guidance and support was unwavering these past six years. Greg is the perpetual optimist — always positive in the face of my many failures along the way. He also gave me the freedom to pursue projects of my own choosing, which contributed greatly to my academic independence, if not the selection of wise projects.

I am much obliged to my thesis committee members: my advisor Greg, Isidore, Bill Green, and Bob Berwick. My committee was always flexible in scheduling and judicious in their application of both carrots and sticks.

There are numerous people who contributed indirectly to this thesis. First among these is my intelligent, lovely, and vivacious wife Kathryn Miller–Jensen. Next, my parents Carl and Julie, my sister Heather, and my in-laws, Ron, Joyce, Suzanne, Jeff, and Mindi. Finally, there are innumerable friends who contributed and to whom I am greatly appreciative including Michael Raab, Joel Moxley, Bill Schmitt, Vipin Guptda, and Jatin Misra.
Contents

Cover page 1
Abstract 3
Acknowledgments 5
Contents 7
List of Figures 13
List of Tables 17

1 Introduction to motif discovery 19
  1.1 Introduction ................................................. 19
  1.2 Fundamental tenets of motif discovery ....................... 27
  1.3 Motif discovery in sequential data ....................... 28
  1.4 Grammatical models of sequences ....................... 31
  1.5 Tools for motif discovery .................................. 55

2 Design of antimicrobial peptides 71
  2.1 Introduction ................................................. 71
  2.2 Motivation ................................................. 72
  2.3 A grammatical approach to annotating AmPs ............. 77
    2.3.1 Collecting a database of antimicrobial peptides .... 77
    2.3.2 Finding more antimicrobial peptides ............. 80
3 A generic motif discovery algorithm

3.1 Introduction ......................................................... 111
3.2 Motivation ......................................................... 112
3.3 Algorithm .......................................................... 116
   3.3.1 Preliminary definitions and nomenclature ................. 117
   3.3.2 Comparison phase ........................................... 120
   3.3.3 Clustering phase ............................................ 122
   3.3.4 Convolution phase .......................................... 123
3.4 Implementation ................................................ 124
   3.4.1 Choice of clustering function .......................... 124
   3.4.2 Summary of user–supplied parameters ................. 125
   3.4.3 Availability .................................................. 125
   3.4.4 Motif Significance ......................................... 126
   3.4.5 Proof of exhaustive maximality ......................... 127
   3.4.6 Two simple examples .................................... 128
3.5 Application .......................................................... 135
   3.5.1 Motif discovery in amino acid sequences ................ 136
   3.5.2 Motif discovery in protein structures .................. 138
4 Other exercises in motif discovery

4.1 Introduction

4.2 Biogrep: a tool for matching regular expressions

4.3 The evolution of updated BLOSUM matrices and the Blocks database

4.4 Bioinformatics and handwriting/speech recognition: unconventional applications of similarity search tools

4.5 Machine learning approaches to modeling the physiochemical properties of peptides

4.6 Identifying functionally important mutations from phenotypically diverse sequence data
A Abbreviations and reference data

A.1 Basic molecular biology data ........................................ 215
A.2 Supplementary data and analyses .................................. 218
   A.2.1 Position weight matrix computation and matching ......... 218
   A.2.2 Antimicrobial design data .................................... 224

B Gemoda file documentation

B.1 Introduction ....................................................... 225
B.2 align.c File Reference ............................................ 226
B.3 bitSet.c File Reference ........................................... 230
B.4 convll.c File Reference ........................................... 249
B.5 convll.h File Reference ........................................... 279
B.6 FastaSeqIO/fastaSeqIO.c File Reference ......................... 290
B.7 FastaSeqIO/fastaSeqIO.h File Reference ......................... 298
B.8 gemoda-r.c File Reference ....................................... 301
B.9 gemoda-s.c File Reference ....................................... 312
B.10 matdata.h File Reference ........................................ 329
B.11 matrices.c File Reference ...................................... 330
B.12 matrices.h File Reference ...................................... 332
B.13 matrixmap.h File Reference .................................... 341
B.14 newConv.c File Reference ..................................... 343
B.15 patStats.c File Reference ..................................... 356
B.16 patStats.h File Reference ..................................... 370
B.17 realCompare.c File Reference .................................. 372
B.18 realCompare.h File Reference .................................. 379
B.19 realIo.c File Reference ....................................... 382
B.20 realIo.h File Reference ....................................... 406
B.21 spat.h File Reference .................................................. 410
B.22 words.c File Reference ................................................. 411

C Gemoda data structure documentation ...........................................
C.1 Introduction ........................................................................ 421
C.2 bitGraph_t Struct Reference ................................................ 421
C.3 bitSet_t Struct Reference .................................................... 423
C.4 cnode Struct Reference ....................................................... 425
C.5 cSet_t Struct Reference ...................................................... 427
C.6 fSeq_t Struct Reference ..................................................... 428
C.7 mnode Struct Reference ...................................................... 429
C.8 rdh_t Struct Reference ....................................................... 430
C.9 sHash_t Struct Reference ................................................... 432
C.10 sHashEntry_t Struct Reference .......................................... 434
C.11 sOffset_t Struct Reference ................................................ 436
C.12 sPat_t Struct Reference .................................................... 438
C.13 sSize_t Struct Reference .................................................. 440

Bibliography ........................................................................... 441
List of Figures

1-1 Exponential growth of sequencing throughput .......................... 20
1-2 Exponential growth of Genbank ............................................ 21
1-3 A sample Genbank record ..................................................... 23
1-4 Usage of “ome” and “omic” words over time ............................ 24
1-5 Finding patterns in generic data .............................................. 29
1-6 Hairpin loops in DNA secondary structures ......................... 36
1-7 Noun–verb dependencies in various languages and their biological analogues . 40
1-8 Regular grammar describing the short hematopoietin receptor family 1 signature 45
1-9 Yeast 3’ splice sites .............................................................. 47
1-10 Yeast 3’ splice site pictogram and logo .................................. 54
1-11 pwmHits ................................................................. 56
1-12 Scanning phase of Teiresias ................................................ 63
1-13 Pattern discovery with Teiresias ........................................... 65
1-14 Schematic of the Gibbs sampling algorithm ......................... 69

2-1 Antimicrobial peptide action ................................................. 74
2-2 The structure of aurein ........................................................ 75
2-3 A phylogenetic tree of amphibian antimicrobial peptides .......... 76
2-4 Schematic of the bootstrapping method .................................. 81
2-5 Graph of the progress of the bootstrapping method ............... 84
2-6 Schematic of the grammar–based alignment method .............. 85
2-7 Example results of the grammar–based alignment method .......... 86
2-8 Directed evolution .............................................................. 90
Design space ................................................................. 93
Antimicrobial peptide action ........................................... 95
Gel picture ........................................................................ 97
Antimicrobial peptide hemolysis ........................................ 98
Example grammar–based dot plot ...................................... 103
Example grammar–based dot plot for computing Q .......... 104
Activity of rationally designed AmPs against *S. aureus* and *B. anthracis* .... 108
Alignment representing the LexA cis–regulatory binding site .... 115
A sketch showing the flow of the Gemoda algorithm for an example input set of protein sequences. ............................... 118
Pseudo–code for the Gemoda convolution ............................ 129
A natural language example illustrating the steps that Gemoda takes ............................................................ 131
A second natural language example ................................. 134
Guanosine–3′,5′–bis(diphosphate) 3′–pyrophosphohydrolase ((ppGpp)ase) (Penta–phosphate guanosine-3′–pyrophosphohydrolase) sequences .............................. 137
The RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences. .............................. 139
Logo representation of the RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences .................................................. 140
Logo representation of the RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences .................................................. 141
The similarity graph for the Gemoda 3.1.7.2 enzyme example .................................................. 142
Alpha carbon trace projection used by Gemoda .................. 143
A motif showing structural conservation between the human galactose-1-phosphate uridylyltransferase and fragile histidine triad proteins originally reported by Holm and Sander [121] .................................................. 144
Structural motif in Gemoda’s 3-D structure viewer ................ 145
The sequence logo for a) the motif implanted in each sequence for the (*l,d*)–motif problem and b) the LexA binding site motif generated from the highest–scoring motif returned by Gemoda. .................................................. 153
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-1</td>
<td>Characteristics of the BLOSUM matrices calculated from successive releases of the Blocks database</td>
</tr>
<tr>
<td>4-2</td>
<td>The relative performance of updated BLOSUM matrices</td>
</tr>
<tr>
<td>4-3</td>
<td>A complete set of Bayesian bootstrap replicates, with inset histogram of coverage difference</td>
</tr>
<tr>
<td>4-4</td>
<td>Plots of the differences in performance of updated RBLOSUM matrices</td>
</tr>
<tr>
<td>4-5</td>
<td>Coverage of a cleaned RBLOSUM matrix compared to the RBLOSUM64 matrix</td>
</tr>
<tr>
<td>4-6</td>
<td>Projection of a digit written with a PDA stylus into protein space</td>
</tr>
<tr>
<td>4-7</td>
<td>A Voice alignment of the spoken–letter “X” recorded from two different speakers</td>
</tr>
<tr>
<td>4-8</td>
<td>A phylogenetic tree of voice–proteins</td>
</tr>
<tr>
<td>4-9</td>
<td>Structure of the HIV–I protease, derived from the Protein Data Bank (PDB) entry 7HVP</td>
</tr>
<tr>
<td>4-10</td>
<td>Schematic of the HIV–I protease active site</td>
</tr>
<tr>
<td>4-11</td>
<td>Decision tree calculated for modeling 8–mer peptides</td>
</tr>
<tr>
<td>4-12</td>
<td>Classification results for all amino acid representations and model types</td>
</tr>
<tr>
<td>4-13</td>
<td>Structure of the PL–TET01 promoter</td>
</tr>
<tr>
<td>4-14</td>
<td>Schematic of the experimental procedure for promoter mutagenesis</td>
</tr>
<tr>
<td>4-15</td>
<td>Statistical distribution of mutations and their effects on mutant fluorescence</td>
</tr>
<tr>
<td>A-1</td>
<td>Amino acid structures and abbreviations</td>
</tr>
<tr>
<td>A-2</td>
<td>Nucleotide base structures and abbreviations</td>
</tr>
<tr>
<td>A-3</td>
<td>Gas and mass spectra for the synth–1 peptide</td>
</tr>
</tbody>
</table>
List of Tables

1.1 “Omic” fields other than genomic and proteomic ............................................. 22
1.2 Gene sequences from *Arabidopsis thaliana* ......................................................... 26
1.3 The construction of a position weight matrix ......................................................... 52
1.4 Motif discovery tools using regular expressions or similar models ....................... 59
1.5 Motif discovery tools using position weight matrices or similar models ............... 67
2.1 Common antimicrobial peptide families ............................................................... 79
2.2 Motif conservation ............................................................................................... 87
2.3 The preliminary design synthetic antimicrobial peptides used in this study ........... 91
2.4 Antimicrobial activity of the synthetic peptides against a variety of bacteria .......... 96
2.5 Minimum inhibitory concentration of the preliminary design synthetic AmPs against a variety of bacteria ................................................................. 97
2.6 Antimicrobial activity of rationally designed and shuffled peptides ..................... 107
2.7 Antimicrobial activity of rationally designed and shuffled peptides against *S. aureus* and *B. anthracis* ................................................................. 108
3.1 Performance on a range of *(l,d)*–motif problems with synthetic data .................. 157
4.1 Performance of Biogrep matching all the 1333 patterns in Prosite. ....................... 161
4.2 Misclassification results for the handwriting and speech recognition problems ... 182
4.3 Handwriting alignment scoring matrix .................................................................. 183
4.4 Machine learning model comparison .................................................................... 196
4.5 Machine learning model ranking .......................................................................... 197
4.6 Machine learning representation comparison ....................................................... 197
Chapter 1

Introduction to motif discovery

1.1 Introduction

The field of biology is changing rapidly from a qualitative discipline to one rooted in quantitation. These changes are driven by advances in microfabrication and microelectronics that continue to yield ever more creative ways to probe cellular function. These advances, in turn, are producing a deluge of data, opening up new ways to think about and analyze life, and attracting engineers and scientists from other disciplines into biology.

Nowhere is this sea change of quantitation more pronounced than in DNA sequencing [227]. As shown in Figure 1-1 on the following page, improvements in DNA sequencing drove down the cost of sequencing many orders of magnitude over the past 30 years, making sequencing commonplace. This rate of sequencing produces a data storage nightmare — each dry gram of DNA can store approximately a zettabyte of information, or a million million gigabytes [205]. Consider the growth of the Genbank DNA database [33], as shown in Figure 1-1 on the next page. Genbank receives over 1000 new submissions of DNA sequences from scientists every day and has doubled in size every 18 months since 1982.

This torrent of data is not restricted to DNA sequencing. Technological advances in recent years produced myriad tools for quantifying biology including DNA microarrays, reverse transcriptase polymerase chain reaction (RT–PCR), flow cytometry, chromatin–immunoprecipitation (chip–chip), yeast two–hybrid assays, fluorescence and confocal microscopy, generalized robotic screening methods, and countless others. These tools enable the continual coining of new
Figure 1-1: Exponential growth of sequencing throughput [227]. The figure tracks the number of nucleotides that can be sequenced for a dollar over time juxtaposed with the advancement of computing power. The hypothesis referred to as “Moore’s Law” — that computational power doubles every 18 months — appears somewhat applicable to DNA sequencing. Engineering advancements in the basic electrophoretic method of DNA sequencing, the so-called “Sanger sequencing,” over the past 30 years decreased the cost of sequencing many-fold. During the 15 year Human Genome Project, widespread investment into innovation and automation drove down the cost tenfold, greatly accelerating the completion of the project — 85% of the genome was sequenced in the project’s last two years [61].
1.1. INTRODUCTION

Figure 1-2: Exponential growth of Genbank [33]. Genbank is a comprehensive database of DNA sequences from over 200,000 organisms that were made public by direct submission from researchers. The database is maintained by the U.S. National Center for Biotechnology Information, and is updated daily. The sequences in Genbank include data from genome sequencing projects, expressed sequence tags, and international sequence data from the European Molecular Biology Laboratory and the DNA Databank of Japan. Genbank receives over 1000 submissions per day and has doubled roughly every 18 months since being started. It is the largest database of its kind and is the basis for many “information-added” databases and resources. See also 1-1 on the preceding page.
Table 1.1: “Omic” fields other than genomic and proteomic. In many cases, fields have been reborn as “omic” fields due to a changing focus towards higher throughput empirical methods. This list is indicative of the shift in biology towards quantitation and the resulting need for new, automated modes of analysis. The rise of these words over time in the scientific vernacular is shown in Figure 1.4 on page 24. This list was compiled by searching over 5 million articles in the life sciences literature using the NCBI eutils API [76]. See also references in the bibliography [93, 244].

<table>
<thead>
<tr>
<th>Anatomics</th>
<th>Biomics</th>
<th>Chromosomics</th>
<th>Cytomics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enviromics</td>
<td>Epigenomics</td>
<td>Fluxomics</td>
<td>Glycomics</td>
</tr>
<tr>
<td>Glycoproteomics</td>
<td>Immunogenomics</td>
<td>Immunomics</td>
<td>Immunoproteomics</td>
</tr>
<tr>
<td>Integromics</td>
<td>Interactomics</td>
<td>Ionomics</td>
<td>Lipidomics</td>
</tr>
<tr>
<td>Metabolomics</td>
<td>Metabonomics</td>
<td>Metagenomics</td>
<td>Metallomics</td>
</tr>
<tr>
<td>Metalloproteomics</td>
<td>Methylomics</td>
<td>Mitogenomics</td>
<td>Neuromics</td>
</tr>
<tr>
<td>Neuropeptidomics</td>
<td>Oncogenomics</td>
<td>Peptidomics</td>
<td>Phenomics</td>
</tr>
<tr>
<td>Phospho-proteomics</td>
<td>phosphoproteomics</td>
<td>Physiomics</td>
<td>Physionomics</td>
</tr>
<tr>
<td>Post–genomics</td>
<td>Postgenomics</td>
<td>Pre–genomics</td>
<td>Rnomics</td>
</tr>
<tr>
<td>Secretomics</td>
<td>Subproteomics</td>
<td>Surfaceomics</td>
<td>Syndromics</td>
</tr>
<tr>
<td>Transcriptomics</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“omes” such as the transcriptome, proteome, and interactome — high–throughput counterparts to traditional areas of study in biology. (See Figure 1.4 on page 24 and Table 1.1.) These new fields do not exist in isolation, but instead contribute to an ever–increasing network of information. Consider the Genbank annotation of the human insulin gene as shown in Figure 1.3 on the facing page. The annotation includes detailed information about insulin culled from the scientific literature by human experts including not just the sequence of the gene, but also its post–translational modifications, cellular localization, interactions with other genes, role in energy metabolism and diabetes, and numerous links to external databases with yet more information.

The Genbank annotation of insulin is the rule rather than the exception. It is a small piece of the growing wealth of data describing life processes from the molecular level all the way to the organism and ecological levels. However, inasmuch as these data hold promise, they also present challenges: if our ability to collect data has increased exponentially, has our ability to interpret and derive meaningful conclusions from these data kept track? Will these diverse data allow us to “connect the dots,” to reveal rich systems–level information? Or, instead, are they
Figure 1-3: A sample Genbank record showing the litany of features annotating the human gene for insulin. As the figure suggests, this “extra” information typically dwarfs the gene sequence in size. The annotations are highly cross-referenced, linking genes to cellular functions, behaviors, localizations and to other databases with yet more information. Viewed *en masse*, a complex network of knowledge emerges linking primary sequences into the understanding of higher-order systems at the cellular, tissue, and organism levels.
Figure 1-4: Usage of “ome” and “omic” words over time divided into three categories: genome and genomic, proteome and proteomic, and other ome and omic. The latter category comprises those words shown in Table 1.1 on page 22.
subject to the law of diminishing returns? The prevailing punditry lends credence to the former: witness the rise of systems biology [126, 127, 216].

But the problem remains: how can the potential of these voluminous and diverse data sets be realized? The sheer amount of data points to the need for automated, computational methods for analysis. This need is driving the co-opting of computer science as a core discipline of biology. That is, computational methods are becoming increasingly necessary to analyze the vast data sets biologists have accrued, and consequently, computer science and mathematics are becoming as fundamental to biology as chemistry and physics. The particular sub-discipline of biology devoted to these computational methods is typically referred to as either bioinformatics or computational biology. Together, these fields comprise many research topics devoted to both data analysis and modeling of biological systems.

It is likely that the need for automated, computational analysis will only increase in the future. There are 334 completely sequenced genomes and over 700 genomes in various stages of completion in the genome database at the U.S. National Center for Biotechnology Information (NCBI) [267]. But, of the finished genomes, only two are mammalian: Homo sapiens [145, 256] and Mus musculus [265], human and mouse. This suggests that, rather than being in the “post-genomic” age, we have just begun the genomic age. Upcoming years will see the completion of many genomes with relevance to human health and disease — such as the rat, rabbit, and chimpanzee genomes — and to industry such as the cow, corn, and potato genomes. Furthermore, these are just the sequencing data. Analogous progress in fields such as metabolomics and proteomics is likely to leave biologists awash in data, further exacerbating the need for automated methods of interpreting and modeling these data.

The motivation for automated data analysis techniques that I have painted in this introduction is quite broad, but what remains of this thesis is not. The focus of the remainder is automated, computational methods for discovering motifs in sequential data such as DNA and protein sequences. For example, imagine a scenario in which we would like to correlate one of the annotations of insulin shown in Figure 1-3 to a particular part of the DNA that encodes insulin. This is a very common problem in bioinformatics. In essence, this is like trying to learn a language we don’t speak by reading many books. As suggested by Figure 1.2 on the next page, this is a nontrivial problem.
Table 1.2: Genes sequences from *Arabodopsis thaliana*, a popular plant model organism. These very small genes are only a few hundred bases long, whereas a typical gene can be many kilobases in length. Given these sequences, how can we find all the small repeated motifs such as `CCACGCGTCCGAAAA`? At a glance, the task seems difficult. Digging further it seems insurmountable — the sequences below are just a small snippet of Genbank. To print just the nucleotide sequences in Genbank using this font would require 30 million pages: a stack of paper 3 km high, or roughly the distance from MIT to Harvard.

<table>
<thead>
<tr>
<th>&gt;gi</th>
<th>8571926 Arabidopsis thaliana lipid transfer protein</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCACGCGTCCGAAAAACACAGGCGGATCAGATCAGTCTAGCCAAAAGTTGGAATGATGGCTAATTCATCAGGCTCAAACACCTGCAAAAGAGAAGGAGGGGGTGCCACCTCCGTGCTGTGCAGGAGTCAAAAAGTTGAACGGTATGGCTCAAACCACACCCGACC GCCAACAAGCTCAGATGCTTACCTCCGCTGCTGTGCAGGAGTCAAAAAGTTGAACGGTATGGCTCAAACCACACCCGACC GCCAACAAGCTCAGATGCTTACCTCCGCTGCTGTGCAGGAGTCAAAAAGTTGAACGGTATGGCTCAAACCACACCCGACC</td>
<td></td>
</tr>
<tr>
<td>&gt;gi</td>
<td>8571922 Arabidopsis thaliana lipid transfer protein</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>CCACGCGTCCGAAAAACACAGGCGGATCAGATCAGTCTAGCCAAAAGTTGGAATGATGGCTAATTCATCAGGCTCAAACACCTGCAAAAGAGAAGGAGGGGGTGCCACCTCCGTGCTGTGCAGGAGTCAAAAAGTTGAACGGTATGGCTCAAACCACACCCGACC GCCAACAAGCTCAGATGCTTACCTCCGCTGCTGTGCAGGAGTCAAAAAGTTGAACGGTATGGCTCAAACCACACCCGACC GCCAACAAGCTCAGATGCTTACCTCCGCTGCTGTGCAGGAGTCAAAAAGTTGAACGGTATGGCTCAAACCACACCCGACC</td>
<td></td>
</tr>
<tr>
<td>&gt;gi</td>
<td>8571922 Arabidopsis thaliana lipid transfer protein</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>CCACGCGTCCGAAAAACACAGGCGGATCAGATCAGTCTAGCCAAAAGTTGGAATGATGGCTAATTCATCAGGCTCAAACACCTGCAAAAGAGAAGGAGGGGGTGCCACCTCCGTGCTGTGCAGGAGTCAAAAAGTTGAACGGTATGGCTCAAACCACACCCGACC GCCAACAAGCTCAGATGCTTACCTCCGCTGCTGTGCAGGAGTCAAAAAGTTGAACGGTATGGCTCAAACCACACCCGACC GCCAACAAGCTCAGATGCTTACCTCCGCTGCTGTGCAGGAGTCAAAAAGTTGAACGGTATGGCTCAAACCACACCCGACC</td>
<td></td>
</tr>
</tbody>
</table>
Motif discovery in sequential data is only one tiny sliver of the vast spectrum of topics spanned by bioinformatics and computational biology. However, many of the principles and tools I describe have broader implications for learning and automated discovery methods in biology. Chapter 1 of this thesis is devoted to familiarizing the reader with basic concepts of motif discovery, with a particular focus on linguistic methods of modeling sequences. In Chapter 2, these concepts are applied to the annotation and design of novel antibiotics, called antimicrobial peptides. The principal contribution of this thesis is the third chapter, in which I develop a framework and software tool for motif discovery that can be generalized to diverse types of data and is superior to existing tools in many ways. Finally, the last chapter comprises a series of vignettes that take a more broad approach to motif discovery and explore a number of issues adjoining the central theme of the first three chapters.

1.2 Fundamental tenets of motif discovery

I will begin with a series of elementary, almost philosophical examples that serve to illustrate some of the fundamental tenets of motif discovery. These examples may seem pedantic; however, the tenets they illustrate will be recurring themes throughout this thesis. Further, the following sections will serve to establish a common vocabulary and mode of thought that will enable the development of more complex ideas in later chapters.

In the most basic sense, the task of motif discovery is to find a repeated feature in a data set. Take, for example, the objects below.

\[ \square \triangle \square \square \]

(1.1)

What features are repeated at least twice in the objects? One answer is that the square shape appears three times. Yet another, is that three of the objects are darkened. And finally, a more sophisticated answer is that all of the objects are regular polyhedrons. Which of these is correct? All three are. The first two answers are intuitively obvious, whereas the final answer is rooted in a knowledge of geometry. The degree to which one answer is “more correct” than the others depends on what kinds of features we are interested in a priori. This is the first and most important tenet of motif discovery.
Consider a second question: is the dark square more similar to the dark triangle or to the hollow square?

\[ \square \sim \triangle \sim \square \]  

Again, there is no correct answer. The response obviously depends on our relative preference for color versus shape. This variety of question is even more difficult when we seek to quantify the degree of similarity or difference — the distance — between two objects. Is a human more similar to an alligator, or to an elephant? Based on body temperature, the human is more similar to the elephant; however, based on weight, the opposite true. This is the second tenet of motif discovery: the measurement of “distance” between objects is inherently relative, or dependent on predefined metrics.

Now consider a more complicated example shown schematically in Figure 1-5 on the facing page. The figure shows an X–Y scatter plot with 12 data points. How many groups of points, or clusters, are in this figure? As before, there are many correct answers: three groups of four; one group of four and one group of eight; or one single group of twelve. The answer depends on our preconceived notion of how small the distance between objects must be in order for them to be grouped together. Or, equivalently, how similar objects must be to be considered the same. This is our third and final tenet of motif discovery.

The three tenets we have just developed can be rephrased as follows. The answer to any motif discovery problem will always depend on what kinds of motifs we are looking for and the search for these motifs will always depend on a predefined metric of similarity and method for grouping together similar objects.

### 1.3 Motif discovery in sequential data

In this section, I define “sequential data” and introduce some basic concepts of motif discovery in sequential data. I will build upon tenets developed in the previous section and develop more complex ideas in motif discovery that are specific to sequential data.

In the most general sense, sequential data are any data in which there is a natural ordering, such that rearranging the data would result in lost information. In later chapters, we will deal
Figure 1-5: Clustering to find patterns in generic data. Given this data set, we would cluster the data together to find patterns (indicated here by the ellipses). But, there are many different patterns we could find. Which ones are correct?
with the more general case of sequential data that encompasses series of multidimensional real–
valued data; however, for the time being, we can consider sequential data to be a series of
alphanumerical characters such as the four sequences shown below.

\[
\begin{align*}
djndfduckdeicfjnfmABRAHAMLINCOLN \\
idkeioddsnABRAHAMLINCOLNaknkwbad \\
ioxcABRAHAMLINCOLNabkJwlkdaxlakj \\
xkasnkjlfABRAHAMLINCOLNkdsjkjsdl
\end{align*}
\]  

(1.3)

We can refer to these sequences more formally by calling the set of sequences \( S \), where \( S \) is
defined such that \( S = \{s_0, s_1, s_2, \ldots, s_n\} \), and where sequence \( s_i \) has length \( W_i \). So, in the above
example, \( n = 3 \) and \( s_0 = djndfduckdeicfjnfmABRAHAMLINCOLN \). Furthermore, let
the the \( j^{th} \) member of the \( i^{th} \) sequence be denoted by \( s_{i,j} \). So, \( s_{0,0} = d, s_{0,1} = j \), etc. Each \( s_{i,j} \) is a
primitive, or atomic unit, for the data that is being analyzed. For now, we will say that the prim-
itives are alphanumerical characters selected from some alphabet \( \Sigma \). In the example above, this
alphabet is the set of 52 lowercase and uppercase characters from the Roman alphabet. How-
ever, this alphabet can be defined in many different ways depending on the context of our motif
discovery problem. For example, for a DNA sequence the alphabet would comprise the charac-
ters \( \{A, T, G, C\} \), representing the four bases found in DNA: adenosine, thymine, guanine, and
cytosine (see Figure A-2 on page 217 in the Appendix. Or, for protein sequences, the alphabet
would be the set of characters representing the one letter abbreviations for the 20 naturally oc-
curring amino acids \( \{A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V\} \), which are
defined in Figure A-1 on page 216 in the Appendix.

In the set of sequences above, the motif “ABRAHAMLINCOLN” is inherently obvious in
the otherwise random series of characters that make up each of the individual sequences. A
similar motif is obvious even when the sequences are “mutated” as shown below.
In two of these sequences, Abraham is abbreviated but is still recognizable. But now, it is not appropriate to call the motif simply “ABRAHAMLINCOLN.” At this point, it is important to develop a more rigorous definition of “motif” that will allow us to describe all the possible permutations shown in the above mutated sequences. A motif is, henceforth, a mathematical model used to describe a set of locations in a set of sequences. These locations are referred to as “motif instances.” In the example above, the motif instances are the substrings: “ABELINCOLN,” “ABRAHAMLINCOLN,” “ALINCOLN,” and “ABRAHAMLINCOLN” in sequences \( s_0, s_1, s_2, \) and \( s_3 \), respectively.

Any mathematical model used to describe these instances should say that each instance begins with the letter A, optionally followed by either BE or BRAHAM, and necessarily followed by LINCOLN. That is, a model that describes the ordered arrangement of objects, in this case characters. Such models are commonplace in the field of linguistics and are called grammars.

### 1.4 Grammatical models of sequences

**Introduction**

The theory underlying grammatical models of sequences can be traced back to Noam Chomsky’s early work on syntax theory \([53–55]\). Chomsky’s work is the basis for much of formal language theory and computational linguistics. However, in general, most research in these areas has used grammars for pattern recognition vice pattern discovery, and focuses on machine learning techniques for computer understanding of natural languages. Only recently have these pattern recognition techniques been applied to problems of interest to biologists \([222–224]\).
As I will show in the following sections, most motif discovery algorithms in bioinformatics use motif models that can be reduced, in general, to a grammatical model. However, for the most part, such reductions are rare in the bioinformatics literature. This is because motif discovery and bioinformatics evolved independently from formal language theory. In a sense, the two fields are distant homologs of each other, both making use of models that can be reduced to grammars.

A grammar is a mathematical construct that describes the ordered arrangement of objects, usually words or characters, in a sequence \([3]\). More rigorously, a grammar is a 4–tuple \(G = (N, \Sigma, P, S)\) wherein

1. \(N\) is a finite set of non-terminal symbols, also called variables or syntactic categories;

2. \(\Sigma\) is a finite set of terminal symbols, disjoint from \(N\);

3. \(P\) is a finite subset of

\[
(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*,
\]

(1.5) each element in \(P\) is called a “production” and is written in the form \(\alpha \rightarrow \beta\); and,

4. \(S\) is a special symbol in \(N\) that is the start symbol.

To illustrate how a grammar can be used to model sequences, consider the following simple grammar:

\[
G = (\{\alpha, S\}, \{0, 1\}, P, S),
\]

(1.6) where the set of productions, \(P\), is given by

---

\(^1\)The star symbol, \(*\), is called the Kleene star and is interpreted as “zero or more” of the expression it follows. For example, \(ZA^*\) would be interpreted as a \(Z\) followed by zero or more \(A\) characters. The Kleene star and other similar operators will be discussed later in the section on regular grammars and regular expressions \(1.4\) on page \(42\).
Each line in the set of productions is essentially a replacement rule. For example, the op-
eration \( S \rightarrow 0\alpha1 \) should be read as “replace the character \( S \) with the sequence of characters \( 0\alpha1 \).” Then, the subsequent line should be read as “replace the two character sequence \( 0\alpha \) with the sequence \( 00\alpha1 \).” Finally, the last production should be read as “replace the charac-
ter \( \alpha \) with the character \( e \), which is the termination character.” These productions follow a
few conventions that are used throughout this manuscript. First, as defined on page 32, \( S \) is
a special non–terminal symbol that is always used as the starting symbol. Second, in order to
distinguish terminal symbols from non–terminal symbols, the former will always be displayed
in a fixed width font. Third, the character \( e \) is used always to represent the termination of a
sequence and is referred to as the termination character.

Now consider how to construct a sequence using the set of productions in the grammar
shown in Equations 1.6 and 1.7. Starting with the special non–terminal character \( S \), the first
production produces a three letter sequence.

\[
P = \begin{cases} 
  S \rightarrow 0\alpha1 \\
  0\alpha \rightarrow 00\alpha1 \\
  \alpha \rightarrow e. 
\end{cases}
\]  

(1.7)

Using the second production, produces a four character sequence.

\[
S \Rightarrow 0\alpha1 \Rightarrow 00\alpha1
\]

Finally, using the third production terminates the sequence.

\[
S \Rightarrow 0\alpha1 \Rightarrow 00\alpha1 \Rightarrow 001
\]  

(1.8)

The sequences produced by following the production rules of the grammar are called deriva-
tions of a grammar, or equivalently the sentences or sentential forms of the grammar. The collection of sentential forms of a grammar are collectively called the language generated by \( G \), or \( L(G) \).

Notably, the final sequence shown in Equation 1.8 is not the only sequence that can be derived from the grammar \( G \). Because the symbol \( \alpha \) appears in two different productions, either production can be used and neither production is preferred \textit{a priori}. For example the following sequence is an equally valid derivation of the same grammar.

\[
S \Rightarrow 0\alpha \Rightarrow 00\alpha \Rightarrow 000\alpha \Rightarrow 0000\alpha \Rightarrow 000001
\]

(1.9)

As the derivation above suggests, any sequence that

- begins with a 0,

- that is followed by zero or more 0s, and

- is terminated by a single 1

could be a derivation of the grammar shown in Equation 1.6 on page 32. Collectively, these derivations form the language of the grammar.

Now I will return to the Abraham Lincoln example shown in Equation 1.4 on page 31. Recall that the motif instances are the substrings “ABELINCOLN,” “ABRAHAMLINCOLN,” “ALINCOLN,” and “ABRAHAMLINCOLN” in sequences \( s_0, s_1, s_2, \) and \( s_3 \), respectively. A grammar that describes these motif instances is

\[
G = (\{\alpha, S\}, \{A, B, C, E, H, I, L, M, N, R\}, P, S),
\]

(1.10)
where $P$ is given by the set of productions

\begin{equation}
P = \left\{ 
\begin{align*}
S & \rightarrow A\alpha \\
\alpha & \rightarrow \beta \\
\alpha & \rightarrow BE\beta \\
\alpha & \rightarrow BRAHAM\beta \\
\beta & \rightarrow LINCOLN\gamma \\
\gamma & \rightarrow e.
\end{align*}
\right.
\end{equation}

Again, in a manner similar to Equation 1.7 on page 33, the grammar shown in Equation 1.11 has a non–terminal character, $\alpha$, in multiple productions. In such cases, the production can usually be abbreviated using the “|” character, which is to be read as “or.” For example, the productions in Equation 1.11 can be written equivalently as

\begin{equation}
P = \left\{ 
\begin{align*}
S & \rightarrow A\alpha \\
\alpha & \rightarrow \beta \mid BE\beta \mid BRAHAM\beta \\
\beta & \rightarrow LINCOLN\gamma \\
\gamma & \rightarrow e.
\end{align*}
\right.
\end{equation}

Notice that the grammar shown in Equation 1.11 describes exactly all four of the motif instances in the sequences shown in Equation 1.4 on page 31. The three possible derivations of the grammar are shown below.

\begin{align*}
S & \Rightarrow A\alpha \Rightarrow AB \Rightarrow ALINCOLN\gamma \Rightarrow ALINCOLN \\
S & \Rightarrow A\alpha \Rightarrow ABE\beta \Rightarrow ABELINCOLN\gamma \Rightarrow ABELINCOLN \\
S & \Rightarrow A\alpha \Rightarrow BRAHAM\beta \Rightarrow ABRAHAMLINCOLN\gamma \Rightarrow ABRAHAMLINCOLN
\end{align*}

This is an ideal case. In general, in constructing any model describing any motif instances,
we would like to use a grammar that is sensitive for the instances — i.e. all the instances are derivations of $G$ — and specific for the instances — i.e. the language $L(G)$ includes few derivations that are not motif instances.

Now consider a more complicated case in which a grammar is used to model DNA sequences that are likely to assume a hairpin structure, such as those shown in Figure 1-6. Hairpins in DNA and RNA sequences play an important role in the regulation of many processes, including transcription and translation. A hairpin is essentially a structure that bends back upon itself and is held together by Watson–Crick pairing. The paired bases in the hairpin structure are referred to as the “stem” and the unpaired, bulging bases are referred to as the “loop” (see Figure 1-6).

In order to form a stem–loop, or hairpin structure, the two sequences in the stem must be reverse complements of each other. This type of relationship is captured in the following
The grammar shown in Equation 1.14 can describe any hairpin loop in which the stem consists of one or more complementary bases and the loop consists of zero or more bases. For example, consider the following derivation of the grammar.

\[
S \Rightarrow \alpha \\
\Rightarrow A\alpha T \\
\Rightarrow AG\alpha CT \\
\Rightarrow AGG\alpha CCT \\
\Rightarrow AGGC\alpha GCCT \\
\Rightarrow AGGCT\alpha AGCCT \\
\Rightarrow AGGCT\beta AGCCT \\
\Rightarrow AGGCTA\beta AGCCT \\
\Rightarrow AGGCTAAGCCT
\]

This derivation produces a sequence that can form a hairpin structure with a stem size of five base pairs and a loop of a single base pair.

The grammar shown in Equation 1.14 is more complex than the grammar used to model the Abraham Lincoln motif (Equation 1.10 on page 34), because there are a long–range depen-
dencies in the sequences. That is, a particular base produced by the grammar in Equation 1.14 is guaranteed to be complementary to a base on the other side of the sequence. In contrast, the productions used to model the Abraham Lincoln motif produced a set of simple derivations in a left–to–right order. Indeed, even more complicated grammars can describe still more long–range, complex interactions between the characters in a sequence.

Hierarchy of restricted grammars

Linguists classify grammars into four increasingly complicated groups based on the format of their productions. A grammar is

1. right–linear, or type–3, if each production in P is of the form $A \to xB$, where $A$ and $B$ are in $N$ and $x$ is any string in $\Sigma*$;

2. context–free, or type–2, if each production in $P$ is of the form $A \to \alpha$, where $A$ is in $N$ and $\alpha$ is in $(N \cup \Sigma)^*$;

3. context–sensitive, or type–1, if each production in $P$ is of the form $\alpha A \beta \to \delta y \Gamma$, where $A$ is in $N$, $y$ is non–null, and $\alpha$, $\beta$, $\delta$, and $\gamma$ are in $(N \cup \Sigma)^*$;

4. unrestricted, or type–0, if it adheres to none of these restrictions.

This classification system is referred to as the Chomsky hierarchy [53]. Each of these grammars defines a corresponding class of language, which is the set of all sequences that can be produced using a particular type of grammar.

Right–linear, or type–3 grammars are also called “regular” grammars and are the simplest type of grammar. These grammars are called right–linear because derivations of these grammars are produced stepwise from left to right, never growing from the center of the sequence as in the derivation shown in Equation 1.16. As I will show in Section 1.4 on page 42, despite their simplicity, regular grammars are the most frequently used motif model in bioinformatics.

Context–free grammars are the next most complicated class of grammatical model. Indeed, the hairpin grammar shown in Equation 1.14 on the page before is a context–free grammar. This type of grammar is characterized by “nested” dependencies (see Figure 1-7). The dependencies
are nested in the sense that derivations of the grammar “grow” from the center, due to the structure of the productions.

Context–sensitive grammars and unrestricted grammars are the most complex classes of grammatical models. As shown in Figure 1.7 on the following page, context–sensitive grammars are characterized by long–range dependencies that are “crossing.” Derivations of these grammars can typically “grow” from anywhere inside the sequence. For example, consider the following grammar that describes a card player arranging a deck of cards:

\[
G = (\{\gamma, \beta, S\}, \{\spadesuit, \heartsuit, \diamOND\}, P, S),
\]

where P is given by

\[
P = \begin{cases} 
S & \rightarrow \gamma \\
\gamma & \rightarrow \spadesuit\heartsuit | \spadesuit\gamma\beta\diamOND \\
\spadesuit\beta & \rightarrow \beta\diamOND \\
\heartsuit\beta & \rightarrow \heartsuit\heartsuit 
\end{cases}
\]

This grammar is one of the most simple context–sensitive grammars. As well, it serves to illustrate that sequential data are not restricted to characters per se. Indeed, in Chapter 3 on page 111, I will extend the definition of sequential data to include ordered arrangements of multidimensional real–valued data sampled from a continuous distribution. Returning to the
**Figure 1.7:** Noun–verb dependencies in various languages and their biological analogues. Part A) shows the sentence “Dick saw Jane help Mary draw pictures” translated grammatically into German and Dutch. That is, the words in the sentence are rearranged to reflect the rules of grammar in these two languages, but the sentence is not translated *per se*. As shown, the English version of the sentence has a relatively simple dependency structure between the nouns and verbs that can be modeled using regular grammars. In contrast, German and Dutch require more complicated grammatical models [43, 134, 228]. Part B) shows the biological analogue of the three sentences in Part A). Typically, restriction sites can be modeled using regular grammars, whereas complex DNA secondary structures require context-free or context-sensitive grammars [209]. In the first example, the arches are used to represent a “must be followed by” dependency. In the second two examples, they represent a “must be complementary to” dependency.
current example, consider the following derivation of this grammar:

\[
S \Rightarrow \gamma \\
\Rightarrow \spadesuit \gamma \heartsuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \gamma \heartsuit \heartsuit \spadesuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \spadesuit \gamma \heartsuit \heartsuit \heartsuit \spadesuit \spadesuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \spadesuit \spadesuit \gamma \heartsuit \heartsuit \heartsuit \heartsuit \spadesuit \spadesuit \spadesuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \gamma \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \gamma \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \gamma \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \gamma \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \gamma \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \gamma \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \gamma \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \\
\Rightarrow \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \gamma \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \heartsuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \spadesuit \\n(1.19)
\]

Notice that the derivation bears much similarity to the hairpin loop example shown in Equation 1.16 on page 37. However, as I showed earlier, hairpin loops can be described with a context-free grammar, which is more simple than the grammar used in the current playing card example. What distinguishes the two is the size of the “loop,” the series of hearts in this example. Here, any derivation of the grammar has exactly the same number of clubs as it does hearts and spades. That is, if there are \( n \) clubs, there must be \( n \) hearts followed by \( n \) spades as below.

\[
\begin{array}{c}
\text{exactly } n \text{ clubs} \\
\text{exactly } n \text{ hearts} \\
\text{exactly } n \text{ spades}
\end{array}
\]

(1.20)

In contrast, the hairpin loop example introduced earlier was allowed to have an arbitrary number of intervening nucleotides. The extra restriction in this case can be thought of as a three-way dependency between the first clubs card, the first hearts card, and the first spades card. The same is true for the second cards in the succession, resulting in crossing dependencies, much like the
Dutch example in Figure 1-7 on page 40. The moral of this example is that subtle changes in the structures that need to be modeled can have a profound effect on the appropriate choice of grammars.

**Regular grammars and regular expressions**

**Building regular grammars and regular expressions**

For many applications in bioinformatics and computer science, regular grammars are an appropriate motif model and more complicated context–free or context–dependent grammars are not required. For example, most compilers make wide use of regular grammars to interpret programming languages, such as C, C++, or Java. That is, these programming languages are regular languages in the mathematical sense — they have a rigid structure and lack long–range dependencies.

Similarly, there are many phenomena in biology that can be modeled using regular grammars. For example, restriction enzymes, used for cutting DNA and RNA, typically recognize a set of motif instances that are easily modeled using regular grammars (see Figure 1-7 on page 40, part B).

In such cases, regular grammars are a convenient tool for two reasons. First, it is computationally simple to determine whether or not a string is a derivation of the given grammar, i.e. if the string is in the language of the grammar. This is not the case for more complicated grammars. In general, the computational complexity of this task rises rapidly for more complicated grammars and can take arbitrarily long for unrestricted grammars. Second, regular grammars can be represented compactly using a form called a regular expression. Consider the BstSFI restriction sites shown in Figure 1-7, reproduced below.

\[
\begin{align*}
\text{CGGCCG} \\
\text{CGGTCG} \\
\text{CGACCG} \\
\text{CGATCG} \\
\end{align*}
\]
The sequences are described by the following regular grammar:

\[ G = (\{\alpha, \beta, \gamma, S\}, \{A, G, T, C\}, P, S), \]  

where \( P \) is given by

\[
P = \begin{align*}
S & \rightarrow CG\alpha \\
\alpha & \rightarrow A\beta \mid G\beta \\
\beta & \rightarrow C\gamma \mid T\gamma \\
\gamma & \rightarrow CG
\end{align*}
\]

This regular grammar can be represented much more succinctly in the following regular expression: \( CG[AG][CT]CG \). The regular expression should be read as “any string starting with a \( C \) and a \( G \), followed by either an \( A \) or a \( G \), followed by either a \( C \) or a \( T \), that ends with a \( CG \).” The term \( [AG] \) is called a bracketed expression and is used to indicate a production rule in which multiple characters are allowed. For example, the bracketed expression \( [ATGC] \) would indicate that any of the four nucleotides is permitted.

In order to introduce more complex features of regular expressions, consider the motif describing the short hematopoietin receptor family in Figure 1-8 on page 45. The motif is described by the following regular expression.

\[
[LIVF] \ldots [LIV][RK].(9,20)WS.WS\ldots[FYW].
\]

In this regular expression the individual characters represent amino acids (see Figure A-1 on page 216 in the Appendix). Here, the first bracketed expression \( [LIVF] \) indicates that leucine, isoleucine, valine, or phenylalanine are equally acceptable. The term “.” is called a “wild–card” and indicates that any amino acid is acceptable. \( \Sigma \) is the set of terminal symbols for a grammar. The next special term in Equation 1.24 is “.(9,20)” This term indicates that the wildcard should be repeated for between nine and 20 places. For example, the regular expression
consisting only of the term “KR (2, 4)” has the following derivations: KRR, KRRR, KRRRR. Note that the strings KR and KRRRRR are not derivations of the grammar.

Because it is a more compact representation, regular grammars are usually recorded in regular expression form. In contrast, more complex grammars cannot be represented as a simple series of characters and symbols. This ease with which they can be communicated has been one of the factors promoting the widespread use of regular expressions — it would be inconvenient to discover a new protein motif and not be able to record the motif in an easily interpretable form for publication.

The regular expression formalisms presented here, such as the bracketed expression and the wild–card, are not exhaustive. There are many more terms that increase the richness of regular expressions, such as the Kleene star, “*”, which means “zero or more of the preceding expression” and the “+” symbol, which means “one or more of the preceding expression.” For an exhaustive treatment of regular expressions, the reader is referred to publications by Sipser and Friedl.

**Matching regular grammars and regular expressions**

Thus far, I have described how regular expressions be used to model a set of motif instances. However, a very common task is to then use a regular expression to look through new, longer sequences for “matches,” i.e. subsequences of a given sequence that are derivations of the grammar that the regular expression encodes. For example, consider the following regular expression: A[KR].Q[LV]C. We would like to know if there are any derivations of this grammar within the sequence shown below.

$$FLGARRQLCVVKLAAKFQVCSCAKKWQCLVFPAVFGKV$$  \(\text{(1.25)}\)

A simpleminded approach to this problem is to start with a beginning of the sequence, at letter F, and ask whether or not a derivation of the grammar could start that position. Obviously, any derivation of the grammar must begin with an A, so the answer is “no.” Moving on to the first A in the sequence, we see that it is followed by a K, which is allowed by the grammar, and that the K can be followed by any character, etc. Following this procedure reveals three matches
Figure 1-8: Regular grammar describing the short hematopoietin receptor family 1 signature [118]. These proteins are mostly receptors for interleukin cytokines. They are selectively expressed in lymphoid tissues and are typically membrane-bound [190]. The region shown in the figure is characterized by the regular expression [LIVF] . . . . . . . . [LIV] [R] . (9, 20) WS . . . . [FYW]. This motif is required for proper protein folding, efficient intracellular transport, and cell-surface receptor binding. The motif is relatively sensitive for the receptor family; however, it misses the rodent thymic lymphopoietin protein receptors, which are in the same family. Furthermore, the motif is not as specific as it could be — as shown above, the motif matches five receptors for the leptin obesity factor, which are not in the same family. Notice that the bar at the top shows the degree of conservation at each position; the amino acids are colored to reflect their physiochemical properties; and, the bracketed expressions, such as [LIV], tend to group together amino acids with similar physiochemical properties.
of the regular expression in the sequence, which are underlined below.

\[ \text{FLGARRQLCVFKLAAKFQVCSKAKWQLCFPAVFGKV} \] (1.26)

In general, algorithms designed to match regular expressions against sequences or other kinds of text use an approach that is, at its core, the same as the simpleminded approach above. One such algorithm and piece of software is described in Section 4.2 on page 160.

**Position weight matrices**

**Building position weight matrices**

Despite their utility, regular grammars and regular expressions are not suitable for modeling all kinds of motifs. As I showed earlier, regular grammars cannot describe long–range, nested, or crossing dependencies between characters. However, there are also motifs where these dependencies do not exist and yet regular expressions are not accurate models.

Consider the collection sequences shown in Figure 1-9 on the facing page. This collection comprises numerous 3’ splice sites from the fission yeast *Schizosaccharomyces pombe*. Each sequence is seven nucleotides in length and straddles the intron/exon boundary in a gene. After transcription, these sites will form a “branch point” allowing the introns to be excised from the pre–RNA to form the mature mRNA.

Notice that to sensitively describe these sequences using a regular expression, we would use \([ATGC]\ [ATGC] \ [CT]\ T [ATG] A [CT]\). This motif will match all of the instances, but it could also match many more: based on the number of bracketed expressions, this regular expression would match 192 unique sequences.

Notice too that each column of the aligned instances shown in Figure 1-9 has a particular “preference” for one kind nucleotide. For example, all but 10 of the sequences have a thymine at the last position. But, in the motif \([ATGC] \ [ATGC] \ [CT]\ T [ATG] A [CT]\), either cytosine or thymine is allowed in the last position, without any preference. Obviously, this regular expression would be more specific if we labeled the last bracketed expression with these preferences, i.e. “either cytosine or thymine, but with a seven–fold preference for the thymine.”

Incorporating such preferences into the grammatical framework requires only minor changes.
Figure 1-9: Yeast 3’ splice sites. The figure shows the 3’ splice junction of 75 introns from the fission yeast *Schizosaccharomyces pombe*. The coloring of the sequences is used to show the degree of conservation. Notice that certain columns, such as the final column, have strong preferences for certain nucleotides. The final column has a strong preference for cytosine; however, thymine is allowed occasionally. The consensus sequence at the bottom shows the most common nucleotide at each position. If the nucleotide is not perfectly conserved it is shown in lowercase. This kind of motif is poorly modeled using regular grammars. Instead, position weight matrices preserve much of the “preference information.”
Recall from the definition on page 38 that a grammar is regular (or right-linear or type-3) if each production in $P$ is of the form $A \rightarrow xB$, where $A$ and $B$ are in $N$ and $x$ is any string in $\Sigma^*$. A similar set of restrictions defines a position weight matrix, which is a grammar in which each production in $P$ is of the form $A \xrightarrow{p_i} xB$, where $A$ and $B$ are in $N$ and $x$ is any character in $\Sigma$, and $p_i$ is the probability of production $i$. As well, $\sum_i p_i$ must equal one for all of the productions on which $A$ is on the left side. In loose terms, the position weight matrix, or PWM, can be thought of as a probabilistic regular expression. Using this new structure, the regular expression $[ATGC][ATGC][CT]T[ATG]A[CT]$ can be written as a PWM grammar,

$$G = ([S, \alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta], \{A, T, G, C\}, \{P, S\})$$

(1.27)
where P is the set of productions below.

\[
P = \begin{align*}
S & \rightarrow \alpha \\
\alpha & \xrightarrow{0.067} A\beta \\
\alpha & \xrightarrow{0.773} T\beta \\
\alpha & \xrightarrow{0.091} G\beta \\
\alpha & \xrightarrow{0.067} C\beta \\
\beta & \xrightarrow{0.617} A\gamma \\
\beta & \xrightarrow{0.240} T\gamma \\
\beta & \xrightarrow{0.120} G\gamma \\
\beta & \xrightarrow{0.013} C\gamma \\
\gamma & \xrightarrow{0.120} T\delta \\
\gamma & \xrightarrow{0.880} C\delta \\
\delta & \xrightarrow{1.000} T\epsilon \\
\epsilon & \xrightarrow{0.893} A\zeta \\
\epsilon & \xrightarrow{0.027} T\zeta \\
\epsilon & \xrightarrow{0.080} G\zeta \\
\zeta & \xrightarrow{1.000} A\eta \\
\eta & \xrightarrow{0.133} T \\
\eta & \xrightarrow{0.867} C
\end{align*}
\]
Equation 1.15 can be represented much more compactly as a frequency matrix

\[
f = \begin{pmatrix}
0.067 & 0.627 & 0.000 & 0.000 & 0.893 & 1.000 & 0.000 \\
0.773 & 0.240 & 0.120 & 1.000 & 0.027 & 0.000 & 0.133 \\
0.093 & 0.120 & 0.000 & 0.000 & 0.080 & 0.000 & 0.000 \\
0.067 & 0.013 & 0.880 & 0.000 & 0.000 & 0.000 & 0.867
\end{pmatrix}
\] (1.29)

in which each row corresponds to a single character in \( \Sigma \) and each column corresponds to a single non-terminal character in \( N \) (where \( \Sigma \) is disjoint from \( N \), as usual). So, in Equation 1.29, the rows correspond to A, T, G, and C; and the columns correspond to \( \alpha \), \( \beta \), \( \gamma \), \( \delta \), \( \epsilon \), \( \zeta \), and \( \eta \), where \( S \) was omitted.

Notice that a derivation of the grammar in Equation 1.27 on page 48 is necessarily also a derivation of the regular grammar \([ATGC][ATGC][CT][ATG]A[CT]\), and vice versa. As such, the two grammars describe the same language, or the set of all derivations. But, because the productions of Equation 1.27 are weighted by probability, certain derivations are more probable than others. The degree to which one derivation of the grammar is more probable than another is characterized by the derivation’s log-odds score. To compute the log-odds score, first requires a log-odds matrix, \( \Theta \), where

\[
\Theta_{ij} = \log_2 \left( \frac{f_{ij}}{q_j} \right).
\] (1.30)

The calculation of the frequency and log-odds matrices for the 3’ yeast splice sites is shown in Table 1.3. Here, \( \Theta \) is given by

\[
\Theta = \begin{pmatrix}
-1.907 & 1.326 & \emptyset & \emptyset & 1.837 & 2.000 & \emptyset \\
1.629 & -0.059 & -1.059 & 2.000 & -3.229 & \emptyset & -0.907 \\
-1.421 & -1.059 & \emptyset & \emptyset & -1.644 & \emptyset & \emptyset \\
-1.907 & -4.229 & 1.816 & \emptyset & \emptyset & \emptyset & 1.794
\end{pmatrix},
\] (1.31)

where \( \emptyset \) is used to indicate values that are undefined because \( f_{ij} = 0 \) and \( \log_2 0 \) is undefined. Given the log-odds matrix form of a PWM, the score of any derivation of the PWM is com-
puted merely by looking up values in $\Theta$. Consider the sequence AGCTGAC, which is both a derivation of the grammar shown in Equation 1.27 on page 48 and the first of the sequences shown in Figure 1-9 on page 47. The log–odds score for this sequence is

$$\text{score} = \Theta_{0,0} + \Theta_{2,1} + \Theta_{3,2} + \Theta_{1,3} + \Theta_{2,4} + \Theta_{0,5} + \Theta_{3,6}$$

$$= -1.907 - 0.059 + 1.816 + 2.000 - 1.644 + 2.000 + 1.794$$

$$= 4.000.$$

Notice that the score for a sequence that is not a derivation of the grammar is undefined, or, effectively $-\infty$. Table 1.3 on the following page shows the calculation of the frequency matrix, log–odds matrix, and the scoring of example sequences for this PWM. Also, a small program for calculating a PWM from a set of sequences is provided in Section A.2.1 on page 218 of the Appendix.

The “strength” of a PWM motif is measured by a quantity called its entropy. The motif entropy is the sum of the entropies of each column, or position in the motif. This entropy of a given column in a PWM is a measure of the disorder, or the randomness of the distribution of letters. The column entropy is measured in bits and is given by

$$h_i = - \sum_j f_{ij} \log_2 f_{ij}$$

where, $f_{ij}$ is the frequency matrix as shown in Figure 1.3 on the following page. The entropy of the whole motif is just the sum of the entropies of the columns:

$$H = - \sum_i \sum_j f_{ij} \log_2 f_{ij}.$$  \hspace{1cm} (1.34)

Typically, the entropy is measured relative to the background entropy. As above, the background entropy of a single column is

$$h_i^\circ = - \sum_j q_j \log_2 q_j$$  \hspace{1cm} (1.35)

where, $q_j$ is the a priori, background frequency of the letter denoted by index j. In the case of
Table 1.3: The construction of a position weight matrix from the collection of sequences shown in Figure 1-9 on page 47. Part A) shows the number of nucleotides of each type that occur in each of the seven positions of the aligned sequences. For example, in the first position, there are 58 thymines. Part B) shows the frequency matrix \( f \), where each \( f_{ij} = \frac{c_{ij}}{\sum_j c_{ij}} \). Part C) shows the log–odds matrix \( \Theta \), where each \( \Theta_{ij} = \log\left(\frac{f_{ij}}{q_j}\right) \) and \( q \) is the vector of background frequencies for the nucleotides. Part D) shows the scoring of three different sequences. To compute the score for a sequence, the corresponding nucleotide at each column is looked up in \( \Theta \) and the columns are summed together.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>T</th>
<th>G</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5</td>
<td>47</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>T</td>
<td>58</td>
<td>18</td>
<td>9</td>
<td>75</td>
</tr>
<tr>
<td>G</td>
<td>7</td>
<td>9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
<td>1</td>
<td>66</td>
<td>0</td>
</tr>
</tbody>
</table>

B) Frequency Matrix \( (f_{ij}) \):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>T</th>
<th>G</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.067</td>
<td>0.627</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>T</td>
<td>0.773</td>
<td>0.240</td>
<td>0.120</td>
<td>1.000</td>
</tr>
<tr>
<td>G</td>
<td>0.093</td>
<td>0.120</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>C</td>
<td>0.067</td>
<td>0.013</td>
<td>0.880</td>
<td>0.000</td>
</tr>
</tbody>
</table>

C) Log–odds Matrix \( (\Theta_{ij}) \):

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>T</th>
<th>G</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-1.907</td>
<td>1.326</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>T</td>
<td>1.629</td>
<td>-0.059</td>
<td>-1.059</td>
<td>2.000</td>
</tr>
<tr>
<td>G</td>
<td>-1.421</td>
<td>-1.059</td>
<td>∅</td>
<td>∅</td>
</tr>
<tr>
<td>C</td>
<td>-1.907</td>
<td>-4.229</td>
<td>1.816</td>
<td>∅</td>
</tr>
</tbody>
</table>

D) Example sequence scoring:

<table>
<thead>
<tr>
<th>Query</th>
<th>T</th>
<th>A</th>
<th>C</th>
<th>T</th>
<th>T</th>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.629</td>
<td>1.326</td>
<td>1.816</td>
<td>2.000</td>
<td>1.837</td>
<td>2.000</td>
<td>1.794</td>
</tr>
<tr>
<td>2</td>
<td>1.629</td>
<td>-0.059</td>
<td>1.816</td>
<td>2.000</td>
<td>1.837</td>
<td>2.000</td>
<td>1.794</td>
</tr>
<tr>
<td>3</td>
<td>-1.421</td>
<td>-0.059</td>
<td>∅</td>
<td>T</td>
<td>T</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

\[ \Sigma = 7.335 \]

\[ \Sigma = 11.017 \]

\[ \Sigma = ∅ \]
identically distributed nucleotides, each $q_j$ is 0.25; i.e. $q_A = 0.25$, $q_C = 0.25$, $q_T = 0.25$, and $q_G = 0.25$. The background entropy of the entire motif is

$$H^\circ = - \sum_i \sum_j q_j \log_2 q_j. \quad (1.36)$$

The difference between the background entropy and the motif entropy is referred to as the information content, $I$, of the PWM. Using Equations 1.33–1.36 above, the information content can be calculated as

$$I = H^\circ - H = - \sum_i \sum_j q_j \log_2 q_j - \left( - \sum_i \sum_j f_{ij} \log_2 f_{ij} \right) \quad (1.37)$$

$$= \sum_i \sum_j f_{ij} \log_2 \left( \frac{f_{ij}}{q_j} \right).$$

Notice that the information content of the motif is minimized when the nucleotide distribution for each column is exactly the background distribution of nucleotides. That is, when $f_{ij} = q_j$ for all $i$, the $\log_2 (f_{ij}/q_j)$ terms are zero. This makes sense intuitively: if the PWM describes the background distribution, the motif can obviously not be distinguished from the background and therefore contains no information. In this same case, the entropy of the motif is maximized and is equal to the background entropy.

PWMs are commonly represented by two varieties of schematics: pictograms and sequence logos. An example of each of these is shown in Figure 1-10 on the next page. A pictogram is essentially a visualization of the frequency matrix representing a PWM, whereas a sequence logo is a pictogram that is scaled to reflect the information content at each position in the PWM.

**Matching position weight matrices**

Thus far, I have shown how a PWM can be used to model a set of motif instances and how a derivation of a PWM grammar can be scored. PWMs can also be used to search in new, long sequences for regions of the sequence that appear to match the motif. This is accomplished by
A) Pictogram of the PWM in Table 1.3 on page 52

```
TACTA A C
```

B) Logo of the PWM in Table 1.3 on page 52

```
 2
 T

 1
 C

 0
 A
```

**Figure 1-10:** Yeast 3’ splice site pictogram and logo. Part A) shows the PWM in Table 1.3 on page 52 represented as a pictogram. At each position in the motif, the height of the nucleotides is scaled in proportion to their frequency in f, with the more frequent nucleotides always placed on top. The pictogram clearly shows that positions four and six are perfectly conserved, whereas the other positions are distributed between many nucleotides. Part B) shows a sequence logo representation of the same PWM. The sequence logo is a pictogram in which the height of each column is scaled in proportion to the information content contributed by that position to the motif (see Equation 1.37 on the page before). Taller columns have a nucleotide distribution that deviates strongly from the background distribution. In this sequence logo, the background distribution is arbitrarily set to equal *a priori* probability of each nucleotide. As such, the maximum information content in any column is two bits, which is achieved only in the two perfectly conserved positions of the motif.
“sliding” the PWM over the length of the sequence to look for subsequences that have high log–odds scores.

Consider searching the sequence TAGCTGACTGAC. To slide the PWM over this sequence is equivalent to evaluating the score of each seven nucleotide substring: TAGCTGA, AGCTGAC, GCTGACT, CTGACTG, TGACTGA, and GACTGAC. These are $\emptyset$, 4.0, $\emptyset$, $\emptyset$, $\emptyset$, and 5.871. That is, there are two matches for the PWM, one stronger than the other. This method can be used to search for a PWM in much larger sequences as well. For example, Figure 1-11 (page 56) shows the distribution of scores obtained by searching the PWM in Table 1.3 on page 52 against chromosomes 1–4 of the Saccharomyces cerevisiae genome.

1.5 Tools for motif discovery

Introduction

In general, the goal of motif discovery is to derive a set of grammars that sensitively matches a set of given sequences. This is the inverse of many of the examples in the previous section. That is, imagine a case in which you are given a set of derivations and then asked what kind of grammar could have produced the derivations? This is what is called grammar induction in the computational linguistics literature and is equivalent to guessing the grammar of an unknown language given a few sentences in the language.

In bioinformatics, this task is usually presented in a slightly more difficult form. To illustrate this, consider a hypothetical challenge in which a colleague hides derivations of the regular grammar $[KR]QTRP.[RT]K$ in a set of sequences that consist otherwise of random characters. You are presented with the sequences and asked what grammar was hidden therein.

\[
\begin{align*}
AJDFOASODVIKQTRPXKIIWEJSJ \\
JKQTRPCRXUCIQWEMFIOAKLGS \\
ADUHFIKACRQTRPXKMSKDIAOAS
\end{align*}
\]

Without any prior knowledge, this task is nearly impossible. The colleague could have hidden a
Figure 1-11: Distribution of log–odds scores obtained by searching the PWM in Table 1.3 on page 52 against chromosomes 1–4 of *Saccharomyces cerevisiae*. Of the nearly 3 million possible sites, only 46,000 had non-null scores. As the figure shows, the score distribution is roughly Gaussian. The dashed line indicates the information content of the PWM. Scores above this line are generally considered strong matches. PWMs are more specific than regular grammars, because the threshold above which a match is considered “true” can be varied. In contrast, with a regular grammar is either a match or not a match: there is no variable threshold.
regular grammar that consisted solely of $S$, which would be undetectable since there are many characters that occur in all of the sequences. Further, he could have hidden “....”, in which case there would be no evidence in the sequences. This conundrum is closely related to one of the three tenets of motif discovery developed in Section 1.2 on page 27: the answer to any motif discovery question is invariably dependent on at least some prior knowledge about what forms a motifs may take.

Suppose then that the colleague says the motif is at least five characters long, is a regular grammar, and all the derivations of the grammar look “pretty similar.” Given this information, a logical approach would be to look for subsequences of five characters that look relatively similar and occur in all three sequences. After diligently scanning the sequences, you can find two sets of three that seemed to fit this description: \{FIOAS, FIOAK, FIOAS\} and \{KQTRP, KQTRP, RQTRP\}. Knowing the answer, it is obvious that we are on the right track; however, again we are at an impasse. It would be easy to write a regular expression describing either of these sets. But, \textit{a priori} it is impossible to tell which set may be the correct answer. The first set has a $K$ that is mismatched with a $S$, whereas the latter set has a $K$–$R$ mismatch. If these were amino acid sequences we could say that lysine, $K$, is more similar to arginine, $R$, than it is to serine, $S$. Therefore, we might choose the latter set. This decision is related to the remaining two tenets of motif discovery developed in Section 1.2: the answer to any motif discovery problem will always depend on a predefined metric of similarity and a method for grouping together similar objects.

In the following sections, I review a number of approaches for problems such as the example given above, focusing on the two most common classes of approach: those that use regular expression motif models and those that use PWMs. All of these approaches, without exception, always require some degree of intelligent guidance by the user that can be reduced to the three tenets discussed above. In general, motif discovery tools that do not have such requirements have made assumptions on the user’s behalf.

\textbf{Teiresias and other regular expression–based tools}

Because they are convenient from both a computational perspective and from the perspective of communicating results, regular expressions are the most common form of motif model used
in bioinformatics. Table 1.4 on the facing page shows a list of publications introducing motif discovery tools in this class. Within the field, these algorithms are commonly referred to as “motif driven” or “pattern driven” algorithms [41]. At their most basic conceptual level, all of these approaches work by first enumerating possible patterns and then checking for the patterns in the sequence set [41].

There are three principal characteristics that distinguish between the various algorithms shown in Table 1.4, which are as follows:

1. The regular expression class complexity.

2. The completeness of the returned motif set. That is, does the algorithm return all patterns present in the input sequences?

3. The motif maximality. For instance, in the two strings “KYLEJ” and “KYLEL”, the motif “KYLE” is maximal, whereas “KYL” is not, because we could add an “E” without decreasing the number of times it occurs. In essence, maximality is a proxy for specificity.

The most important of these distinguishing features is the complexity of the regular expression class that an algorithm returns. No motif discovery tool can search for the “universe” of regular expressions. Recall from the previous section that “...” and other types of motifs will always be present, and therefore such a result is meaningless. Furthermore, enumerating regular expressions is NP-complete [90, 161], meaning that, in general, the runtime of a motif discovery tool will increase exponentially with the size of the sequence set it is given. As I showed in the previous section, the answer to any motif discovery problem will always require some a priori knowledge of the kinds of motifs that might be found, and simply specifying that the grammar is regular is not enough. Accordingly, most motif discovery tools restrict themselves to finding a particular subclass of regular expressions. This motif class determines the form of each pattern, \( p_i \) that we find. Below, a few motif classes, commonly used in biological sequence analysis, are enumerated in order of increasing complexity [82]:

- \( p_i \in \Sigma^* \): This is the class of all “solid” patterns, for example “KAGTPT” and “TAGCGGGAT”.

- \( p_i \in (\Sigma \cup \{\} \}^* \): This is the class of all patterns that can have “wildcard” positions, which
Table 1.4: Motif discovery tools using regular expressions or similar models. This list is not intended to be exhaustive; however, it includes many of the well-known motif discovery tools used in bioinformatics. Early methods tended to use consensus strings or simple word counting approaches, i.e. counting the occurrences of “n–mers” such as the 4–mer ATGC. Words that are statistically over–represented are called motifs. Later approaches used more complex regular expressions, cf. Rigoutsos and Floratos [207].

<table>
<thead>
<tr>
<th>Authors</th>
<th>Year</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Queen et al.</td>
<td>1982</td>
<td>[202]</td>
</tr>
<tr>
<td>Galas et al.</td>
<td>1985</td>
<td>[87]</td>
</tr>
<tr>
<td>Mengeritsky and Smith</td>
<td>1987</td>
<td>[169]</td>
</tr>
<tr>
<td>Staden</td>
<td>1989</td>
<td>[237]</td>
</tr>
<tr>
<td>Neuwald and Green</td>
<td>1994</td>
<td>[179]</td>
</tr>
<tr>
<td>Jonassen et al.</td>
<td>1995</td>
<td>[132]</td>
</tr>
<tr>
<td>Wolferstetter et al.</td>
<td>1996</td>
<td>[270]</td>
</tr>
<tr>
<td>Sagot et al.</td>
<td>1997</td>
<td>[215]</td>
</tr>
<tr>
<td>Rigoutsos and Floratos</td>
<td>1998</td>
<td>[82, 207]</td>
</tr>
<tr>
<td>van Helden et al.</td>
<td>1998</td>
<td>[252, 253]</td>
</tr>
<tr>
<td>Jacobs Anderson and Parker</td>
<td>2000</td>
<td>[128]</td>
</tr>
<tr>
<td>Marsan and Sagot</td>
<td>2000</td>
<td>[167]</td>
</tr>
<tr>
<td>Pevzner and Sze</td>
<td>2000</td>
<td>[195]</td>
</tr>
<tr>
<td>Bussemaker et al.</td>
<td>2000</td>
<td>[47]</td>
</tr>
<tr>
<td>Kielbasa et al.</td>
<td>2001</td>
<td>[138]</td>
</tr>
<tr>
<td>Horton</td>
<td>2001</td>
<td>[123]</td>
</tr>
<tr>
<td>Keich and Pevzner</td>
<td>2002</td>
<td>[137]</td>
</tr>
<tr>
<td>Eskin and Pevzner</td>
<td>2002</td>
<td>[78]</td>
</tr>
<tr>
<td>Buhler and Tompa</td>
<td>2002</td>
<td>[46]</td>
</tr>
<tr>
<td>Sinha and Tompa</td>
<td>2002</td>
<td>[232]</td>
</tr>
<tr>
<td>Price et al.</td>
<td>2003</td>
<td>[199]</td>
</tr>
<tr>
<td>Sinha</td>
<td>2003</td>
<td>[231]</td>
</tr>
<tr>
<td>Danilova et al.</td>
<td>2003</td>
<td>[64]</td>
</tr>
<tr>
<td>Ganesh et al.</td>
<td>2003</td>
<td>[88]</td>
</tr>
<tr>
<td>Liang et al.</td>
<td>2004</td>
<td>[151]</td>
</tr>
<tr>
<td>Fogel et al.</td>
<td>2004</td>
<td>[83]</td>
</tr>
<tr>
<td>Pavesi et al.</td>
<td>2004</td>
<td>[191]</td>
</tr>
<tr>
<td>Hernandez et al.</td>
<td>2004</td>
<td>[114]</td>
</tr>
<tr>
<td>Markstein et al.</td>
<td>2004</td>
<td>[166]</td>
</tr>
<tr>
<td>Frith et al.</td>
<td>2004</td>
<td>[86]</td>
</tr>
<tr>
<td>Sumazin et al.</td>
<td>2005</td>
<td>[240]</td>
</tr>
</tbody>
</table>
are denoted by ".", for example "K . G . PT" and "TA . . GGAT". The wildcard means that any character from the alphabet will suffice in that position.

• \( p_i \in (\Sigma \cup R)^* \): This is the class of all patterns that can have “bracketed” expressions, for example “K [ADG] G [KQ] PT” and “TA [GA] [TC] GGAT”. The bracketed expression “[TC]” means that either “T” or “C” will suffice in that position. In this notation, \( R \) represents the set of characters in the brackets, for example \( R = \{TC\} \) or \( R = \{GA\} \).

• \( p_i \in (\Sigma \cup .)^* \): This is the class of “flexible” patterns. For example “K . (1, 3) G . (2, 5) PT”, where “.(2,5)” means that anywhere between two and five wildcards can exist at that position, that is “.(2,5)” can be any one of {., ., ., ., ., .}.

In general, the more complex these patterns are, the more expressive the languages will be that we find. However, with increasing complexity, the computational difficulty of the motif discovery problem increases drastically [90, 161].

Also, for some of these tools, it is possible to guarantee the completeness of the set of returned patterns. That is, a particular tool may guarantee that all regular expressions meeting particular characteristics are discovered. However, this guarantee comes at the price of increased time and space complexity. That is, the set of all possible patterns is very large and can take a large space to enumerate and a long time to search through. As such, many motif driven algorithms use heuristics to limit the space of patterns that are searched.

Here, I will focus on the Teiresias algorithm as a representative regular expression–based motif discovery tool. Notably, Teiresias is the basis for much of the work in this thesis, particularly in Chapters 1 and 2. A more detailed description of Teiresias is available elsewhere [82, 207].

Given a set of sequences \( S = \{s_0, s_1, \ldots, s_n\} \), and integers \( L \), \( W \), and \( K \), Teiresias finds all patterns involving at least \( L \) non–wildcard characters that occur at least \( K \) times and have a fraction of non–wildcard characters of at least \( L/W \). This set of patterns is called \( \mathcal{C} \), where \( \mathcal{C} = \{p_1, p_2, \ldots, p_m\} \) and each \( p_i \in (\Sigma \cup \{.\}) \Sigma \). This is the set of all regular expressions that begin and end with a character, but may have an arbitrary number of wildcards and characters in the middle subject to the \( L \) and \( W \) restriction, for example, A\(XG\), A . G, K . R . G, etc. For each motif \( p_i \) in \( \mathcal{C} \), Teiresias returns an offset list \( \mathcal{L} (p_i) \) that specifies each sequence–position combination where the motif occurs (cf. Figure 1-13 on page 65).
The support of a motif is equal to the number of its occurrences (or, equivalently, “instances” or “embeddings”), \(|L(p)|\). Essentially, \(L\) defines the minimum size of patterns in which we are interested, and \(L/W\) defines the minimum specificity (the fewer wildcards, the more specific a motif). The four distinguishing characteristics of the Teiresias algorithm are as follows:

1. All maximal patterns are reported (see below for a definition of “maximal”).
2. Only the maximal patterns are reported.
3. Running time depends only on the number of patterns present in the data, that is it is output sensitive.
4. Patterns can be arbitrarily long.

The most important characteristic of Teiresias is that it returns the complete set of maximal patterns. And, because of the manner in which these patterns are handled internally by Teiresias, the algorithm runs very quickly.

In the Teiresias parlance, a maximal motif is a regular expression which has the following properties:

1. The motif cannot be made more specific without producing a motif with fewer embeddings (i.e., without \(|L(p)|\) decreasing); and
2. The motif is not missing any instances, i.e. \(L(p)\) 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. Here, “specific” has a particular meaning: a pattern \(p_i\) is more specific than \(p_j\) if \(p_j\) can be transformed into \(p_i\) by substituting one or more wild–cards for a character, or by appending wild–cards and characters to either side of \(p_j\). For example, CH.MEN..N is less specific than all of the following regular expressions: CHEMEN..N, CH.MEN..NE.R, and CH.MEN..IN. Necessarily, if a pattern \(p_i\) is more specific than a pattern \(p_j\), then

\[
|L(p_i)| \leq |L(p_j)|. \tag{1.39}
\]
Teiresias works in two phases: scanning and convolution. During the scanning phase, Teiresias enumerates all “elementary motifs” with exactly $L$ characters and at most $W - L$ wildcards (see Figure 1-12 on the facing page). Elementary motifs are short regular expressions that can be stitched together to form longer regular expressions that are more specific, using the definition of specificity above. For example, as shown in Figure 1-12, the sequences

$$
\text{KDQVQKRK} \\
\text{CWCQKRK} \\
\text{WDQKRKNP}
$$

have five motifs with 1) exactly $L = 3$ characters, 2) no more than $W - L$ wildcards for every window of $L = 3$ characters, and that 3) occur at least three times: $\text{W.QK}, \text{QKR}, \text{Q.K}, \text{KRK},$ and $\text{Q.KR}$. These are the elementary motifs.

In the convolution phase, the elementary motifs are stitched together to see if more specific motifs can be found. The process of convolution is defined as follows:

$$
p_k = p_i \oplus p_j
$$

$$
= \begin{cases} 
p_k' & \text{if } \text{prefix}_L(p_i) = \text{prefix}_L(p_j), \\
\emptyset & \text{otherwise.}
\end{cases}
$$

(1.40)

In the equation above prefix$_L(p_i)$ is the sub-pattern at the beginning of $p_i$ with exactly $(L - 1)$ characters. Similarly, suffix$_L(p_i)$ is the sub-pattern at the end of $p_i$ with exactly $(L - 1)$ characters. For example:

$$
\text{prefix}_3(\text{W.K}) = \text{W.Q} \\
\text{suffix}_3(\text{W.K}) = \text{QK} \\
\text{prefix}_3(\text{QKR}) = \text{QK} \\
\text{suffix}_3(\text{QKR}) = \text{KR}.
$$

(1.41)
>seq 0
KD沃QKRK
>seq 1
C沃QKRK
>seq 2
沃QKRKNP

Teiresias
L/W/K = 3/4/3

<table>
<thead>
<tr>
<th>motifs →</th>
<th>W.QK</th>
<th>QKR</th>
<th>QK.K</th>
<th>KRK</th>
<th>Q.RK</th>
</tr>
</thead>
<tbody>
<tr>
<td>offset #0</td>
<td>KD沃QKRK</td>
<td>KD沃QKRK</td>
<td>KD沃QKRK</td>
<td>KD沃QKRK</td>
<td>KD沃QKRK</td>
</tr>
<tr>
<td></td>
<td>(0,2)</td>
<td>(0,4)</td>
<td>(0,4)</td>
<td>(0,5)</td>
<td>(0,4)</td>
</tr>
<tr>
<td>offset #1</td>
<td>C沃QKRK</td>
<td>C沃QKRK</td>
<td>C沃QKRK</td>
<td>C沃QKRK</td>
<td>C沃QKRK</td>
</tr>
<tr>
<td></td>
<td>(1,1)</td>
<td>(1,3)</td>
<td>(1,3)</td>
<td>(1,4)</td>
<td>(1,3)</td>
</tr>
<tr>
<td>offset #2</td>
<td>沃QKRKNP</td>
<td>沃QKRKNP</td>
<td>沃QKRKNP</td>
<td>沃QKRKNP</td>
<td>沃QKRKNP</td>
</tr>
<tr>
<td></td>
<td>(2,0)</td>
<td>(2,2)</td>
<td>(2,2)</td>
<td>(2,3)</td>
<td>(2,2)</td>
</tr>
</tbody>
</table>

**Figure 1-12**: Scanning phase of Teiresias. During the scanning phase, Teiresias enumerates all elementary motifs with exactly L characters and at most W − L wild-cards. Using the input sequences above, Teiresias finds five such elementary motifs as shown in the table: F.AS, AST, AS.S, STS, and A.TS. The offset list for each of these is shown in the table. In the next phase of the algorithm, these elementary motifs are convolved together to form the final, maximal motifs.
To illustrate this, consider the following examples:

\[
\begin{align*}
DF.A.T \oplus A.TSE &= DF.A.TE \\
L.XF.A.MM \oplus A.MSE &= L.XF.A.MME \\
WX.N.N \oplus N.PSE &= \varnothing.
\end{align*}
\]

If two motifs can be convolved — i.e. \( p_i \oplus p_j \neq \varnothing \) — then the offsets of the new, longer regular expression, \( p_k \) are given by

\[
\mathcal{L}(p_k) = \{(x, y) \in \mathcal{L}(p_i) \mid \exists (x, z) \in \mathcal{L}(p_j) \text{ such that } z - y = \mathcal{W}(p) - \mathcal{W}(%suffix\mathcal{L}(p))\}.
\]

(1.42)

If \(|\mathcal{L}(p_k)| < K\) then the motif does not have sufficient support and is discarded. Conversely, if \(|\mathcal{L}(p_k)| = |\mathcal{L}(p_i)|\) then \( p_i \) is not a maximal motif. Or if \(|\mathcal{L}(p_k)| = |\mathcal{L}(p_j)|\) then \( p_j \) is not a maximal motif. But, if

\[
|p_i \oplus p_j| < K \forall j,
\]

(1.43)

then \( p_i \) is maximal.

Obviously, by convolving each elementary motif with every other elementary motif, i.e. by repeating \( p_k = p_i \oplus p_j \) for all \( i \) and \( j \), the maximal motifs can be discovered. Teiresias uses an intelligent method of sorting the elementary motifs that does not require doing the all–by–all comparison and yet still guarantees that all the maximal motifs are discovered. The set of these maximal motifs are then returned to the user as in Figure 1-13 on the next page.

The broad applicability of Teiresias has been shown in numerous studies. In particular, the algorithm has been very successful in multiple sequence alignment [189], motif dictionary building [208], and gene finding in microbial genomes. In the work here, we will expand upon these studies in our application of the Teiresias motif discovery engine to practical problems of interest to the biology community, cf. Chapter 2.

**Gibbs sampler and other position weight matrix–based tools**

As described in Section 1.4 on page 46, PWMs can be much more specific than regular expressions for modeling a set of motif instances. But, this motif model also presents some unique
Figure 1-13: Pattern discovery with Teiresias. Here we have three protein sequences and we use Teiresias to find all patterns involving at least $L = 5$ non–wildcard characters that occur at least $K = 2$ times and have a fraction of non–wildcard characters of at least $L/W = 5/8$. These are called $5/8/2$ patterns and there are three such patterns in this set of sequences. Along with each motif is an offset list $\mathcal{L}(p_i)$ that specifies each sequence–position combination where the motif occurs. For motif $p_j =$“A.D.HGV” the associated offset list is $\mathcal{L}(p_j) = \{(1,46),(2,17)\}$, indicating that this motif occurs twice: once in sequence 1 at position 46 and once in sequence 2 at position 17.
difficulties. Recall that, as described in Section 1.5 on page 57, most regular expression–based motif discovery tools are “pattern driven” in the sense that, at some level, they rely on enumerating possible regular expressions and then determining which of those has a significant support within a given set of sequences. A similar approach does not work for PWMs because the set of production probabilities (see Equation 1.28 on page 49), or equivalently the target frequencies in the \( f \) matrix (see Equation 1.29 on page 50), are sampled from a continuous distribution. Therefore, the set of possible PWMs cannot be enumerated \textit{a priori} because they are effectively infinite.

Most motif discovery tools that use PWM models skirt this issue by taking a more focused approach. Instead of returning a large set of motifs, as is common for regular expression–based tools such as Teiresias, PWM–based tools usually return either one or a small set of motifs. Table 1.5 on the facing page shows a list of publications introducing motif discovery tools that use PWMs. Most of these tools use a procedure whereby they are initialized with a random PWM and progressively optimize the PWM to maximize its sensitivity and specificity for the input sequences. However, some of the algorithms, such as Mitra–PSSM, which was proposed by Eskin [77], work in a much different fashion, somewhat similar to some of the regular expression–based tools described in the previous section.

Here, I will describe the algorithm by Lawrence et al. [147], which is generally referred to as the Gibbs sampler. This algorithm is the basis for many of the other algorithms shown in Table 1.5. As such, it is somewhat indicative of the class has a whole. The Gibbs sampler is a Markov chain Monte Carlo, or MCMC method [155, 170]. The Monte Carlo aspect of the method refers to optimization routine by which the PWM is successively refined. This routine is a Markov chain in the sense that the new, refined PWM depends only on the previous, unrefined PWM.

The Gibbs sampler is shown schematically in Figure 1-14 on page 69. The input to the algorithm is a set of sequences \( S = \{s_0, s_1, \ldots, s_n\} \) and an integer \( W(p) \), which is the width of the motif \( p \) that we are trying to “discover.” (Obviously, \( p \) is assumed to be represented by a PWM.) The Gibbs sampler assumes that the motif occurs exactly once in each sequence in \( S \); however, more recent alterations of this basic framework allow for multiple instances in a single sequence or for sequences to be missing an instance. Here, I described the most simple
Table 1.5: Motif discovery tools using position weight matrices or similar models. As discussed in the text, PWMs are more specific than regular expressions; however, in general, there are fewer algorithms utilizing this motif model. Most of the later tools shown in the table are geared towards finding binding sites for regulatory proteins upstream of sets of co–regulated genes. Of these publications, the seminal manuscript is that by Lawrence et al. [147].

<table>
<thead>
<tr>
<th>Authors</th>
<th>Year</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stormo and Hartzell</td>
<td>1989</td>
<td>[238]</td>
</tr>
<tr>
<td>Lawrence et al.</td>
<td>1993</td>
<td>[147]</td>
</tr>
<tr>
<td>Liu</td>
<td>1994</td>
<td>[154]</td>
</tr>
<tr>
<td>Bailey and Elkan</td>
<td>1994</td>
<td>[19]</td>
</tr>
<tr>
<td>Leung et al.</td>
<td>1996</td>
<td>[149]</td>
</tr>
<tr>
<td>Goffeau</td>
<td>1998</td>
<td>[99]</td>
</tr>
<tr>
<td>Hertz and Stormo</td>
<td>1999</td>
<td>[115]</td>
</tr>
<tr>
<td>Workman and Stormo</td>
<td>2000</td>
<td>[273]</td>
</tr>
<tr>
<td>Hughes et al.</td>
<td>2000</td>
<td>[124]</td>
</tr>
<tr>
<td>GuhaThakurta and Stormo</td>
<td>2001</td>
<td>[101]</td>
</tr>
<tr>
<td>Bi and Rogan</td>
<td>2004</td>
<td>[35]</td>
</tr>
<tr>
<td>Raphael et al.</td>
<td>2004</td>
<td>[204]</td>
</tr>
<tr>
<td>Eskin</td>
<td>2004</td>
<td>[77]</td>
</tr>
<tr>
<td>Siddharthan et al.</td>
<td>2005</td>
<td>[229]</td>
</tr>
<tr>
<td>Liu et al.</td>
<td>2005</td>
<td>[156]</td>
</tr>
<tr>
<td>Leung and Chin</td>
<td>2005</td>
<td>[148]</td>
</tr>
<tr>
<td>Zhong et al.</td>
<td>2005</td>
<td>[282]</td>
</tr>
<tr>
<td>Tharakaraman et al.</td>
<td>2005</td>
<td>[243]</td>
</tr>
<tr>
<td>Down and Hubbard</td>
<td>2005</td>
<td>[70]</td>
</tr>
<tr>
<td>Macisaac et al.</td>
<td>2006</td>
<td>[159]</td>
</tr>
</tbody>
</table>
case based on the original manuscript by Lawrence et al.

As shown in Figure 1.14, the Gibbs sampler has five major steps.

1. Choose random starting locations for the motive in all but one of the given sequences.
2. Use these sites to compute a PWM.
3. Score the sequence that was left out in step 1 over its entire length.
4. Choose a site within the sequence probabilistically, based on the scores of each possible site, i.e. choose sites that have higher scores with higher probability.
5. Recompute the PWM using the site selected in step 4 and leaving out a different, randomly selected sequence. Then, go to step 1 and repeat until the PWM no longer changes significantly.

Most of the other PWM–based motif discovery tools listed in Table 1.5 use an approach that is similar to the Gibbs sampler. In general, these tools excel at finding motifs in DNA sequences such as cis–regulatory binding sites. (See Tompa et al. [249] for an excellent review of this problem and a demonstration of the power of PWM–based tools.) Other motif discovery tools use different optimization procedures than the Gibbs sampler, which are slight variations on the MCMC method, such as simulated annealing [142] or expectation maximization. Most of these refinement procedures guarantee that the algorithm will converge to a maximum [92]; however, it is not guaranteed that a maximum is globally optimal. The optimization can become trapped in a local optimum, which is called “slow–mixing” of the Markov chain. New procedures that avoid this are an active area of study [155].
Figure 1-14: Schematic of the Gibbs sampling algorithm. As the figure shows, the Gibbs sampling method is an iterative algorithm that progressively refines a position weight matrix starting from a random PWM. If the input sequences contain a very strong motif, Gibbs sampling tends to converge very quickly upon it. However, in its original manifestation [147], the method was not able to find motifs that either occurred multiple times in a single sequence, or were found in some sequences but not others. In general, most motif discovery tools using PWMs bear a great deal of similarity to the original Gibbs sampling method.
Chapter 2

Design of antimicrobial peptides

2.1 Introduction

In the previous chapter, I introduced grammars as a generalized method for modeling motifs in sequences of characters. In addition, I presented a detailed look at the Teiresias motif discovery tool. In this, the second chapter of my thesis, I show how Teiresias can be used to derive sets of regular grammars describing a particular class of protein sequences — antimicrobial peptides. In what follows, I present a general background on antimicrobial peptides and then provide a rationale for why these peptides are particularly well-suited for being modeled using regular grammars. I detail the construction of an annotation tool for finding new antimicrobial peptides and validating the general hypothesis that regular grammars can be used as a sensitive and specific indicator of antimicrobial function in peptide sequences. Next, I describe the preliminary design of synthetic antimicrobial peptides using an evolutionary approach, which, although ultimately inconclusive, provided motivation for a more focused design. The final section of this chapter describes a more focused design approach, detailing the successful construction of numerous novel peptides with strong antimicrobial activity against a wide spectrum of bacteria.

The research described in this chapter is drawn largely from a publication that is in preparation in collaboration with Christopher Loose, Isidore Rigoutsos, and Gregory Stephanopoulos. (Some experimental work in Section 2.4 on page 88 was also performed by Gyoo Yeol Jung.) Throughout this chapter, the use of the pronoun “we” refers to this group of authors.
2.2 Motivation

Antimicrobial peptides are small proteins that attack and kill microbes. These peptides are effectors of the innate immune system: the phylogenetically ancient first line of defense against pathogen assault [141, 213]. Antimicrobial peptides are ubiquitous amongst multicellular eukaryotes and found in diverse contexts including frog skin [230], scorpion venom [172], and human sweat [219].

There is a growing interest in antimicrobial peptides, due largely to the proliferation of multi–drug resistant pathogen strains [50]. These strains are resistant to one or more common antibiotics such as penicillin, tetracylin, or vancomycin. In the United States alone, the cost of treating and preventing infections by these pathogens is estimated to be many billions of dollars annually [184]. In the arms race against microbes, mankind is losing — only a single new class of antibiotics was developed in the past 30 years [182, 260]. However, there is mounting evidence that antimicrobial peptides are less likely to induce bacterial resistance and will make a strong contribution to our therapeutic arsenal [91, 279, 280].

Human antimicrobial peptides, such as the defensins and cathelicidins, help to maintain a passive defense against pathogens in the environment. A malfunction of these peptides leads to severely immunocompromised phenotypes. For example, a deficiency of the LL-37 cathelicidin leads to morbus Kostmann, a congenital neutropenia characterized by recurrent bacterial infection and short life–expectancy [40, 201]. In addition, the pathogenesis of cystic fibrosis (CF) is indirectly caused by antimicrobial peptide impairment [89]. CF patients have a defective Cl– ion channel in the pulmonary airway epithelia that causes unusually high salt concentrations. The salt disrupts the function of the epithelial defensins, leading to chronic infections and ensuing respiratory failure [234, 280]. More severe phenotypes have been produced in loss–of–function animal models. For example, Wilson et. al. [268] showed that mice with depressed defensin activity required a 10–fold lower dose of the S. typhimurium pathogen to produce a fatality. In contrast, gain–of–function mice expressing human enteric defensin HBD–5 have a markedly increased resistance to S. typhimurium assault [218].

In addition to their more publicized antibiotic capabilities, antimicrobial peptides appear to be important in a variety of other diseases. For example, the antimicrobial peptides of Anophe-
les gambiæ, the malaria mosquito, are upregulated after malaria (Plasmodium berghei) infection [58] and, in some cases, are capable of killing the ookinets of the parasite [24, 257]. Antimicrobial peptides are also indicated in a resistance to the AIDS–causing virus, HIV. Long–term HIV nonprogressors display elevated levels of α–defensins that inhibit the proliferation of the virus [281]. Finally, a limited class of antimicrobial peptides may form the basis for novel cancer treatments [74, 140]. For example, the antimicrobial peptide tachyplesin can repress the growth of cancerous tumors both in vitro and in vivo [52].

The many disease–relevant behaviors of antimicrobial peptides are a result of their ability to broadly distinguish eukaryotic cells from pathogenic invaders. There are two features that give the peptides this ability: a net positive charge and an amphipathic 3–D structure [75, 94]. These features endow the peptide with an affinity for negatively charged outer leaflet of the bacterial cytoplasmic membrane (see Figure 2–1 on the next page). This affinity leads to permeabilization of the bacterial membrane, which is the basis for the bactericidal activity of antimicrobial peptides. Although this mode of action is common to almost all antimicrobial peptides, there are many diverse primary sequences that can produce this behavior. These sequences form a handful of conserved families, the most common of which are the α–helical and β–sheet antimicrobial peptides [251].

Figure 2–2 on page 75 shows the structure of aurein–1.2, an archetypal alpha helical antimicrobial peptide from the Australian Southern bell frog [261]. Alpha helical AmPs form the largest single family of AmPs. They are particularly common in amphibians because species such as frogs tend to inhabit wet and warm ecological niches that are conducive to the proliferation of bacteria. (See Figure 2–3 for a phylogenetic tree of the more than 400 amphibian AmPs.) Alpha helical AmPs tend, in general, to have a amphipathic structure in which positively charged residues are segregated on to a particular side of the longitudinal axis of the helix. Negatively charged or neutral residues tend to be isolated on the opposite side from the positively charged residues. Evidence suggests that this confirmation allows the peptide to position itself judiciously relative to the bacterial membrane, facilitating entry of the peptide into the membrane and, ultimately, membrane disruption [280].

The characteristic membrane–attack of antimicrobial peptides is the primary rationalization of the peptides' propensity to not induce bacterial resistance to the same degree as small
Antimicrobial peptide action. In the figure above, amphipathic antimicrobial peptides with net positive charges are attracted via electrostatic forces to the negatively charged outer–leaflet of the microbial membrane (step 1) [280]. This membrane is either the lipopolysaccharide layer or the peptidoglycan layer of gram–negative and gram–positive bacteria, respectively [75]. In addition, the β–1,3–glucan in fungal membranes and the phosphoglycan of certain parasites can give membrane characteristics that are exploited by certain classes of antimicrobial peptides. The peptides cover and lyse the membrane via either a “barrel stave” or “carpet” mechanism (step 2) [226]. Although some antimicrobial peptides are hemolytic, in general, they are not damaging to multicellular organisms because 1) the negatively charged phosphatidylinerines of their outer leaflet are sequestered on the cytoplasmic side of the membrane, and 2) the membranes are stabilized by cholesterols [280].
2.2. MOTIVATION

A) Ball and stick

B) Ribbon

C) Peptide wheel view

Figure 2-2: The structure of aurein–1.2 [261]. Aureins constitute a large family of secreted proteins originally isolated from the skin of frogs. This particular structure was isolated from the Australian Southern bell frog. The peptide conforms to the classic amphipathic alpha helical structure and has wide-spectrum antimicrobial activity. Part A) shows a ball–and–stick representation of the structure in which nitrogen atoms are colored blue and oxygen atoms are colored red. Part B) shows the same structure using a cartoon representation that clearly shows the alpha helix. Finally, part C) shows a helical wheel projection in which uncharged residues are boxed in order to highlight the segregation of charges on the helix. Graphics created using PyMol (DeLano Scientific, San Carlos, CA, USA).
related, including the anurans, bombyliid, bombolus, and brevipes. Figure 2.4 shows the diversity of amphibians and reveals that 400 amphi's from amphibian sources are high-sourced. However, many amphibians have many families of amphi's, some of which are only distantly related. The figure shows the diversity of sequence similarity and variation within each species. This figure shows the diversity of sequence similarity for over 400 amphibians and amphi's. Figure 2.4: A phylogenetic tree of amphibian amphi's because they tend to inhabit environments that are conducive to the growth of bacteria.
molecule pharmaceuticals. That is, because the peptides leverage a pervasive polygenic trait of bacteria, the structure of the cell wall, it is “expensive” for the bacteria to evolve a resistance \[279, 280\]. For this reason, many companies are developing therapeutics based on antimicrobial peptides, many of which are in phase III FDA trials \[102\]. Even more encouraging, some peptides show strong \textit{in vitro} bactericidal activity against pathogen strains that have developed a resistance to multiple conventional antibiotics \[91, 239, 246\].

\section*{2.3 A grammatical approach to annotating AmPs}

Our preliminary studies of natural AmPs indicated that their amphipathic structure gives rise to a modularity among the different AmP amino acid sequences. The repeated usage of sequence modules — which may be a relic of evolutionary divergence and radiation — is reminiscent of phrases in a natural language, such as English. For example, the grammar 
\[Q.EAG.L.K..K\] (the “..” is a “wildcard”, which indicates that any amino acid will suffice at that position in the grammar) is present in over 90% of cecropins, an AmP common in insects. Based on this observation we modeled the AmP sequences as a formal language — a set of sentences using characters from a fixed alphabet, in this case the alphabet of amino acid one-letter symbols \[134\].

We conjectured that the “language of AmPs” could be described by a set of regular grammars and that these grammars, in turn, could be used to annotate and design novel AmPs. As discussed in Chapter 1, regular grammars are, in essence, simple rules for describing the allowed arrangements of characters. These grammars, such as the cecropin grammar mentioned previously, are commonly written as regular expressions and are widely used to describe patterns in nucleotide and amino acid sequences \[118, 225\].

To find a set of grammars describing AmPs we used the Teiresias pattern discovery tool \[207\] (see Section 1.5 on page 57 to discover an exhaustive, maximal set of regular grammars in a collection of antimicrobial peptides assembled from a variety of sources.

\subsection*{2.3.1 Collecting a database of antimicrobial peptides}

Our collection of known antimicrobial peptides was taken principally from two databases: the Antimicrobial Sequences Database (AMSDb) \[250\] and SwissProt/TrEMBL \[22\]. The
AMSDb contains about 750 antimicrobial peptides, all of which are a subset of SwissProt/TrEMBL. Some of the entries in the AMSDb are sequence fragments that are derived from larger precursors via post-translational modification. We discarded these peptides unless the reported antimicrobial fragment comprised at least 80% of the length of its parent sequence. From the remaining entries, we selected all that were from eukaryotic organisms, including the complete length of the parent peptides in our database.
### Table 2.1: Common antimicrobial peptide families

<table>
<thead>
<tr>
<th>acaloleptin</th>
<th>achacin</th>
<th>adenoregulin</th>
<th>alpha–defensin</th>
</tr>
</thead>
<tbody>
<tr>
<td>androctonin</td>
<td>andropin</td>
<td>apidaecin</td>
<td>attacin</td>
</tr>
<tr>
<td>aurein</td>
<td>azurocidin</td>
<td>bactenecin</td>
<td>bactericidin</td>
</tr>
<tr>
<td>bactinecin</td>
<td>beta–defensin</td>
<td>bombinin</td>
<td>bombolitin</td>
</tr>
<tr>
<td>bactinecin</td>
<td>bactinecin</td>
<td>caerin</td>
<td>caltrin</td>
</tr>
<tr>
<td>cathelin</td>
<td>cecropin</td>
<td>ceratotoxin</td>
<td>citropin</td>
</tr>
<tr>
<td>clavanin</td>
<td>coleoptericin</td>
<td>corticostatin</td>
<td>crabrolin</td>
</tr>
<tr>
<td>defensin</td>
<td>demidefensin</td>
<td>dermaseptin</td>
<td>dermcidin</td>
</tr>
<tr>
<td>dipterica</td>
<td>drosocin</td>
<td>drosomycin</td>
<td>enbocin</td>
</tr>
<tr>
<td>formaecin</td>
<td>gaegurin</td>
<td>gallinacin</td>
<td>glovein</td>
</tr>
<tr>
<td>granulysin</td>
<td>hadrurin</td>
<td>hemiopicin</td>
<td>hemiptericin</td>
</tr>
<tr>
<td>hemolin</td>
<td>hepcidin</td>
<td>histatin</td>
<td>holotricin</td>
</tr>
<tr>
<td>hymenoptaecein</td>
<td>hyphancin</td>
<td>indolicidin</td>
<td>lebocin</td>
</tr>
<tr>
<td>macin</td>
<td>maculatin</td>
<td>maximin</td>
<td>metalnikowin</td>
</tr>
<tr>
<td>metchnikowin</td>
<td>misgurin</td>
<td>moricin</td>
<td>myticin</td>
</tr>
<tr>
<td>mytilin</td>
<td>mytimycin</td>
<td>nk–lysin</td>
<td>penaeidin</td>
</tr>
<tr>
<td>permatin</td>
<td>phormicin</td>
<td>phylloxin</td>
<td>pleurocidin</td>
</tr>
<tr>
<td>polyphemusin</td>
<td>ponericin</td>
<td>protegrin</td>
<td>pseudin</td>
</tr>
<tr>
<td>pyrrhocoricin</td>
<td>ranalexin</td>
<td>ranatuerin</td>
<td>rhinocerosin</td>
</tr>
<tr>
<td>royalisin</td>
<td>rugosin</td>
<td>salmocidin</td>
<td>sapecin</td>
</tr>
<tr>
<td>sarcotoxin</td>
<td>sillucin</td>
<td>spingerin</td>
<td>styelin</td>
</tr>
<tr>
<td>tachycitin</td>
<td>tachyplecin</td>
<td>temporin</td>
<td>tenecin</td>
</tr>
<tr>
<td>termicin</td>
<td>thanatin</td>
<td>tricholongin</td>
<td>zeamatin</td>
</tr>
</tbody>
</table>
SwissProt/TrEMBL is a database of about 120 thousand heavily annotated sequences. Included in the annotations are keywords grouping proteins into functional categories. For our initial database of antimicrobial peptides we extracted all the eukaryotic sequences matching the keywords “antibiotic”, “fungicidal”, or “defensin”. These sequences were added to the peptides we collected from the AMSDb.

Using the sequences that we extracted from AMSDb and SwissProt/TrEMBL, we made a list of common antimicrobial peptide names — such as “defensin” or “tenesin” — and collected sequences from SwissProt/TrEMBL matching these names. From the name–matched sequences, we manually selected those eukaryotic sequences that had literature evidence of antimicrobial activity but were not explicitly labeled as such in SwissProt/TrEMBL. These sequences, together with the sequences from AMSDb and the first set from SwissProt/TrEMBL, formed our initial database of antimicrobial peptides. In the following section, we describe how these sequences were used, via a homology–based bootstrapping method, to find even more antimicrobial peptides within SwissProt/TrEMBL.

2.3.2 Finding more antimicrobial peptides

A minority of the antimicrobial peptides within SwissProt/TrEMBL, were not found using either the keyword or “common–name” searches. To find these sequences we used two approaches in parallel. First, we used a FastA sequence alignment based approach. Second, we used a grammar matching–based approach with TEIRESIAS. Both of these approaches are detailed below and summarized in Figure 2-4.

Seeding the peptide database through similarity searching

Starting with our initial database of sequences (S₀ in Figure 2-4) from SwissProt/TrEMBL and AMSDb, we aligned each sequence in S₀ against the entire SwissProt/TrEMBL database. If a sequence in SwissProt/TrEMBL aligned with a sequence from S₀ with 80% or greater pairwise identity over the length of both sequences, the new sequence was marked as a possible antimicrobial peptide. Of the marked sequences, we selected those that were from eukaryotic organisms and had literature evidence of antimicrobial activity. These sequences are shown as S₁ in Figure 2-4, meaning sequences found using FastA in the first iteration.
From $S_0$, use TEIRESIAS to generate the two pattern sets $C_i^{8/8/2}$ and $C_0^{6/15/2}$.

- Combine and process the two pattern sets to make the $\phi'$ set.

- Use $\phi'$ and $S_i$ to find new homologous sequences in SwissProt/TrEMBL: $S_{i+1}$

**Figure 2-4:** A schematic of the bootstrapping method used to collect antimicrobial sequences from SwissProt/TrEMBL. On the left, using TEIRESIAS, we computed an $8/8/2$ grammar set ($C_i^{8/8/2}$) from the initial set of sequences, $S_0$. These grammars were masked from $S_0$ to make $S_0 \mid_{m-8/8/2}$, from which the $6/15/2$ grammar set, $C_0^{6/15/2}$, was found using TEIRESIAS again. The two grammar sets were combined and processed (see Appendix) to make $\phi'$. This final grammar set was used to find more antimicrobial sequences in SwissProt/TrEMBL. On the right side of the schematic, sequences from $S_0$ were aligned against SwissProt/TrEMBL to find new antimicrobial sequences.
Seeding the peptide database through use of grammar discovery

In the second stage of our bootstrapping method, we used TEIRESIAS to find grammars that could be used to search for antimicrobial sequences in SwissProt/TrEMBL. As shown in Figure 2-4, from the $S_0$ sequence set, we derived two separate grammar sets ($C_0^{8/2/2}$ and $C_0|^{6/15/2}_{m-8/8/2}$), which we combined together. This combined set, $\phi$, was processed (a detailed description of this processing is in the appendix) to increase the selectivity and sensitivity of the grammars for antimicrobial peptide sequences. Finally, in each sequence from SwissProt/TrEMBL, we searched for instances of grammars from $\phi'$. If 80% of the amino acids in a peptide from SwissProt/TrEMBL were contained within instances of grammars from $\phi'$ the peptide was marked. Of the marked sequences, we selected those sequences that were from eukaryotic organisms and had literature evidence of antimicrobial activity, calling these sequences $S_T^T$.

Iterating the bootstrapping method

The new antimicrobial peptides found using FastA and TEIRESIAS, $S_0^C$ and $S_0^T$ respectively, were added to the initial database, $S_0$, to make $S_1$. Next, a bootstrapping method was repeated on the $S_1$ sequence set to make larger and larger sets ($S_2$, $S_3$, ...) until no more antimicrobial peptides in SwissProt/TrEMBL could be found. This process is shown in Figure 2-4 and detailed below.

1. Finding Highly Conserved grammars

First we found all the highly conserved ($8/8/2$) grammars in $S_0$. These grammars are substrings in $S_0$ that are repeated exactly, that is, grammars without any wild-cards or bracketed expressions. Let these grammars be called $C_0^{8/2/2}$, meaning $8/8/2$ grammars from the first iteration. In order to simplify the grammar discovery process for the next step, the sequence set $S_0$ was masked with the $C_0|^{8/2/2}_{m-8/8/2}$ grammars to make the sequence set $S_0^T$.

2. Finding Loosely Conserved grammars

“Masking” is described in detail in Rigoutsos and others [208]. In brief, by masking a grammar, we tag each instance of a grammar except for the instance in the longest sequence in which the grammar is found. Tagged regions are then excluded from further grammar discovery processes.
Using the $S_o \mid_{m=8/15/2}$ sequences, we found all $8/15/2$ grammars, which we will call $C_o \mid_{m=8/15/2}$. These grammars are more loosely conserved than the $C_o \mid_{m=8/2/2}$ grammars and are typically greater in number.

3. Post–Processing the grammars

Let the union of the two grammar sets computed above be $\phi_o = C_o \mid_{m=8/2/2} \cup C_o \mid_{m=8/15/2}$. We would like to match grammars in $\phi_o$ against SwissProt/TrEMBL to find any remaining unknown antimicrobial sequences. But, to gain greater specificity and sensitivity, we first processed the grammars in $\phi_o$ to make a grammar set $\phi'$. 

(a) For every grammar in $\phi_o$, we de–referenced each wild–card character that could be expressed as a bracketed expression with no greater than four characters. That is, in the grammar “K.T”, the “.” might be replaced with “[AG]” if, for each instance of “K.T”, only “A” and “G” are found in the wild–card position. If more than four characters were needed in the bracketed expression, we left the wild–card character instead.

(b) For each of the altered grammars in $\phi_o$ we decomposed the grammar into a set of smaller, redundant grammars by using a sliding window of ten non–wild–card characters. So, a grammar such as “[FWY]FK.[GQ][KRQ]CPDAY” would be decomposed into three distinct grammars: “S[RKM][FWY]FK.[GQ][KRQ]CPD”, “[RKM][FWY]FK.[GQ][KRQ]CPDA”, and “S[RKM][FWY]FK.[GQ][KRQ]CPDAY”, each ten non–wild–card amino acids in length.

(c) From this new, redundant $\phi_o$ we kept only those grammars that were statistically significant. These are grammars that have a log–odds probability less than or equal to $-30$.

Let these the processed $\phi_o$ grammar set be called $\phi'_o$.

The final sequence set, from the last iteration, became our database of known antimicrobial peptide sequences and the final $\phi'$ became our antimicrobial grammar database.
CHAPTER 2. DESIGN OF ANTIMICROBIAL PEPTIDES

Figure 2-5: A plot of the progress of the bootstrapping method. The figure shows that our antimicrobial sequence database grew from 836 sequences to 931 sequences in 3 iterations.

2.3.3 Antimicrobial sequence and grammar databases

The initial database of antimicrobial peptides collected from AMSDb and SwissProt/TrEMBL contained a total of 836 sequences. Starting with these sequences, the bootstrapping method described previously went through 3 iterations until no more sequences were found in SwissProt/TrEMBL. The last sequence set, S₃, which contained a total of 931 sequences, was used as our antimicrobial sequence database and is available on-line at http://cbsrv.watson.ibm.com/Tspd.html. The final grammar set (φ' from the last bootstrapping iteration) contained a total of 241,642 grammars covering the sequence space of the final sequence database.

2.3.4 Annotator design and validation

Together, these ~200K grammars describe the “language” of the AmP sequences. In this linguistic metaphor, the peptide sequences are analogous to sentences and the individual amino acids are analogous to the words in a sentence. Each grammar describes a common arrangement of amino acids, similar to popular phrases in English.

Given an arbitrary sequence of amino acids, it is possible that some parts of the sequence are “matched” by one or more of the grammars in our database. For example, the white mustard plant AmP Afp1 (Genbank accession no. P30231) contains the amino acid sequence fragment CICYFPC, which matches the grammar CICY [FVK]PC from our database. (As discussed in Section 1.4 on page 42, the bracketed expression [FVK] indicates that, at the fifth position in the grammar, either phenylalanine, valine, or lysine is equally acceptable.) Based on this match, we would say that the Afp1 fragment is “grammatical.”
A GRAMMATICAL APPROACH TO ANNOTATING AMPS

Figure 2-6: A schematic of our grammar–based alignment method. In the figure, a user–supplied input sequence is searched for instances of grammars \( p_A, p_B, p_C \ldots \) that occur in the set of known antimicrobial sequences \( S \). Grammars that occur in both the input sequence and sequences in \( S \) are then used to create alignments. As indicated in the figure, the input sequence shows homology to \( s_1 \) in two distinct regions, so both possible alignments are shown. See Figure 2-7 on the following page for an example of how these grammar–based alignments appear in practice.

Using the antimicrobial grammar database, we created an on–line tool for annotating antimicrobial peptides by determining the degree to which a query sequence is grammatical. (This tool is available online at \( \text{http://cbcsrv.watson.ibm.com/Tspd.html} \).) A user–supplied input sequence is annotated by generating grammar–based alignments of the input against sequences in our database of known antimicrobial sequences \( S \). This alignment takes place in two distinct steps. First, we search the input sequence for instances of grammars from the antimicrobial grammar database (the final \( \phi' \)). Second, for each contiguous stretch of shared grammars between the input sequence and a sequence from \( S \), an alignment is produced. Figure 2-6 show a schematic of the alignment process.
Figure 2-7: Example results of the grammar-based alignment method. The figure shows the annotation of a query sequence after it has been compared to a reference sequence.
QUERY
P
P
P
P
P
P
P
P
P
P
P
P
P
P
P
P
P
P
P
P
P
P
P
P
Q

LQAQAEPLQA
LVLLAFQVQA
LVLPSASAQA
LVLPSASAQA
LVLLSFQVQA
LVLLAFQVQA
LVLLAFQVQA
LVLLAFQVQA
LVVPSASAQA
LVVPSASAQA
LVVPSASAQA
LVVPSASAQA
LVVPSASAQA
LVLLAFQVQA
LVLLAFQVQA
LVLLAFQVQA
LVLLAFQVQA
LVLLAFQVQA
LVLLAFQVQA
LVLLAFQVQA
LVLLAFQVQA
LVVPSASAQA
LHAQAEARQA
LHAQAEARQA
LQAQAEPLQA
LQAKAEPLQA
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.
.

Cryptdin-related protein C-
Bactenecin  precursor (BAC)
Bactenecin  precursor (BAC)
Cryptdin- precursor
Cryptdin- precursor
Cryptdin- precursor
Cryptdin- precursor
Protegrin  precursor (PG-)
Indolicidin precursor
Antibacterial peptide PMAP-
Antibacterial peptide PMAP-
Antibacterial peptide PMAP-
Cryptdin-/ precursor
Cryptdin- precursor
Cryptdin- precursor
Cryptdin- precursor (Fragmen
Cryptdin- precursor
Cryptdin- precursor (Fragmen
Cryptdin- precursor
Cryptdin- precursor
Prophenin- precursor (PF-) (
Theta defensin-, subunit A pr
Theta defensin-, subunit B pr
Neutrophil defensins ,  and
Defensin  precursor (Defensin

Mus musculus (Mouse).
Bos taurus (Bovine).
Bos taurus (Bovine).
Mus musculus (Mouse).
Mus musculus (Mouse).
Mus musculus (Mouse).
Mus musculus (Mouse).
Sus scrofa (Pig).
Bos taurus (Bovine).
Sus scrofa (Pig).
Sus scrofa (Pig).
Sus scrofa (Pig).
Mus musculus (Mouse).
Mus musculus (Mouse).
Mus musculus (Mouse).
Mus musculus (Mouse).
Mus musculus (Mouse).
Mus musculus (Mouse).
Mus musculus (Mouse).
Mus musculus (Mouse).
Sus scrofa (Pig).
Macaca mulatta (Rhesus macaque).
Macaca mulatta (Rhesus macaque).
Macaca mulatta (Rhesus macaque).
Homo sapiens (Human).

Table .: Motif conservation for the query shown in Figure and the motif L[VQH][ALV][KLPQ][AS][EAF][APQS][ALRV]QA.
.. A GRAMMATICAL APPROACH TO ANNOTATING AMPS



Since it is possible that, for an arbitrary sequence, only a portion of the sequence is matched by one of our grammars, we developed a heuristic metric $Z$, which is the degree to which a query sequence is grammatical. To calculate $Z$, we assign a local score along the backbone of a query sequence that is equal to the number of grammars, or fractions of grammars with at least 10 amino acids, that have matches over the length of the query sequence. The total score for the sequence, $Z$, is the fraction of the sequence’s length that is covered by at least one grammar (see Figure 12-9). For example, a hypothetical sequence LFLTAIDRYIAAA — which is matched by LFLTAI [ID] [TR] [VY] I, but no other grammars in our database — would get a score of 10/13 since the first 10 positions in the sequence are covered by the match.

In order to annotate and design synthetic AmPs, we created a software tool to calculate the score $Z$ for a query sequence and to classify the sequence as either likely to have antimicrobial activity — if its $Z$-score is above a certain threshold — or not. To determine this threshold, we trained the tool on a subset of sequences from our AmP database as follows. We randomly selected 90% of the natural AmP sequences and generated a Teiresias grammar set, using the same Teiresias parameters that were used to generate our ∼200K grammar set. This smaller grammar set was used by our software to classify the remaining 10% of our AmP database, which was hidden among 10% of the non–AmP sequences from Swiss–Prot/TrEMBL (∼78K sequences). This experiment was repeated 300 times, with different random sets, to determine the best $Z$-score. We found that, at an $Z$-score threshold of 0.73, the software tool will correctly classify both the AmP and non–AmP sequences with 99.95% accuracy.

2.4 Preliminary strategy for the design of novel AmPs

2.4.1 Sequence design

As I showed in the previous section, the $Z$-score annotation metric is both sensitive and selective for existing AmPs. We hypothesized that this metric could be used equally well to design unnatural sequences that would have antimicrobial activity. In this section, I describe our preliminary strategy for designing these novel, unnatural sequences. Notably, the experimental data presented in this section were later discovered to not be reproducible due to experimental complications. See Section 2.4.3 on page 99. The data are presented here for the insight they lend to our more
focused, and successful, strategy for designing AmPs, which is described in Section 2.5 on page 99.

Based on the annotation results described in the previous section, we ran a computer simulation to create novel amino acid sequences with high $Z$–scores, but with minimal homology to natural AmPs. This simulation, shown schematically in Figure 2-8 on the next page, used the $Z$–score as a fitness function for the in silico directed evolution of these novel sequences. To begin, we created a randomized database of 100K progenitor sequences of uniform length with the same amino acid composition (i.e., the same percentage of each amino acid type) as our AmP database. Each of these sequences was allowed to have 4 mutated “children,” which were each 100 PAM (point accepted mutations) evolutionary units away from the parent. (The implied rates of mutation from the Blosum–50 matrix were used to make the mutations at the amino acid level.) These children, each of which differed from their parent sequence by at least one amino acid, were added to the total population of sequences. In order to avoid generating sequences that were similar to natural AmPs, the population was purged of any sequences that had 6 or more consecutive amino acids in common with any natural AmP sequence. Finally, the remaining sequences were scored using our annotation software. From the population, the sequences with the top 100 $Z$–scores were propagated to the following round, and the entire process was repeated.

Using the strategy described above, we allowed many populations of sequences to evolve, each with a different sequence length, which remained constant during the simulation. We stopped each simulation after 3,000 rounds of mutation and selection, by which point we found that populations of small sequence length would have converged to $S = 1$. For longer length populations, all the sequences typically reached at least $S = 0.73$ and all tended to be closely related to each other. We chose three sequences of lengths 20, 31, and 63 amino acids to test experimentally for antimicrobial activity: sequences synth–1, 2, and 3 in Table 2.3 on page 91.

Using NCBI Blast [11] (blastp) with the default parameters, we compared these sequences to the entire NCBI NR sequence database. The Blast results showed that none of the three sequences has significant homology to any known protein (E–value $\leq 10$), including the naturally–occurring AmPs. (More extensive similarity searching using PSI–Blast and E–value thresholds up to $\leq 50$ also failed to detect similarity to any natural AmPs.) This is possible because each
Figure 2-8: A schematic of the in silico directed evolution strategy. Position (1) shows the starting point: the database of 100K parental sequences. Each of these sequences has 4 mutated children (2) and the entire population is scored using the Z-score and our database of grammars from natural AmPs. From the scored population (3), the top 100 sequences are chosen and become the parental sequences for the next iteration. In addition to the directed evolution simulation, we considered other methods for generating sequences with high Z-scores. However, we chose this approach because it naturally allows for sequences of arbitrary length and the possibility that grammars may overlap in the designed amino acid sequences.
Table 2.3: The preliminary design synthetic antimicrobial peptides used in this study. For each synthetic AmP we also designed two sequences (“negative” a and b), which have the same amino acid composition as the synthetic peptide but have an S-score of zero. The table also shows statistics relevant to AmPs, which were calculated using the EMBOSS software package [206]. Note that synth–3 has only one negative version. Also, the peptides synth–1, 2, and 3 were the only peptides designed using our grammatical approach that were synthesized and tested experimentally.

<table>
<thead>
<tr>
<th>Peptide</th>
<th>S</th>
<th>Size</th>
<th>Charge</th>
<th>pI</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>synth–1:</td>
<td></td>
<td></td>
<td>4.5</td>
<td>1.92</td>
<td></td>
</tr>
<tr>
<td>synth–1</td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td>NKVKKPLTG AHRL LLPFTLFV</td>
</tr>
<tr>
<td>negative–1a</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>VVLKL LFFKNLPHKRTAG</td>
</tr>
<tr>
<td>negative–1b</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>LVLTFLFA TPKLNGRVKKFH</td>
</tr>
<tr>
<td>synth–2:</td>
<td>31</td>
<td>10.0</td>
<td>11.28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>synth–2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>MKKI KEAGKNLKLAPKEVAAKKS KSPTK</td>
</tr>
<tr>
<td>negative–2a</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>PAAGESKV KANKKKAKILPTMLKKEIKKS</td>
</tr>
<tr>
<td>negative–2b</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>SEALR KAKII AKMKV TKGAKNKPKLP EK</td>
</tr>
<tr>
<td>synth–3:</td>
<td>63</td>
<td>3.0</td>
<td>10.41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>synth–3</td>
<td>0.92</td>
<td></td>
<td></td>
<td></td>
<td>MKDKNS TGPLLS ALLLAVTAGGSPVAAAPWNPFAAILKAALQIAGAAEPEV TAKKKGPTKADA</td>
</tr>
<tr>
<td>negative–3a</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>GWAGLVA ETA IADKMSLKAAGEPNPQNDGAVLKTPKAAASAKPLGA AKTLAFISPVTPLLAK</td>
</tr>
</tbody>
</table>
grammar can be written in a large number of ways. For example, the 10–residue grammar 
\[ [LV] [GA]K[TN] [FL]AGHML \] occurs in 3 natural AmPs, but there are 16 possible 10–
residue sequences that match this grammar. Since our sequences are built from tiled grammars,
the synthetic sequences can quickly deviate from the naturally populated sequence space such
that it is impossible to detect similarity using sequence alignment tools (see Figure 2-9 on the
next page).

For each of the three synthetic peptides, we also designed a set of shuffled sequences, which
we hypothesized would have no antimicrobial activity. These “negative” peptides are shown
along with the three synthetic peptides in Table 2.3. The negative peptides have the same
amino acids as the synthetic sequences (and thus, the same molecular weight, charge, and pI);
however, the order of the amino acids was shuffled so that the negative sequences each have an
Z–score of zero.

### 2.4.2 Peptide synthesis and validation

Using an approach described elsewhere [168], we synthesized all 8 of the peptides shown in
Table 2.3. For each peptide we created a translation template consisting of three parts: green
fluorescent protein (GFP) with a T7 promoter, an enterokinase recognition site (ERS), and
the AmP to be tested. We synthesized the protein–product of each template in an E. coli–
derived in vitro translation system with continuous exchange [139]. The resulting peptides
were proteolytically cleaved with enterokinase and the yield of AmP in the translation mixture
was measured via GFP fluorescence using a 1:1 molar equivalence between the AmP and GFP
concentrations.

We characterized the antimicrobial activity of each synthetic AmP using a broth microdi-
lution assay described previously [12]. The top section of Table 2.4 shows the activities of the
synthetic peptides against four bacterial species: Bacillus cereus, Corynebacterium glutamicum,
E. coli, and Citrobacter rodentium. (See also Figure 2-10 on page 95.) These data suggested that
all three synthetic peptides had antimicrobial activity. Furthermore, none of the negative, “un–
grammatical” sequences had any activity. Thus, it appeared that the activity of the designed
peptides was not an artifact of molecular weight, charge, or pI. Instead, the activity appeared
correlated to the Z–score, suggesting that higher order sequence features are responsible for an-
Figure 2-9: The AmP design space. Part A shows the sequence space surrounding the set of natural AmPs. The “sequence space” is the combinatorially large set of all possible sequences. Even for a 20–residue peptide like synth–1 (see Table 2.3) this space is huge: $20^{20} \approx 10^{26}$ sequences. (For comparison, there are about $10^{22}$ stars in the known universe.) Our linguistic model focuses the search space to the “grammar space,” but allows a deviation from natural AmP sequences. This allows us to design peptides that show no significant homology to any naturally occurring sequences, but have the desired function. Part B shows a subsequence of the synth–2 peptide. Above and below the subsequence are grammars that match the sequence in a tiled arrangement. For each bracketed expression, any of the amino acids listed in the bracket will suffice.
timicrobial activity. (These data were later shown to be not reproducible. Later experiments showed that the sequences in Table 2.4 did not have detectable levels of antimicrobial activity under a more stringent assay. See Section 2.4.3 on page 99.)

The bottom of Table 2.4 shows the measured activities of synth–1 variants that were synthesized chemically — the peptides were purchased in 70% minimum purity from Invitrogen (Carlsbad, CA) — instead of by our in vitro method. As shown, the activity profiles for these peptides appeared to match their in vitro–synthesized counterparts, suggesting that the antimicrobial activity was not a relic of the translation mixture. (We also used the chemically synthesized copy of synth–1 to validate the size of our in vitro synthesized copy; see Figure 32-11.) Furthermore, we found that luciferase (a luminescent protein with no antimicrobial characteristics), when synthesized via our in vitro method, had no activity. Thus, we were confident that the translation mixture had no innate antimicrobial activity that may have produced spurious results.

For each peptide/organism combination, we measured a minimum inhibitory concentration (MIC) at which 80% of colony growth was inhibited (see Table 2.5). Many of the peptides appeared to exhibit strong bacteriostatic activity (MIC$_{80}$ ≤ 8 µg/mL). For example, all of the peptides seemed highly active against B. cereus. However, with the exception of synth–3, all of the AmPs appeared specific to gram–positive bacteria. Such specificity is a common characteristic of natural AmPs. For example, the insect cecropins are usually specific to gram–positive bacteria [226]; whereas, the honey bee AmP apidaecin is active only against gram negative bacteria [51]. In general, the underlying reasons for the variations in the susceptibilities of different bacterial species is unknown [280].

We selected the synth–1 family of peptides (*synth–1, *negative–1a, and *negative–1b in Table 2.3) to characterize more thoroughly. These peptides were tested at concentrations up to 50 µg/mL (a typical MIC for moderately active naturally–occurring AmPs) against the gram–negative bacteria. These additional experiments suggested that that *synth–1 was active against C. rodentium at 50 µg/mL, preventing 99% of the bacterial growth with an MIC$_{80}$ of roughly 40 µg/mL. However, *synth–1 was not active against E. coli at this concentration. As expected, the *negative–1a and *negative–1b peptides did not have any activity against either C. rodentium or E. coli at 50 µg/mL.
**Figure 2-10:** Bacteriostatic activity of the three synthetic AmPs against *B. cereus*. The breakouts show photographs of the colonies, which decreased in number with increasing peptide concentration. The inlaid schematic shows the generally accepted mechanism of AmP action: the electrostatic affinity for the outer–leaflet of the bacterial membrane leads to binding and rupture of the cell \cite{226}. (These data were later shown to be not reproducible. Later experiments showed that these peptides had undetectable levels of antimicrobial activity under a more reliable antimicrobial assay. See Section 2.4.3 on page 99.)
Chemically synthesized: negative – 1a synth– 1

<table>
<thead>
<tr>
<th>Peptide conc. (µM)</th>
<th>C. rodentium</th>
<th>C. glutamicum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.96</td>
<td>1.00</td>
</tr>
<tr>
<td>0.2</td>
<td>0.96</td>
<td>1.00</td>
</tr>
<tr>
<td>0.4</td>
<td>0.96</td>
<td>1.00</td>
</tr>
<tr>
<td>0.8</td>
<td>0.96</td>
<td>1.00</td>
</tr>
<tr>
<td>1.6</td>
<td>0.96</td>
<td>1.00</td>
</tr>
</tbody>
</table>

---

Table 2.4: Antimicrobial activity of the synthetic peptides against a variety of bacteria. Each entry in the table shows the relative viability of the bacteria treated with the peptide in comparison to the control (100% viability). The names prepended with a "*" are the names of the bacteria: the ratio of the cell count at a particular concentration of the peptide to the control is indicated. The entries in dark gray show high viability (low antimicrobial action), and the white entries show low viability (high antimicrobial action). The names of the bacteria are prepended with a "*" if the peptides were chemically synthesized rather than produced via in vivo translation.
**Figure 2.11:** SDS–PAGE gel showing the synth–1 *in vitro* translation product (lane B). Lane A shows the translation mixture with no peptide and lane C shows the synth–1 peptide, which was produced via solid phase synthesis and validated by mass spectroscopy.

**Table 2.5:** Minimum inhibitory concentration of the preliminary design synthetic AmPs against a variety of bacteria. In the table MIC<sub>50</sub> is the concentration of peptide, in µg/mL, required to inhibit 50% of the bacterial growth. A “-” indicates that the MIC is greater than 15 µg/mL.

<table>
<thead>
<tr>
<th></th>
<th>synth–1</th>
<th>synth–2</th>
<th>synth–3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MIC&lt;sub&gt;50&lt;/sub&gt;</td>
<td>MIC&lt;sub&gt;80&lt;/sub&gt;</td>
<td>MIC&lt;sub&gt;50&lt;/sub&gt;</td>
</tr>
<tr>
<td><strong>Gram–positive:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>B. subtilis</em></td>
<td>7.5</td>
<td>10</td>
<td>3.5</td>
</tr>
<tr>
<td><em>C. glutamicum</em></td>
<td>4</td>
<td>13.5</td>
<td>3.5</td>
</tr>
<tr>
<td><strong>Gram–negative:</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>E. coli</em></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><em>C. rodentium</em></td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Figure 2-12: Activity of the synth–1 family of peptides against human erythrocytes determined using a procedure described elsewhere [153]. The ordinate shows the degree of hemolysis relative to 50 µg/mL of melittin, which causes complete hemolysis.

In addition, we tested the synth–1 family of peptides for cooperativity with the polymyxin B nonapeptide, which is known to permeabilize the outer membrane of gram–negative bacteria. We found that the nonapeptide did not sensitize *C. rodentium* or *E. coli* to *synth–1, *negative–1a, or *negative–1b, suggesting that the outer membrane may not be the limiting factor in the activity of synth–1 against gram–negative bacteria.

Finally, we measured the activity of the synth–1 family of peptides against human erythrocytes (see Figure 2-12). Our results show that *negative–1a was moderately hemolytic and suggest an HM₅₀ of approximately 60 µg/mL. *Synth–1 and *negative–1b were less active against erythrocytes, with HM₅₀ concentrations (by extrapolation) of roughly 260 and 290 µg/mL, respectively.
2.4.3 Later experimentation

As mentioned briefly in the preceding sections, the experimental data suggesting that synth−1 had antimicrobial activity, were later shown not to be reproducible. Specifically, the *synth−1 peptide was shown to have no antimicrobial activity up to 50 µg/mL. These experiments implied that the data for all of the synthetically generated peptides discussed in the previous section were suspect, with the exception of the data on the hemolytic potential of the peptides. Based on these new findings, we revisited the Amp design problem and developed a more focused approach on the assumption that the original evolutionary methodology would not succeed. In particular, the lack of activity by the *synth−1 peptide indicated that perhaps the Z-score was an inadequate metric for designing Amps, despite its power for annotation.

2.5 Focused design of Amps

2.5.1 Derivation of highly conserved Amp grammars

In the previous section, I described a strategy for designing novel Amps that have a strong emphasis on sensitivity. That is, much effort was expended collecting a database of Amp sequences that was exhaustive so that the set of grammars derived from that database would be exhaustive as well (see Section 2.3 on page 77). The annotation experiments suggest that this strategy is sensitive for discovering novel Amps; however, the lack antimicrobial activity by *synth−1 suggests that perhaps this approach (and the metric Z) is not selective. This lack of selectivity may be rooted in the exhaustive database of Amp sequences, which contains many sequences spanning a wide range of activities. That is, there are some Amp sequences in the database that have very low activity and some with very high activity. In addition, many of the sequences in this database are in precursor form. For the sequences, the precursor undergoes a series of post translational modifications before yielding a mature, active antimicrobial peptide. The regions of the proteins that are cleaved off, or otherwise not responsible for the antimicrobial activity of the peptide are essentially “noise” in the derived set of ~200K grammars derived in the previous section.

In order to focus instead on specificity, not sensitivity per se, we decided to use a database of
well-characterized eukaryotic AmP sequences from the Antimicrobial Peptide Database (APD) [263]. The APD is unique in that it is the only database of antimicrobial peptides that restricts the sequences it catalogs to only those for which there is a large body of experimental data confirming the activity of each AmP. Furthermore, the AmPs listed in the APD are mature in the sense that they are not precursor proteins. Therefore, we know with high confidence that each sequence in the APD has antimicrobial activity and that we are unlikely to be training on sequences that are not responsible in some part for this activity.

As in Section 2.4 on page 88, we used the Teiresias pattern discovery tool to derive regular grammars that occur commonly in the set of 526 well-characterized eukaryotic AmP sequences from the APD. Using these APD sequences, we ran the Teiresias pattern discovery tool with the following settings: \( L = 6, W = 6, \) and \( K = 2 \) (a detailed description of the Teiresias input parameters and associated tools is available in Section 1.5 on page 57). The resulting grammar set was masked from the input sequences and the process was repeated using \( L = 7, W = 15, \) and \( K = 5 \) with the following amino acid equivalency groups \([\text{AG}], [\text{DE}], [\text{FYW}], [\text{KR}], [\text{ILMV}], [\text{QN}], [\text{ST}]\)\). The equivalency groups mean that Teiresias will consider any two characters in the same group to be exactly equivalent. Thus, in the groups above alanine is treated exactly as glycine. In effect, using equivalency groups allows us to find motifs that are more weakly conserved, but that have similar chemistries. (As I will show in Chapter 3 on page 111, Teiresias is unable to use a more fine-grained metric for the similarity between two amino acids. That is, Teiresias can only use equivalency groups to say “equivalent” or “not equivalent” but it cannot use metrics such as “alanine is five arbitrary units in a way from glycine, which is 10 arbitrary units away from leucine.”)

As I discussed in Section 1.5 on page 57, Teiresias outputs its grammars in regular expression format, using wildcards. To make the grammars more selective, we dereferenced each wildcard in the grammars to a bracketed expression, using the same procedure described in Section 2.3 on page 77. That is, we replaced each wildcard with the set of amino acids implied by the grammar’s offset list. Finally, as in Section 2.3, to allow partial matches as short as 10 amino acids, we divided each grammar into sub-grammars using a sliding-window of size 10, resulting in 1551 grammars of length ten.

By design, these 1551 are sensitive for the AmP sequences from the APD. That is, these
sequences from the APD are likely to be matched by the grammars. However, the grammars are not necessarily specific for the APD AmPs. That is, non–AmP sequences may also be matched by the grammars. As discussed above, in our revised strategy, we used the APD sequences to enhance specificity. Here, we reinforce this specificity by eliminating noninformative grammars.

To select only those AmP grammars that are both sensitive and selective, we searched each of the grammars against the nearly exhaustive set of all known AmPs that was assembled in Section 2.3 on page 77. These sequences consisted of the ~750 AmPs from the AMSdb [250], which were supplemented with an additional ~200 antimicrobial peptides from Swiss–Prot/TrEMBL that were not included in the AMSdb. In addition, we searched each of the grammars against sequences from Swiss–Prot/TrEMBL that were not AmPs. Using these two searches, we eliminated grammars that were not at least 80% selective for AmPs. That is, at least 80% of the matches for a single grammar had to come from the set of all known AmPs.

The resulting, final set of 684 ten amino acid grammars was used as the basis set of grammars to design the unnatural AmPs. As before, we say that these ~700 grammars describe the “language” of the AmP sequences and any sequence that is matched by one of the grammars is, at least in part, “grammatical.”

### 2.5.2 Design of synthetic AmP sequences

To design unnatural AmPs, we combinatorially enumerated all grammatical sequences based on the set of ~700 grammars. First, for each grammar, we wrote out all possible grammatical amino acid sequences. So, for example, for the grammar [IVL]K[TEGDK]V[GA]K[AELNH][VA][GA]K produced 600 sequences, where \(3^*5^*2^*5^*2^*2 = 600\), due to the option of choosing one of many amino acids at each bracketed position. There are roughly 3 million such 10–mers that correspond to antimicrobial patterns. Then we wrote out all possible 20 amino acid sequences for which each window of 10 amino acids is found in the set of 3 million 10–mers.

This process is somewhat analogous to the convolution step of Teiresias. That is, we have essentially “stitched” small grammatical sequences together to form longer grammatical sequences. For example, the grammatical 10–mer IKTVAKEVGK would be stitched together with any other 10–mer beginning with the nine amino acid sequence KTVAKEVGK. In this way, the small set of ~700 grammars can give rise to a tremendous number of 20 amino acid
sequences.

From this set, we removed any 20–mers that had six or more amino acids in a row in common with a naturally occurring Amp. There are roughly 12 million such 20–mers, each of which is a “tiling” of ten 10–mers.

In the last section (see page 88), I described a metric, $Z$, for scoring sequences against a database of grammars. Recall that $Z$ essentially is a measure of what fraction of a query sequence is matched by grammars from the database of Amp grammars. However, this metric was not dependent upon how many grammars matched the query sequence. That is, there was no weighting of grammars that were particularly common in Amps relative to grammars that may have only occurred once or twice. Consistent with the approach in the previous section, this metric is sensitive, rather than specific. In our new, more focused approach, we developed a different metric $Q$, which is the degree to which a given 20–mer is grammatical. This score is computed by making a sequence dot plot matrix [162] (see Figure 2-13 on the facing page). In the dot plot, the columns represent the positions, 1–20, of the query 20–mer and the rows represent the concatenated sequences of the ~1000 naturally occurring Amps. A dot is placed in the matrix wherever a grammar matches both a naturally occurring Amp and the 20–mer. Then score $Q$ is then just the number of dots in the matrix. That is, the score $Q$ is the area shown at the bottom of Figure 2-14 on page 104. As shown in the figure, the score $Q$ is more indicative of how homologous a query sequence is to the naturally occurring Amps than the score $Z$ developed in the previous section. (Rather than the area under the curve, the score $Z$ is just the fraction of the query that is matched by grammars.)

In order to choose a representative set of sufficiently different synthetic sequences to test experimentally, we clustered the 12 million sequences using the Mcd–hit software [150] at 70% identity. From these clusters, we chose 42 high scoring sequences to test experimentally. These sequences have varying degrees of similarity to naturally occurring Amps, as determined by sequence alignment. Notably, from each cluster, we took the highest scoring synthetic sequence based on the $Q$ metric. These 42 sequences are shown in the left–hand side of Table 2.6 on page 107.

For each of the 42 synthetic peptides, we also designed a shuffled sequence, in which the order of amino acids was rearranged randomly such that the sequence did not match any gram-
Figure 2.13: An example grammar–based dot plot. The figure shows a query sequence all the top and a single known Amp sequence on the vertical axis along the side. The matrix has an entry for each pair of amino acids between the two sequences. The breakout shows a grammar from our database that matches both of the sequences. Note that both sequences begin a grammar with a proline residue; however, the grammar is not entirely conserved. The next residue in a grammar differs in each of the two sequences. To calculate our scoring metric Q we compute many grammar–based dot matrices using this same approach. For example, see Figure 2.14 on the next page. Activity of rationally designed Amps.
Figure 2-14: An example grammar–based dot plot for computing $Q$. The figure shows a “zoomed out” view of many dot matrices concatenated together. (See the dot matrix shown in Figure 2-13 on the page before.) At the top of the matrix is the query sequence, which is the synthetic, hypothetical AmP that is to be scored. Along the vertical axis lay the concatenated sequences of the set of ~900 known AmPs. The diagonal streaks show places where a grammar matches both the query sequence and a known AmP. At the bottom, to score $Q$ is shown. The score is the area under the curve and is simply a tally of the total number of dots in the dot matrix. War, equivalently, the total length of all streaks in the figure. In this sense, the score $Q$ has a greater emphasis on specificity than did the score $Z$, which was merely be extent of the query sequence covered by grammars.
mars. These shuffled peptides are shown in the right–hand side of Table 2.6 on page 107. Necessarily, these peptides had the same amino acid composition as their synthetic counterparts and thus, the same molecular weight, charge, and pI: bulk physiochemical factors often correlated with antimicrobial activity. We hypothesized that because the shuffled sequences were “ungrammatical” they would have no antimicrobial activity, despite having the same bulk physiochemical characteristics. In addition, we selected 9 peptides from the APD as positive controls (Cecropin P1, Cecropin Melittin Hybrid, Cecropin–A Magainin 2 Hybrid, Melittin, Magainin 2, Hepcidin, Pyrrhocoricin, Ranalexin, and Parasin) and six 20–mers selected randomly from the middle of non–antimicrobial proteins as negative controls.

### 2.5.3 Assay for antimicrobial activity

Each of the peptides shown in Table 2.6 on page 107 was synthesized using solid–phase, Fmoc chemistry on an Intavis Multi pep Synthesizer (Intavis LLC, San Marcos, CA) at the MIT Biopolymers Lab. Mass spectrometry was used to confirm the accuracy of the synthesis — typical purities obtained with the synthesizer were >85%.

We characterized the activity of each synthetic AmP using a broth microdilution assay described elsewhere [274]. This assay measures the MIC at which the peptide inhibits growth of the target organism. The assay is based on the NCCLS M26A and the Hancock assay for cationic peptides (Hancock, NB, Canada). Briefly, serial dilutions of peptides in 0.2% Bovine Serum Albumin and 0.01% Acetic acid were made at 10x the desired testing concentration. Target bacteria were grown in Mueller Hinton Broth (BD, Franklin Lakes, NJ) to OD600 between 0.1 to 0.3 and diluted down to $2 - 7 \times 10^3$ cfu/mL in fresh MHB, as confirmed by plating serial dilutions. Five µL of the peptide dilutions was incubated with 45 µL of the target in sterile, capped, polypropylene strip tubes for 16–20 hours. The minimum concentration that prevented growth based on visual inspection of OD was defined as the MIC. When desired, the samples that did not grow were streaked on an MHB agar plate to see if the peptide was bacteriocidal.

Recombinantly produced standards for Cecropin P1, Cecropin Melittin Hybrid, Melittin, Magainin 2, and Parasin were purchased from the American Peptide Company (Sunnyvale, CA). In antimicrobial assays, four of the five recombinant peptides had identical activities to
the chemically synthesized versions from MIT biopolymers, with the last being one dilution different (Cecropin P1).

2.5.4 Results and conclusions

Table 2.6 on the next page shows the MICs of synthetic peptides against *B. cereus* and *E. coli*, as representative gram positive and gram negative bacteria. (Two of the designed and 4 shuffled peptides were insoluble). Of the 40 soluble designed peptides, 18 had activity against at least one of the bacterial targets at 256 µg/ml or less. Only 2 of the soluble shuffled peptides displayed activity. Thus, the activity is not an artifact of molecular weight, charge, or pI.

Of the the negative controls — 6 peptides randomly selected from the middle of non–antimicrobial proteins from Swiss–Prot/TrEMBL — none had activity. Six of the nine naturally–occurring AmPs in the positive control group show activity and one was insoluble.

Two of the designed peptides, D28 (FLGVFKLASKVFPAVFGKV) and D51 (FLFR-VASKVFPALIGKFKKK), inhibited *B. cereus* growth at 16 µg/mL, which is close to the MICs of the strong positive controls melittin and cecropin–melittin hybrid (8 µg/mL). (Here we use the letter “D” to distinguish a designed peptide from its shuffled equivalent with the same number.) Peptides with gram positive activity are particularly exciting because of the prevalence of drug–resistant nosocomial *S. aureus* and the threat of bioterror agents such as *B. anthracis*, or anthrax. Therefore, we assayed the seven designed peptides that had gram positive activity, including the highly active D28 and D51 peptides, against the Smith Diffuse strain of *S. aureus* and the Sterne strain of *B. anthracis*. As shown in Figure 2-15 on page 108, all seven peptides had activity against both bacteria, whereas only one of the seven shuffled controls had activity. Moreover, two designed peptides, D28 and D51, had activity against *Bacillus anthracis* at 16 µg/mL, which is equivalent to the activity of cecropin–melittin hybrid, a strong natural peptide.

Also, D28 was synthesized by MIT biopolymers 4 separate times and the resulting peptides had consistent activities against both *E. coli* and *B. cereus*.

In an attempt to generate strong, synthetic AmPs, we optimized our best candidate, peptide D28, using a heuristic approach. We created 44 variants of D28 by introducing mutations that were selected to increase positive charge, increase hydrophobicity, remove an interior proline
# Table 2.6: Antimicrobial activity of rationally designed and shuffled peptides. Each entry shows the minimum inhibitory concentration in µg/mL. “+” = MIC greater than 256 µg/mL. ++ = MIC greater than 128 µg/mL, not sufficiently soluble to test at 256 µg/mL.

<table>
<thead>
<tr>
<th>Peptide</th>
<th>Sequence</th>
<th>E. coli</th>
<th>B. subtilis</th>
<th>Shuffled Sequence</th>
<th>E. coli</th>
<th>B. subtilis</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ALFSLASKVPSFSVMVTK</td>
<td>+</td>
<td>+</td>
<td>MVFSVPFKSTVAKLSSA</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>VVFRVASKVFPAVYCTSVKK</td>
<td>128</td>
<td>+</td>
<td>TAKVVFVSYSVVPKRRAC</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>5</td>
<td>FFLGLASKVFPAVYCKVTTRK</td>
<td>64</td>
<td>256</td>
<td>FLPIVFLKFRYSKKTAAGCF</td>
<td>++</td>
<td>64</td>
</tr>
<tr>
<td>6</td>
<td>LSAVGIASKVPVPSVIGAFK</td>
<td>+</td>
<td>+</td>
<td>GVSPF1AVKFGAVALIK</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>7</td>
<td>PVIKLASKVPSVFSLIKR</td>
<td>+</td>
<td>+</td>
<td>SRVPLKSPVKGVSKMFA</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>9</td>
<td>GLMSLVKIDAKLAKQAQK</td>
<td>256</td>
<td>+</td>
<td>GLKDALQLS7VAKAALMG</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>15</td>
<td>SALGRVASKVFPAVYCSITK</td>
<td>+</td>
<td>+</td>
<td>LYSPTCVKAAVSFIGKVS</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>22</td>
<td>LGALFRVASKVFPAVISMVK</td>
<td>256</td>
<td>64</td>
<td>SVPSGAVLFFKRAAVMKLI</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>24</td>
<td>GFIKLASKVPSYCVKVTG</td>
<td>128</td>
<td>+</td>
<td>GSSLGFKSKCALCIVFY</td>
<td>Not soluble</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>PVVPSVASKVPSVLISAKR</td>
<td>+</td>
<td>+</td>
<td>KSFVLLVSRVAAVKSISP</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>28</td>
<td>FLGUFVKLSKVFPAVFLKVGK</td>
<td>64</td>
<td>16</td>
<td>GVSAGAKKVKVLVFPLFLF</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>29</td>
<td>PAVFKLASKVPSYCVKCSR</td>
<td>128</td>
<td>+</td>
<td>KVVYKVIAPCFPKSARVS</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>30</td>
<td>GALFGLASKVFPAVFGAFK</td>
<td>256</td>
<td>+</td>
<td>KVVLFAGAKLFIKASFG</td>
<td>Not enough material</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>SAVGKLASKVPVPSVMVTK</td>
<td>+</td>
<td>+</td>
<td>FMKLAVFGSVPATRAK</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>33</td>
<td>VKDLAKFIATVKQGCGYCL</td>
<td>+</td>
<td>+</td>
<td>ALVYAGKKTAFKVKDCD</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>34</td>
<td>GVVGLASKVPSVFSSTFK</td>
<td>+</td>
<td>+</td>
<td>SVKPGAVSVKGTAVFLFG</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>35</td>
<td>LPVFVRASKVFPAVISKLTT</td>
<td>256</td>
<td>64</td>
<td>KVFIALTVSSPLLAKPVR</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>36</td>
<td>SAVGVSASLVPVLISKVT</td>
<td>+</td>
<td>+</td>
<td>STVKVASKLAVSVSTKGS</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>39</td>
<td>MSKIKAFAKTKVQGKAKG</td>
<td>+</td>
<td>+</td>
<td>AKIAKQSGATIKIFAKGM</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>42</td>
<td>LPAAVFKLASKVPSVFLVK</td>
<td>+</td>
<td>+</td>
<td>VVAKKFFVLKGLAFLPS</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>43</td>
<td>SFVFKLSKVFPAVSLTR</td>
<td>256</td>
<td>256</td>
<td>ASPFVFRSSVFLFVAKK+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>44</td>
<td>SVIGKLASKVPSYCOLSK</td>
<td>+</td>
<td>+</td>
<td>IASAPVVCVKGIKSYISV</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>45</td>
<td>PVGVRASKVFPAVISLKKV</td>
<td>+</td>
<td>+</td>
<td>VRAGKGVAVLPVFKLHV</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>51</td>
<td>FLFRVASKVFPAVINGKK</td>
<td>64</td>
<td>16</td>
<td>RKVAPALIKSDVFLFPPK</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>55</td>
<td>LSFVGRASKVPVSPLSMIK</td>
<td>256</td>
<td>+</td>
<td>SSSIPKMLVRLVFKVSG</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>56</td>
<td>SALKGLASKVPVAVGKTK</td>
<td>+</td>
<td>+</td>
<td>TLGVKAFLVKAISGSSPA</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>57</td>
<td>LGVGSLSLASKVPAVISKVK</td>
<td>64</td>
<td>+</td>
<td>PKGGVSLSVAVKXASAL</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>62</td>
<td>LPVFKLSKVFPAVYCKAS</td>
<td>128</td>
<td>+</td>
<td>PSLYKAKAVFCPSAVAF</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>63</td>
<td>LPVFLKLSKVFPAVISLKK</td>
<td>256</td>
<td>64</td>
<td>VSVKVLFFAPLKLSSFAF</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>65</td>
<td>VFGRASKVFPAVGLTTK</td>
<td>+</td>
<td>+</td>
<td>FKVLISFKGeglALTVT</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>69</td>
<td>SSVFGVASKVPVSIGVKT</td>
<td>+</td>
<td>+</td>
<td>VSVKGVGYKVPSVIKVVAST</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>77</td>
<td>GKKLAKTIKAKQGKAF</td>
<td>64</td>
<td>+</td>
<td>VIAFKTKEAKKLVGQAG</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>81</td>
<td>PFGVRASKVPVSIGVCTR</td>
<td>Not soluble</td>
<td>+</td>
<td>PAVYKISVGFSPVARVTC</td>
<td>Not soluble</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>FCGLSLASKVPVSIGAVKTK</td>
<td>+</td>
<td>+</td>
<td>SKFLAGIFSVPVGKXV</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>83</td>
<td>LPVFKLASKVPVSISKIKT</td>
<td>+</td>
<td>+</td>
<td>KIVKVTIKSPASLVPVF</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>84</td>
<td>GAVGVRASKVPVSISAK</td>
<td>+</td>
<td>+</td>
<td>SVMKSAVPSVFAGKVF</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>85</td>
<td>FGGVASKVPVSYCKVS</td>
<td>+</td>
<td>+</td>
<td>KVGGKSYPCSFVVKVAKSV</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>88</td>
<td>VVFKLASKVPSYCTITK</td>
<td>256</td>
<td>+</td>
<td>VTKCSPVAVYVYLKVTFS</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>96</td>
<td>GALFSLASKVPVAVGLIK</td>
<td>256</td>
<td>+</td>
<td>LPVLFSAIAKVGKLGAK</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
Table 2.7: Antimicrobial activity of rationally designed and shuffled peptides against *S. aureus* and *B. anthracis*. Each entry shows the minimum inhibitory concentration in µg/mL. “+” = MIC greater than 256 µg/mL. ++ = MIC greater than 128 µg/mL, not sufficiently soluble to test at 256 µg/mL.

<table>
<thead>
<tr>
<th>Peptide</th>
<th>Sequence</th>
<th>S. aureus</th>
<th>B. anthracis</th>
<th>Shuffled Sequence</th>
<th>S. aureus</th>
<th>B. anthracis</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>FLGVFKLASKVPAVFAGKV</td>
<td>8</td>
<td>16</td>
<td>GSVVAGAKKVLVVFAPFLF</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>51</td>
<td>FLFRVASKVFPPALIGKFKKK</td>
<td>16</td>
<td>16</td>
<td>RVAPALIKSVFLKFKKG</td>
<td>128</td>
<td>256</td>
</tr>
<tr>
<td>22</td>
<td>LGALFRVASKVFPAVISVVK</td>
<td>64</td>
<td>64</td>
<td>SVPSVGAVLFFKRAAVMKLI</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>63</td>
<td>LPVLFKLASKVPAVFSSLK</td>
<td>128</td>
<td>128</td>
<td>VSVPKLFAPLKSLSSFAD</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>5</td>
<td>FLGLASKVFPAVYCKVTRK</td>
<td>256</td>
<td>128</td>
<td>FLPVLSKFRYSKKTAKAFC</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>43</td>
<td>SFLFKLASKVPSVFSAHTR</td>
<td>256</td>
<td>128</td>
<td>ASPTVFRRSVFLSVVAKK</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>35</td>
<td>LPVFRVASKVFPAHSLT</td>
<td>256</td>
<td>128</td>
<td>KVFIATLVSSFLLAKPRV</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Figure 2.15: Activity of rationally designed AmPs against *S. aureus* and *B. anthracis*. The figure shows that shuffled peptides (the hashed bars) tend to be grouped on the right side of the plot, indicating that they have little or no antimicrobial activity. Only one of the shuffle peptides shows activity; however, it appears twice on the plot, once at 128 µg/mL against *S. aureus* and once act 256 µg/mL against *B. anthracis*. In contrast, all of the designed peptides show some degree of activity. The most highly active peptide is that the left–hand side of the plot.
residue, and improve segregation of positive and hydrophobic residues based on a helical projection. 16 of the 44 D28 variants showed improved activity against *E. coli* or *B. cereus*. All of the D28 variants with improved activity against *B. cereus* included a mutation at an internal proline, either to lysine or glycine. D28 and six of its variants were assayed for bacteriocidal activity, and all had activity within a 2–fold dilution of their MIC. One variant had MICs of 16 µg/mL against *E. coli* and 8 µg/mL against *B. cereus* (relative to 64 and 16 µg/mL, respectively, for D28).

We suspect that our linguistic approach to designing synthetic AmPs is successful due to the pronounced modular nature of naturally–occurring AmP amino acid sequences. As we have shown, this approach can be used to rationally expand the AmP sequence space without using structure–activity information or complex folding simulations. The peptides designed in this work are different from previously designed synthetic AmPs [246, 251] in that they bear limited homology to any known protein, which may be desirable for AmPs used in clinical settings. Some critics argue that widespread clinical use of AmPs that are too similar to human AmPs will inevitably elicit bacterial resistance, compromising our own natural defenses and posing a threat to public health [29]. We hope that this approach will help to expand the diversity of known AmPs well beyond those found in nature, possibly leading to new candidates for AmP–based antibiotic therapeutics. Our designed AmPs show some degree of homology with natural AmPs because the grammars are based on native sequences. Peptide D28, for example, was matched by grammars derived from 11 natural AmPs including brevinin, temporin, and ponericin. However, Smith–Waterman alignments of our designed peptides against all natural AmPs in the Swiss–Prot/TrEMBL database reveal that the degree of homology is, by design (see Methods), limited. In particular, our two most active peptides, D51 and D28, have 50 and 60% sequence identity with the nearest natural AmP, respectively. Peptide D51 has 6 semi–conservative and 4 nonconservative substitutions relative to its closest neighbor, Ponerin W5. Our linguistic design approach may be most valuable as method for rationally constraining a sequence–based search for novel AmPs. Diverse leads generated by our algorithms may be optimized using approaches described in the literature [117]. But, the linguistic approach described here has a number of limitations. First, sequence families that are poorly conserved on an amino acid level would not benefit from this approach. Second, we suspect that the small
size of AmPs is helpful. Due to the simple nature of regular grammars, they would be less useful for designing larger proteins and, in particular, proteins with complex tertiary or quaternary structures.
Chapter 3

A generic motif discovery algorithm

3.1 Introduction

In the previous chapter, I described the use of regular grammars for modeling the primary sequences of antimicrobial peptides. In that work, I showed that our specific approach to the design of novel AmPs yielded peptides with strong antimicrobial activity. However, recall that, in order to achieve specificity with some degree of sensitivity, the grammars had to be split into tiled 10 amino acid windows for increased sensitivity and then compared against a database of non–AmP sequences in order to increase specificity by throwing out uninformative grammars. This is because, as discussed in Chapter 1 on page 19, regular grammars are inherently more “coarse grained” than other models such as position weight matrices. Thus, to design AmPs, we had to use large sets of redundant, overlapping regular grammars. In such situations, the underlying sequence information might be better modeled by a position weight matrix or many other kinds of models.

In this chapter, I present a GEneric MOtif DIscovery Algorithm (Gemoda) for sequential data. Gemoda is a motif discovery tool very similar to Teiresias; however, Gemoda’s output motifs are representation–agnostic: they can be represented using regular expressions, position weight matrices, or any number of other models. In addition, Gemoda can be applied to any dataset with a sequential character, including both categorical data such as protein and amino acid sequences, and real–valued data such as the price of a stock as a function of time. As I show in the following sections, Gemoda deterministically discovers motifs that are maximal in
composition and length. As well, the algorithm allows any choice of similarity metric for finding motifs. I demonstrate a number of applications of the algorithm, including the discovery of motifs in amino acids sequences, a new solution to the (l,d)–motif problem in DNA sequences, and the discovery of conserved protein sub–structures.

The research described in this chapter is drawn largely from two publications:


Throughout this chapter, the use of the pronoun “we” refers to the authors of these manuscripts.

### 3.2 Motivation

As discussed in Chapter 1 on page 19, motif discovery encompasses a wide variety of 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 [119, 174].

Motif discovery in sequence analysis typically involves the discovery of binding sites, conserved domains, or otherwise discriminatory subsequences. There are many publicly–available tools, a large number of which are listed in Section 1.5 on page 55, each of which is quite adept at addressing a specific subclass of motif discovery problems. Some of the commonly–used tools for motif discovery in nucleotide and amino acid sequences include MEME [19], Gibbs sampling [147], Consensus [115], Block Maker [113], Pratt [132], and Teiresias [207]. Newer, less-widely used tools include Projection [45], MultiProfiler [136], MITRA [78], and ProfileBranching [199]. This list is not intended to be exhaustive; however, it is indicative of the wealth of options available for solving such problems (see also Tables 1.4 & 1.5 in Chapter 1 on pages 59 & 67, respectively).
All of the existing motif discovery tools for nucleotide and amino acid sequences can be classified on a spectrum ranging 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.

One of the primary motivation for this work is the modeling of cis–regulatory sequences. We found that regular expressions are poor representations of binding sites and that, instead, these were better captured with PWMs. From a biological perspective, this makes more sense — the $k_D$ of binding between the trans and cis factors are probabilistic, not deterministic. Thus, in order to model these sites using regular grammars or regular expressions, one must, in general, use combinations of patterns in an effort to piece together the information that would be contained within a PWM from many regular expressions.

Consider the following example. The LexA regulon consists of 9 gene sequence that are regulated by a single protein trans factor. The binding site of this trans factor is found in 8 of these sequences. Using the Teiresias motif discovery tool, with parameters $L = 10$, $W = 20$, $K = 5$ (see Section 1.5 on page 57) returns the following patterns:

| 5 4 | CTGTATAT.....CAG   | 0 355 0 376 4 298 6 326 7 363 |
| 5 5 | CTGTAT....A..CAG    | 0 376 1 322 4 298 6 326 7 363 |
| 5 5 | ACTGTA......A..CAG  | 0 375 1 321 3 358 4 297 7 362 |
| 5 5 | CTGTA.AT..A..CAG    | 0 376 3 359 4 298 6 326 7 363 |
| 6 5 | CTGTA.AT......CAG   | 0 355 0 376 3 359 4 298 6 326 7 363 |
| 5 5 | ACTGT.T.....A..CAG  | 0 375 1 321 4 297 5 307 7 362 |
| 5 5 | ACTGT....T..A..CAG  | 0 375 3 358 4 297 5 307 7 362 |
| 5 5 | CTGT.T.T..A..CAG    | 0 376 4 298 5 308 6 326 7 363 |

where the above grammars have been left and the native output form of Teiresias. The numbers on the right hand side indicate the offset list for each grammar. So, collectively, these patterns hit 7 of the 8 sequences; however, none of the patterns individually hits more than 5
sequences. Basically, this is because regular expressions don’t capture such sites well.

And this chapter, I described Gemoda: a motif discovery tool that has many of the strengths of Teiresias, but can find motifs that are best represented as PWMs. The details of Gemoda are discussed in the later sections of this chapter. But here, for motivation, consider the following output from the Gemoda all over them. If a user tells Gemoda to find all patterns in the LexA sequences such that, on a pairwise basis, each window of 20 nucleotides in each instance contains at least 10 nucleotides in common to each other instance, and the pattern occurs in at least 8 sequences; Gemoda returns only a single pattern:

```
0 353 TGCTGTATATACCTACAGCA
0 374 AACTGTATATACACCCAGGG
1 320 TACTGTATAGCATAACAGTA
2 230 ACCTGAATGAATATACAGTA
3 357 TACTGTACATACATTACAGTA
4 296 TACTGTATATACCTACAGTA
5 306 AACTGTTTTTTATCCAGTA
6 324 ATCTGTATATACCTACAGTA
7 361 TACTGTATATAAAAAACAGTA
```

where, instead of one pattern per line, each line represents one of the offsets and the numbers on the left-hand side are, collectively, the offset list. Notice that here, only a handful of the positions within the pattern are fully conserved. However, most of the positions have “preferences.” For example, the seventh position is mostly A. This pattern can be expressed as a PWM, has in Figure 3-1 on the next page, thus preserving these preferences in the matrix probabilities. Notably, this pattern is exactly the experimentally determined motif.

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 [249], whereas the regular expression–based tools are well–suited to finding conserved domains in large protein families [208]. Generally, it can be difficult to know a priori which motif discovery tool will be right. Accordingly, there is an unmet need for motif discovery tools that can use a variety of motif models.
Figure 3-1: Alignment representing the LexA cis–regulatory binding site. Part A) of the figure shows the aligned sequences colored to indicate the degree of conservation. Part B) of the figure shows a sequence logo representing the information content of a PWM computed from the alignment of the motif instances. Part C) of the figure shows a sequence logo, wherein the height of each letter is proportional to its frequency, rather than to the information content it in codes as is the case in part B).
3.3 Algorithm

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 [207]; Winnower [195]; the algorithm by [163]; and a variety of algorithms for association mining [277, 278]. 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.

The principle difference between Teiresias and most frequent itemset mining algorithms is that Teiresias acts on categorical sequential data, usually biosequences or integers. Most frequent itemset mining tools use market basket data sets, for example, a collection of products that a customer bought. Patterns in market basket data can be used to predict what other products a customer might buy (this is how Amazon.com works). The difference between categorical sequential data and market basket data (both are stochastic in that they consist of discrete values sampled from some real space) is that the former is ordered, whereas the latter is an unordered set. For similarly sized datasets, this makes sequential pattern discovery much easier. However, typically sequential datasets, such as biosequences or time-series stock data, are much larger. For example, a person may only purchase a few products from Amazon; however, gene sequences can consist of may thousands of characters.

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, 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 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 3-2 on the following page). 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.

### 3.3.1 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\}.

To demonstrate this notation and how he can be used to represent real–valued sequential data, rather biosequences consider the following example. Say we have two small peptides and we are interested in their structural properties. For each amino acid, we have a two–dimensional feature vector. The first feature is the hydrophobicity index [14] and the second is the size of the
A sketch showing the flow of 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 windows. (1) In the comparison stage, each window is compared to each other window on a pairwise 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 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 that are clustered together are shown by a clique or maximal fully-connected subgraphs in the thresholded matrix \( A \). Finally, the clustered windows 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.
amino acid: 1 if it is over the 50\textsuperscript{th} percentile and 0 otherwise. The two peptides are AIKDWR and DIHV. Our two sequences are then

\[
\text{seq}_0 = \begin{pmatrix} 0.61 & 2.22 & 1.15 & 0.46 & 2.65 & 0.60 \\ 0 & 1 & I & 0 & 1 & 1 \end{pmatrix},
\]

\[
\text{seq}_1 = \begin{pmatrix} 0.46 & 2.22 & 0.61 & 1.32 \\ 0 & 1 & I & 0 \end{pmatrix},
\]

such that

\[
s_{0,0,0} = 0.61
\]

\[
s_{1,0,0} = 0.46
\]

\[
s_{1,1,1} = 1
\]

\[
s_{0,3,1} = 0
\]

\[
s_{1,2,0} = 0.61,
\]

and so on.

Typically, one seeks motifs of a minimal, domain-dependent length. We denote this minimum length by \(L\) (similar to Teiresias) 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\textsuperscript{th} window of size \(L\) in the 5\textsuperscript{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 \(\mathcal{H}(s_{i,j:L-1}) \mapsto q \in [1, N]\). (For simplicity, we define \((s_{i,j})\) to be \(s_{i,j+1}\), unless \(s_{i,j+1}\) does not exist, in which case \((s_{i,j})\) is undefined.) Similarly, \(\mathcal{H}^{-1}(q) \mapsto (s_{i,j:L-1})\) such that \(i \in [1, n]\) and \(j \in [1, W_i - L + 1]\).

We also define a similarity function \(\mathcal{S}(s_{i,j:L-1}, s_{q,z:L-1})\), 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, \(\mathcal{S}\) 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 \( W(p) \) and a list of locations in the data where the motif occurs, \( \mathcal{L}(p) \). A motif has the property that the locations in \( \mathcal{L}(p) \) meet some predefined clustering requirements (discussed below) based on the similarity function \( \mathcal{S} \) for each window of length \( L \) within the motif. The support of a motif is equal to the number of its occurrences (or, equivalently, “instances” or “embeddings”), \( |\mathcal{L}(p)| \).

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

1. The motif’s width cannot be extended in either direction (left or right) without producing a motif with fewer embeddings (i.e., without \( |\mathcal{L}(p)| \) decreasing); and

2. The motif is not missing any instances, i.e. \( \mathcal{L}(p) \) 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. A simple natural-language example illustrating how each phase proceeds is included in the supplementary materials.

### 3.3.2 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 3-2 on page 118). Formally, \( A_{i,j} \) is equal to \( \mathcal{S}(\mathcal{M}^{-1}(i), \mathcal{M}^{-1}(j)) \) = \( \mathcal{S}(s_{i,j}^{L-1}, s_{q,z}^{L-1}) \).

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

Consider the following example. Say we have two DNA sequences — seq–0 = AATTGGCC and seq–1 = GATAGGA — and that we are interested in patterns that are at least \( L = 5 \) bases long. Also, here, we will consider the sequences as just a series of characters, that is, a one–
dimensional feature vector. We will define $F(A, B)$ to be the Hamming distance: the number of mismatches between string $A$ and string $B$.

There are 7 windows of size 5 in the sequences:

$$M^{-1}(0) = s_{0:0:4}$$
$$= AATTG$$

$$M^{-1}(1) = s_{0:1:5}$$
$$= AATTG$$

$$\vdots$$

$$M^{-1}(6) = s_{1:2:6}$$
$$= TAGGA.$$  

The members of the matrix $A$ are computed as follows:

$$A(0, 0) = F(AATTG, AATTG) = 0$$
$$A(0, 1) = F(AATTG, ATTGG) = 2$$
$$\vdots$$

$$A(2, 5) = F(TTGGC, ATAGG) = 3$$
$$\vdots$$

$$A(6, 6) = F(TAGGA, TAGGA) = 0.$$
The matrix $A$ is then

$$A = \begin{bmatrix}
0 & 2 & 5 & 5 & 2 & 3 & 4 \\
-0 & 3 & 5 & 3 & 1 & 4 \\
-0 & 0 & 2 & 5 & 3 & 2 \\
-0 & -0 & 5 & 5 & 3 \\
-0 & -0 & -0 & 4 & 4 \\
-0 & -0 & -0 & -0 & 4 \\
-0 & -0 & -0 & -0 & -0
\end{bmatrix},$$

where the $-$ is used because the matrix is symmetric.

Obviously, depending on the type of sequential data being analyzed, the similarity function should be changed accordingly. However, any kind of data can always be used to produce a generic similarity matrix $A$, which is the input to the next phase of the algorithm. From this point onward, the algorithm data-agnostic in the sense that subsequent phases act only on $A$ and $M$ — they are independent of the specific data that produced these structures.

### 3.3.3 Clustering phase

The purpose of the clustering phase is to use the similarity matrix $A$ to group similar windows into clusters. These clusters will become “elementary motifs” from which the final, maximal motifs will be constructed in a manner similar to the Teiresias algorithm.

We define a clustering function $C(A) = c^L = \{c^L_1, c^L_2, \ldots, c^L_Z\}$ where each $c^L_i$ is a set of indices in $A$ and $c^L_i[q]$ is the $q^{th}$ member of $c^L_i$. Note that $C$ can be any function; common clustering functions include hierarchical clustering, $k$–nearest–neighbors clustering, and many others. We call each $c^L_i$ 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 $c^L_i$ may have a non-null intersection with any $c^L_j$. That is, a single window may appear in an arbitrarily large number of clusters.
3.3. ALGORITHM

3.3.4 Convolution phase

The purpose of this phase is to “stitch together” the elementary motifs to generate the final, maximal motifs [207]. For the purposes of Gemoda (and consistent with the above concept of convolution), we say that a motif \( h \) of width \( \mathcal{W}(h) > L \) meets the similarity criterion if for each window of length \( L \) completely within the motif, all instances participate in a cluster together based on \( \mathcal{S} \) and \( \mathcal{C} \). 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, \( c_i^k \preceq c_j^l = c_{r}^{l+i} \), where \( c_{r}^{l+i} \) is the set of those indices \( q \) in \( c_i^k \) such that \( \mathcal{M}(\mathcal{M}^{-1}(c_i^k[q]) + 1) \) is in \( c_j^l \). That is, \( c_{r}^{l+i} \) is the set of indices in \( c_i^k \) that are located, in the sequences \( S \), one position earlier than the indices in \( c_j^l \). \( c_{r}^{l+i} \) is then a motif of length \( L + 1 \).

We define the operation “\( \sqsubset \)” as follows: \( c_i^k \preceq c_j^l \sqsubset c_{r}^{l+i} \) is true if the set of indices \( c_i^k \preceq c_j^l \) is a subset or a superset of the indices in any member of \( c_{r}^{l+i} \). 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 \( c_i^k \preceq c_j^l \sqsubset c_{r}^{l+i} \), then all super– or sub–sets of the proposed convolved motifs are removed from \( c_{r}^{l+i} \); these windows are then taken together with the proposed motif, and the union of those sets of windows is returned to \( c_{r}^{l+i} \).

Our objective is to find all the maximal motifs in the sequence set using the elementary patterns. We do this by performing \( c_i^k \preceq c_j^k \) 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 \( \mathcal{P} \), 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 \( \mathcal{P} \) given appropriate clustering functions (see supplementary materials).
3.4 Implementation

3.4.1 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 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 3-2 on page 118 and 3-10 on page 142). When constructed as such, we have found that clustering functions based on finding either cliques\footnote{We define a clique as a maximal, fully–connected subgraph.} or connected components (maximal disjoint subgraphs) can be effective for motif discovery in diverse applications.

In the case where the clustering function $\mathcal{C}(A)$ is chosen such that each $c_L^C$ is a clique in the $g$–thresholded $A$ matrix, the Gemoda algorithm has a guarantee of compositional and length maximality, relative to the threshold $g$. 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 inductive proof in the supplementary material).

Clique enumeration is NP–complete \cite{90,248}; 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 $g$).

In the case where the clustering function $\mathcal{C}(A)$ is chosen such that each $c_L^C$ is a maximal disjoint subgraph in the $g$–thresholded $A$ matrix (i.e., $c_L^C$ 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, within the reach of many personal computers. Furthermore, the upcoming examples given in this work can all be done in reasonable times. The amino acid sequence example and protein structure example take at most tens of seconds on an average desktop PC, while the hardest of the DNA sequence examples takes two hours. These times are more than reasonable given the exhaustive guarantees provided by the algorithm.

### 3.4.2 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 $k$. 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.

### 3.4.3 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](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 [118] — or as hidden Markov models, using the popular HMMer software [72].

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.

The Gemoda software is written in the C programming language and is described in detail in Chapters C and B in the Appendix (page 421). The code is segmented in such a way as to allow the extension of the algorithm to varieties of sequential data that were not anticipated by the authors. Furthermore, where possible the code was crafted to be “object–oriented like” for maximum readability. The software makes extensive use of the GNU Scientific Library [1] and the popular Basic Linear Algebra Subprograms (BLAS) [37, 68, 69] to speed–up computationally intensive operations associated with the discovery of motifs in three–dimensional protein structures and other real–valued data.

3.4.4 Motif Significance

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, \( \Phi \), such that

\[
\Phi_{i,j} = \left| \{(x, y) : \eta_{x,y} = i, \chi_{x,y} = j, 0 \leq x, y \leq N \} \right| \tag{3.1}
\]

(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:
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.

3.4.5 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 “\( \sqsubseteq \)” 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 pseudocode sketch in 3.3 on the facing page faithfully encapsulates the inductive algorithm described above.

### 3.4.6 Two simple examples

To demonstrate exactly how the algorithm works, we now provide two simple, natural-language examples along with a step-wise narrative of the Gemoda algorithm and demonstrations of how the examples would be run using the software implementation of the Gemoda algorithm provided by the authors and described in Chapter B on page 225.

**Example 1:** Consider two sequences, ABCDEFG and ABCEFDG, that would be represented with the following Fasta–formatted file:

```
> Sample 1
ABCDEFG
> Sample 2
ABCEFDG
```

Using a window of length 3, a minimum similarity of 1, a clique–finding clustering method, and the similarity function defined as the identity matrix (the same function described by the reviewer), the 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
```
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 \sim c_j
        if |f| ≠ 0
          if f ⊇ c^{n+1} = false
            c^{n+1} := c^{n+1} ∪ f
          else
            choose maximal(f, c^{n+1})
            fi
          fi
        fi
      fi
    od
    if ismaximal = true
      P := P \cup c^n
    fi
  od
  n := n + 1
end

Figure 3-3: Pseudo-code for the Gemoda convolution. The figure shows the recursive algorithm used during the convolution stage of Gemoda. The algorithm produces only maximal motifs and discards motifs that are not maximal in support at each level. Subsequent levels progress to motifs of larger length. As discussed in the text, when Gemoda uses a clique-finding clustering phase, the convolution phase guarantees that the algorithm is both maximal and exhaustive.
Given this command, Gemoda finds the maximal motif $ABC...FG$. How this happens is illustrated in Figure 3-4 on the facing page.

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 allowing the motif to extended past the letter $D$. 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.

Now consider the same sequences with $g = 2$. As described in eariler, a motif of width $W \geq L$ must meet the clustering and similarity requirements for each pair of $L$–length windows that is completely within the motif. In this example, since the third and forth pairs of aligned windows, cde & ced and def & edf, do not meet the criterion of $g = 2$ for a similarity function based on the identity matrix, they are not in elementary motifs that can be convolved.
Figure 3-4: A natural language example illustrating the steps that Gemoda takes. In a), we see the three words, or sequences, being broken into overlapping windows of three 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 a dot 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.
As shown, the first two pairs of $L = 3$ length windows, which surpass the $g = 2$ threshold, form elementary motifs and are convolved together. However, because the third pair does not meet the criteria (and thus form an elementary motif) it is not convolved. A similar logic applies to the final two windows. Thus, the final, convolved, maximal motifs in this problem are $abc\text{.}$ and $fg$, and $abc\text{.}fg$ is not a maximal motif motif (with $L = 3, g = 2$).

**Example 2:** Suppose we have a set of three words,
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.

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_{1}\!\!-c_{6} and
Figure 3-5: A second natural language example illustrating the steps that Gemoda takes. 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.
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 $c_0^L \bowtie c_1^L$, $c_1^L \bowtie c_0^L$, $c_1^L \bowtie c_1^L$, and $c_0^L \bowtie c_0^L$. 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 $c_1^L \bowtie c_0^L$ and $c_1^L \bowtie c_1^L$, 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 $c_0^{L+1} \bowtie c_0^{L+1}$ 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

\[
\{\{\text{MOTIF}, \text{POTIO}\}, \{\text{MOTI}, \text{MOTO}\}\}.
\]

For identity similarity and connected components clustering, the final list of motifs is

\[
\{\{\text{MOTIF}, \text{POTIO}\}, \{\text{MOTI}, \text{MOTO}, \text{POTI}\}\}.
\]

For vowel/consonant similarity and either clustering method, the final list of motifs is

\[
\{\{\text{MOTIF}, \text{MOTOR}, \text{POTIO}\}\}.
\]

### 3.5 Application

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 $\mathcal{S}$ that returns a sequence alignment score, such as the bit–score from an amino acid scoring matrix (e.g., the popular Blosum matrices [109]).

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 [21].

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 $\mathcal{S}$ 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.

The sequences for this example are distributed with the source code for the software implementation of Gemoda written by the authors (see Chapter B on page 225). Using the software, this example would be run as follows, assuming the protein sequences are in a file called “spot.fa”:

```
$ gemoda-s -i spot.fa -l 50 -g 50 -k 8 -m BLOSUM62
```

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 3-7 on page 139 as an alignment of the regions that correspond to instances of this motif (see also Figure 3-10).
Figure 3-6: Guanosine-3′,5′-bis(diphosphate) 3′-phosphorylhydrolase (ppGpp)ase (Penta-phosphate guanosine-3′-phosphorylhydrolase) sequences. These seven 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 [21].
A comparison with the known protein domains in the NCBI Conserved Domain Database (version 2.02) [165] reveals that this motif captures the RelA_SpoT domain (CDD PSSM–id 15904).

The remaining three motifs are not present in the CDD database. However, further inspection using the tools available from the PFAM database [25] revealed that they composed the left, middle, and right regions of the HD domain [13]. 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–base 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.

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 found that, with the Gemoda input parameters described above and using random sequences selected from Swiss–Prot (Release 45.0) [22], the target motifs could be detected in an 8–fold majority of spurious sequences.

### 3.5.2 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 [73]. However, these homologs frequently show conserved tertiary structures [66], 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 [73]. Some of the more popular tools include SSAP [185], VAST [160], Dali [120], and Mammoth [187]. The Gemoda algorithm, when used for struc-
Figure 3-7: The RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences.
tural motif discovery, is most similar to the Sarf algorithm [4, 5] and, to a lesser degree, algorithms by [125] and [133]. 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 [144]). Here we use “minimized” to indicate that the protein structures are optimally super–imposed via rigid–body rotation and translation [15, 122]; occasionally this term is implicit. 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 9 Å 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.

Here, we demonstrate how the Gemoda algorithm can be used for structural motif discovery by “discovering” the structural homology between the human galactose–1–phosphate uridylyltransferase (PDB id 1HXQ) [266] and fragile histidine triad proteins (PDB id 3FIT) [152], originally reported elsewhere [121]. 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 Å or less (based on super-
Figure 3-9: Hidden Markov model representation of the RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences. In this figure, the boxes represent the different possible Markovian states at the first few positions in the motif shown in Figure 3-7 on page 139 [220].
Figure 3-10: 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 3-7 on page 139. 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 3-11: Alpha carbon trace projection used by Gemoda

position 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 3-12 on the next page). 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.

3.5.3 Motif discovery in nucleotide sequences and the \((l,d)-motif problem\)

Introduction

Four years ago, Pevzner and Sze [195] 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
Figure 3-12: A motif showing structural conservation between the human galactose-1-phosphate uridylyltransferase and fragile histidine triad proteins originally reported by Holm and Sander [121]. 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). See also Figure 3-13 on the next page.
Figure 3-13: The human galactose-1-phosphate uridylyltransferase and fragile histidine triad structural motif (see Figure 3-12 on the preceding page) in Gemoda’s 3-D structure viewer, which was written by the authors for viewing output from gemoda–p.
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 presented two algorithms that looked towards solving this problem; Buhler and Tompa 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 original \((l,d)\)--motif problem 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) [45, 46].

As discussed by Buhler and Tompa [45], 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 \(10^{-7}\)) [45, 46].

In a more realistic scenario, a researcher may not know the size \(l\) of the motif \textit{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 \textit{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 \textit{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.

We say that a motif, \(p\), is just a data structure with two features: a width, \(\mathcal{W}(p)\), and a list
of locations in the data where the motif has been embedded, $\mathcal{L}(p)$. A motif has the property that the locations in $\mathcal{L}(p)$ are all within a Hamming distance of $2d$ from each other over every window of size $l$.

We will call the Hamming distance function $H$, where $H$ 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.

**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 [115], Gibbs sampling [147], and MEME [19], 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 [195], 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, our algorithm 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, our algorithm 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.
Our algorithm 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. Our algorithm 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.

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.

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 [195], 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. With Gemoda, we can easily identify this instance without any additional work or manipulation. The sequence logo for the planted motif from Pevzner and Sze’s initial dataset is shown in Figure 3.14 on page 153; the consensus sequence is GGCTTTGTAGCTAAC. The “accidental” instance of the embedded motif that can be identified using Gemoda is GGATTGATAGCTAAG.

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 .9 or .95. Examples of the performance of other algorithms are presented in Table 3.1. Clearly, our algorithm 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 3.1), 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.

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 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 our algorithm should be of significant utility for common applications.

**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 problem will lose significant advantage in this more realistic, extended set of circumstances. Our algorithm 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 (discussed qualitatively) which demonstrate the specific benefits of our tool for pattern discovery on \((15,4)\) problems beyond the restricted version.

**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, 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. These results would sometimes have significant overlap with initial planted motif, 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 one constructed from background noise. Our algorithm, on the other hand, returned the initially planted motif along with other smaller patterns that still met the criteria for classification as a motif.

**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. A study using PROJECTION found results that rarely approached acceptable levels and more frequently approached performance coefficients expected from purely random guessing. Again, though, our algorithm solved the problem with only a small increase in computation time over solving the original \((l,d)\)-motif problem.

Identifying natural cis–regulatory elements

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 [217], Gemoda’s top–scoring motif was used to generate the sequence logo found in Figure 3-14. 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. Of course, the difficulty of DNA motif discovery problems varies greatly, and this is only one straightforward example of such problems.

The parameters used for this search were \(L = 20\), \(g = 10\), and \(k = 6\) with the identity matrix scoring scheme and clique–based clustering described above. 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 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 or intuitive and that there was some tolerance in the results for slight perturbations in parameters.

Conclusions

The benefit of our proposed algorithm 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.

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 [120]. The convolution stage could also be made faster by using heuristical, non–exhaustive convolution methods. Also, the clustering phase could be expedited by using approximate clique finding methods.

The lack of an underlying model in Gemoda is a major strength, as this facet of the algorithm allows exhaustive enumeration of motifs that is difficult for methods using complex motif representations. In addition, this aspect of Gemoda makes comparing nucleotide sequences just as easy as comparing real–valued data, like gas chromatography–mass spectrometry (GC–MS) datasets, which may follow different motif models (Styczynski et. al., in preparation).

One weakness may be that the Gemoda algorithm does not natively employ iterative steps for motif discovery. In that sense, the algorithm is similar to Teiresias [207] and MITRA [78]. However, because it employs a user–defined scoring metric (and clustering function) there is nothing to prevent such iteration per se. For example, the output motifs from a run of Gemoda could be used to recompute a refined scoring function. Using amino acid substitution matrices, this would be in the spirit of the method used to compute the Blosum [109] matrices from the Blocks database [113].

Furthermore, the Gemoda algorithm could be modified to find gapped motifs. Gemoda is capable of finding gapped motifs in which the gap length is fixed and small relative to the size
of the flanking conserved regions. However, motifs with larger, variable length gaps cannot be detected natively by Gemoda. In this respect, Gemoda is similar to MEME [19], Teiresias [207], and Block Maker [113]. Other tools, including Consensus [115] and the Gibbs sampler [147], have been altered from their original formulation to account for gaps.

It may be possible to alter the convolution step to allow for large or variable–length gapped motifs. Another option is to look for maximal motifs whose offsets are highly correlated. Our studies indicate that such post hoc analysis of Gemoda’s output can usually find well–conserved gapped motifs, including those with variable gap lengths, as was the case for the (ppGpp)ase example.

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.
Table 3.1: Performance on a range of \((l,d)\)-motif problems with synthetic data. Data from other algorithms are from Buhler and Tompa [46]. GibbsDNA, WINNOWER, and SP-STAR are averaged over eight random instances, while PROJECTION is averaged over 100 random instances. Computation times for our proposed algorithm are averaged over three random instances.

<table>
<thead>
<tr>
<th>(l)</th>
<th>(d)</th>
<th>GibbsDNA</th>
<th>WINNOWER</th>
<th>SP-STAR</th>
<th>PROJECTION</th>
<th>Proposed algorithm</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2</td>
<td>0.20</td>
<td>0.78</td>
<td>0.56</td>
<td>0.80</td>
<td>1.00</td>
<td>8 min</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>0.68</td>
<td>0.90</td>
<td>0.84</td>
<td>0.94</td>
<td>1.00</td>
<td>&lt; 1 min</td>
</tr>
<tr>
<td>12</td>
<td>3</td>
<td>0.03</td>
<td>0.75</td>
<td>0.33</td>
<td>0.77</td>
<td>1.00</td>
<td>10.5 h</td>
</tr>
<tr>
<td>13</td>
<td>3</td>
<td>0.60</td>
<td>0.92</td>
<td>0.92</td>
<td>0.94</td>
<td>1.00</td>
<td>10 min</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>0.02</td>
<td>0.02</td>
<td>0.20</td>
<td>0.71</td>
<td>1.00</td>
<td>&gt; 3 months</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
<td>0.19</td>
<td>0.92</td>
<td>0.73</td>
<td>0.93</td>
<td>1.00</td>
<td>6 h</td>
</tr>
<tr>
<td>17</td>
<td>5</td>
<td>0.28</td>
<td>0.03</td>
<td>0.69</td>
<td>0.93</td>
<td>1.00</td>
<td>3 weeks</td>
</tr>
</tbody>
</table>
CHAPTER 3. A GENERIC MOTIF DISCOVERY ALGORITHM
Chapter 4

Other exercises in motif discovery

4.1 Introduction

The previous two chapters of this thesis were focused on unsupervised methods for motif discovery, with an emphasis on grammatical models. Specifically, in Chapter 2 I demonstrated how regular grammars can be used to model and design novel antimicrobial peptides. In Chapter 3, I addressed some of the weaknesses of grammar–based motif discovery tools by developing a new approach that is generic in the sense that it is applicable to many different kinds of sequential data and is model agnostic. In this chapter, I continue this trend away from the core issues of grammar–based motif discovery, to examine many closely related topics and different approaches to motif discovery.

The first section of this chapter describes the development of an efficient tool for matching regular grammars against large databases of sequences. The topic of the second section is the evolution of amino acid scoring matrices over time. As described in Chapter 3, these matrices are the most common metrics of protein similarity and are used widely by motif discovery and sequence alignment programs. The next section describes how these scoring matrices and sequence alignment programs can be co–opted for solving nontraditional bioinformatics problems such as handwriting and voice recognition. Finally, the last two sections are devoted to exercises in motif discovery that do not use grammatical methods, but instead rely heavily on classical machine learning techniques and simple statistical analyses.
4.2 Biogrep: a tool for matching regular expressions

4.2.1 Introduction

As more genomes are sequenced and annotated, increasing numbers of functional DNA and protein sequence motifs (or patterns) are being discovered. These motifs can be used to detect remote homologies that are missed by sequence alignment tools such as Blast \(^{10}\) and FastA \(^{194}\). Many databases such as Prosite \(^{118}\), PRINTS \(^{17}\), and BLOCKS \(^{108}\) contain collections of biologically significant patterns that are correlated with the function of protein families and are expressed as regular grammars, or equivalently, regular expressions (see Section 1.4 on page 42). For example, the Prosite motif \([AG]\ldots GK[ST]\) is indicative of ATP/GTP binding proteins.

Searching for such regular expressions can be an important part of sequence annotation. There are a variety of tools available for pattern–matching, the most common being the “grep” family of Unix tools, including a number of very fast and sophisticated variants such as agrep \(^{275}\) and NR–grep \(^{178}\). Also, there are many excellent bioinformatics–specific pattern–matching tools including Patscan \(^{71}\), tacg \(^{164}\), and fuzzpro \(^{206}\). However, all of these tools are optimized for searching for single patterns, that is, one–at–a–time.

Biogrep is a pattern–matching tool designed to match large pattern sets (100+ patterns) against large biosequence databases (100+ sequences) in a parallel fashion. This makes biogrep well–suited to annotating sets of sequences using biologically significant patterns.

4.2.2 Implementation and results

Biogrep is written in the C programming language using the GNU regular expression \(^{104}\) and POSIX threads (pthreads) \(^{173}\) libraries. The program reads query patterns from either a plain text file, one–per–line, or from a Teiresias–formated pattern file \(^{207}\) (see Section 1.5 on page 57). These patterns are treated as POSIX extended regular expressions and are searched against a user supplied file, which can be either a FastA–formatted biosequence database or any text file.

Table 4.1 on the facing page shows a comparison of Biogrep with a few common programs. The grep family of pattern matching tools are absent from the table because their run times are
extremely long. This is because many of these tools cannot take sets of patterns and have to be
used on a per pattern basis. The next best alternative to Biogrep is a simple PERL script split
between multiple processors.

**Table 4.1:** Performance of Biogrep matching all the 1333 patterns in Prosite (release 17.01)
against the 782370 protein sequences in Swiss–Prot/TrEMBL [22] (release as of 8 July 2002).
Runs were carried out on an IBM p670 eserver running AIX 5L with 8 Power4 processors.

<table>
<thead>
<tr>
<th>program</th>
<th># processors</th>
<th>execution time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>biogrep</td>
<td>1</td>
<td>8683</td>
</tr>
<tr>
<td>biogrep</td>
<td>2</td>
<td>4477</td>
</tr>
<tr>
<td>biogrep</td>
<td>4</td>
<td>2266</td>
</tr>
<tr>
<td>biogrep</td>
<td>6</td>
<td>1620</td>
</tr>
<tr>
<td>perl</td>
<td>1</td>
<td>11780</td>
</tr>
<tr>
<td>perl</td>
<td>6</td>
<td>1916</td>
</tr>
<tr>
<td>patscan</td>
<td>1</td>
<td>28466</td>
</tr>
</tbody>
</table>

Biogrep has a number of user options, which are described in the documentation that comes
with the software. Most importantly, Biogrep can divide the pattern–matching task between
a user–specified number of processors using threads. This drastically reduces the user–time
required to match large sets of patterns (see Table 4.1). In addition, Biogrep is distributed with
detailed documentation, numerous examples, and various helper–scripts for interfacing with
other pattern matching/discovery programs. The Biogrep source code is available at http://
web.mit.edu/bamel/biogrep.shtml.
4.3 The evolution of updated BLOSUM matrices and the Blocks database

4.3.1 Introduction

As I discussed in Chapter 3, amino acid substitution matrices are a very common way to measure the degree of similarity between protein sequences (see for example Section 3.5.1 on page 136). Indeed, the fidelity of amino acid sequence alignment and motif discovery tools 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.

This work is part of a manuscript that is currently under consideration. The manuscript was co-authored with Mark Styczynski, Isidore Rigoutsos, and Gregory Stephanopoulos. Throughout this section, the use of the pronoun “we” refers to these authors.
4.3.2 Motivation

Many different scoring matrices have been proposed in the literature, but the BLOSUM series \cite{109} and PAM series \cite{65} of matrices are by far the most widely used. For a review of the many different substitution matrices, the reader is referred to articles by Henikoff and Henikoff \cite{110,112} and Vogt et al. \cite{258}. 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 \cite{10} and FastA \cite{194} and multiple sequence alignment tools such as Clustal–W \cite{245} and t–coffee \cite{183}.

The BLOSUM series of matrices was constructed in 1992 from Blocks 5 \cite{109}: a database of protein blocks, or highly conserved protein regions, derived from families in the PROSITE database \cite{118}. 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 \cite{111,197} and other functions \cite{181,214}. 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 \cite{109}. 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 ma-
trices from successive releases of the Blocks database and describe the results of two sequence alignment tests used to evaluate the performance of these matrices.

4.3.3 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 reported elsewhere [129]. 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 reclustering percentages between 20 and 100 was performed (Figure 4-1). The matrix for each Blocks release with relative entropy closest to the originally reported BLOSUM matrix (0.6979) was selected as the representative matrix for that release.

Sequence datasets

Two different database searches were used to judge the ability of each matrix to detect homologs: a search of SWISS–PROT 22 [23] using a set of queries previously determined to reflect “difficult” searches that are able to distinguish the abilities of different matrices [110], and a search of the ASTRAL database [42] 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 [109, 110]. 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 the work by Henikoff and Henikoff [110]. In that work, the 257 PROSITE 9.0 [20] 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.
Figure 4.1: 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. 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 BLOSUM matrix. The right-most panel shows the number of blocks in each release of the Blocks database. 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.) 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.
The second method we used for testing matrices was designed to emulate the work by Price et al. [200]. We used the ASTRAL database [42] as the basis for our more exhaustive experiments for detection of remote homologs. ASTRAL is created based on the SCOP database [175], 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 [272] 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.

Search methods

We chose the Smith–Waterman [235] local alignment algorithm for all searches against both databases for its high sensitivity in detecting remote homologs. In particular, we used the ssearch implementation of the Smith–Waterman algorithm by Pearson [192, 193].

For our database 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 search are -7/-1, while in BLAST they are -11/-1). Moreover, previous work [100] 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.

Most importantly, the determination of completely optimized sets of matrices and parameters is not the ultimate goal of this work. Rather, the goal of this work is to analyze the BLOSUM matrices as affected by the changing entries in the Blocks database. In this sense, the use of globally optimal parameters for each matrix is not imperative; instead, 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. So, though we feel we chose acceptable parameters for our work, it is not of intrinsic importance to determine the optimal parameters for each matrix.

It is worth noting that other works (particularly the early BLOSUM works that the PROSITE–based testing method is based upon) frequently used BLAST instead of Smith–Waterman to evaluate the quality of scoring matrices. In this work, we chose Smith–Waterman because of its sensitivity and to avoid any artifacts due to the heuristic shortcuts in BLAST.

**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, as initially described by Pearson: 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 statisti-

cal significance of the mean difference in coverage between any two substitution matrices [200]. 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.

4.3.4 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 4-1.

The center of panel A lists the reclustering percentage needed for each Blocks release to produce a matrix with entropy closest to that of the original BLOSUM 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 [9]. Merely maintaining (in this case) a reclustering 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 [109]. (Note that, due to the bugfixes mentioned earlier, the BLOSUM matrix computed from Blocks 5 had its entropy analog at a reclustering 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 4-1 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 heatmap in panel A of Figure 4-1 reveals that, as expected, relative entropy
increases with increasing reclustering 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 4-1 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 reclustering 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 4-2 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 [109] 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 4-
3 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) 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 4-4 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 4-3. That is, releases 7, 8, 10, and 11 perform comparably to 5,
release 6 is slightly better, and release 12 is slightly worse, while releases 9, 13, and 14 perform
substantively worse than release 5. These latter releases have statistically significant differences.
This agreement suggests that the original test employed by Henikoff and Henikoff [109, 110]
was rather effective and efficient in that the results of the test would not have changed much
with access to today’s larger databases.
**Figure 4-2:** The relative performance of updated BLOSUM matrices. This figure is designed to emulate Figure 4 from Henikoff and Henikoff [109]. 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 in Henikoff and Henikoff [110], 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 4-3: A complete set of Bayesian bootstrap replicates, with inset histogram of coverage difference. These data were created using the PSCE software [200]. (See Price et al. [200] 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 analysis of the data via the inset histogram of coverage difference reveals that the difference in coverage clearly follows a distribution with non-zero mean. These distributions are used to compute the confidence intervals shown in Figure 4-4.
Figure 4-4: Plots of the differences in performance of updated RBLOSUM matrices. Each matrix is compared to the RBLOSUM64 matrix 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 4-2.
4.3.5 Discussion

The reason for the poor performance of RBLOSUM matrices derived from later releases of Blocks remains to be explained. Figure 4-1 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 4-1 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 reclustering 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 4-5. The cleaned version of the Blocks 14 database gives rise to an RBLOSUM matrix that is superior to any of the other matrices we tested, including RBLOSUM64 from Blocks 5 (Figure 4-5) and the original BLOSUM62 from Blocks 5 (data not shown).

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 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 [107]. 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
Figure 4-5: Coverage of a cleaned RBLOSUM matrix compared to the RBLOSUM64 matrix. 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 RBLOSUM64 matrix (and, transitively, all updated RBLOSUM matrices). 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 the different scale from Figure 4-4 and the statistical significance at all EPQ values since no error bar crosses the origin.
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 that Henikoff and Henikoff [109] initially proposed 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.
4.4 Bioinformatics and handwriting/speech recognition: unconventional applications of similarity search tools

4.4.1 Introduction

Bioinformatics has benefited immensely from tools and techniques imported from other disciplines. Markov models used for gene–finding have their origin in information science, neural networks are imported from machine learning, and the countless clustering methods used for analyzing microarray data are from a wide variety of fields.

Sequence alignment tools are no exception to this trend; however, within bioinformatics, they have reached new levels of speed and sophistication. Tools, such as Blast [10, 11] and FastA [194], are used routinely to search through a database for sequences (DNA or protein) that are similar to a query sequence. Over the years, these tools have been optimized for speed by employing a number of heuristic shortcuts to the dynamic programming algorithms on which they are based. Even searches in very large databases, such as Swiss–Prot/TrEMBL [22] or GenBank [32], take only a few seconds for queries of small to moderate size. This is substantially faster than the time required for a rigorous Smith–Waterman search [264]. In light of the remarkably speed and accuracy that characterize these algorithms, it is intriguing to investigate other applications where similarity search tools might be of material importance. In this work, I present two alternative applications of these fast sequence alignment tools outside the domain of bioinformatics: handwriting recognition and speech recognition. All of the work described in this section is part of a publication appearing in the proceedings of the fourth Singapore–MIT Alliance Programme on Molecular Engineering of Biological and Chemical Systems, which was co–authored with Gregory Stephanopoulos. Throughout this section, the use of the pronoun “we” refers to these authors.

The dynamic handwriting recognition problem is to recognize handwriting from a touch tablet as found on personal digital assistants (PDAs), for example Palm Pilots, or tablet PCs [242]. These writing tablets sample the position of a pen as a function of time to produce a series of \((x, y)\) points that are used by handwriting recognition algorithms to determine which character was written. An excellent review of the most common algorithms is available from Plamodon.
and Srihari, 2000. These include feature analysis, curve matching, Markov models, and elastic matching, the last of which is based on dynamic programming and is related to both Blast and FastA.

To apply similarity search concepts to the handwriting recognition problem, we represented the path of a PDA pen as a protein sequence by translating the \((x, y)\) points into a string of amino acids. Using the protein representation of handwriting samples, we were able to classify unknown samples with FastA. This is analogous to the problem of protein annotation using similarity searching: given a protein (a written character) of unknown function, we annotated the protein by searching for similar sequences (characters with similar \((x, y)\) paths).

We applied the same sequence alignment approach to speech recognition. Automated phone services, security checkpoints, and computer dictation software employ some form of speech recognition. Common speech recognition methods include feature recognition, neural networks, hidden Markov models, dynamic programming \([180]\) and a variety of other statistical and signal processing algorithms. A good review of these techniques and more is available from Juang & Furui, 2000. For this problem, we represented digital speech recordings as sequences of amino acids, and used a database of annotated recordings to classify unknown recordings.

In the following section, we describe the data sets used for the handwriting recognition and speech recognition problems. Then, we detail how these data were represented using strings of amino acids and how we used FastA to annotate unknown samples in four handwriting and speech recognition experiments. We compare our results to more traditional methods of handwriting and speech recognition and, finally, we discuss ways of improving upon the results and extending sequence alignment to other classification problems.

### 4.4.2 System and Methods

#### Handwriting Recognition

For our handwriting recognition experiments, we used data from Alimoglu and Alpaydin, 1996, available in the University of California Irvine repository of machine learning databases \([38]\). These data comprised of 10992 handwritten digits between 0 and 9, written by 44 writers with each writer submitting 250 digits (8 samples were discarded by the original authors).
Figure 4.6: Projection of a digit written with a PDA stylus into protein space. Concatenating the set of points gives a protein sequence representative of the digit. In this case, the sequence is QYKXVVFMWGSNHANQ. An alignment of nines from two different writers. The boxes at the top show the input from each writer and the large grid show the superposition of the two handwritten digits. The FastA alignment between the protein representations of the two digits is shown in the center. Two visualizations of the handwriting recognition problem. In both cases the x and y axes are divided into 23 parts corresponding to the columns and rows in an amino acid scoring matrix. The eight sampled points from the digit are cast from x, y space into protein space by assigning amino acid coordinates to each point.
Figure 4-7: An alignment of the spoken-letter “X” recorded from two different speakers. The plots at the top and bottom are recordings for first and second speakers, respectively. The breakout in the center shows a section of the protein projection of each recording and the alignment generated using FastA as described in the text. This example was taken from the first speech recognition experiment. In this case, the bottom recording was the top scoring alignment against the top recording.
Figure 4-8: A phylogenetic tree of voice–proteins. This tree was created using the Phylip [81] tree drawing program from a multiple sequence alignment of all 26 voice–proteins from a single speaker. The multiple sequence alignment was made using the ClustalW [116] alignment tool, with the scoring matrix in Table 4.3 on page 183. In the tree, similar sounding (homologous) letters are grouped near each other. For example, all the letters containing the /ee/ sound [B, C, D, E, G, P, T, V, Z] are clustered on the left side of the tree.
Table 4.2: Results for the handwriting and speech recognition problems described in the text. For each experiment, the misclassification is the percent of sequences in the unknown set for which the digit or letter was not predicted correctly.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Classification</th>
<th>Classification in Alimoglu &amp; Alpaydin, 1996</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>97.34%</td>
<td>97.80%</td>
</tr>
<tr>
<td>2</td>
<td>99.64%</td>
<td>n/a</td>
</tr>
</tbody>
</table>

(a) Handwriting recognition results.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Classification</th>
<th>Classification with clustering</th>
<th>Classification in Dietterich &amp; Bakiri, 1995</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93.84%</td>
<td>98.91%</td>
<td>96.73%</td>
</tr>
<tr>
<td>2</td>
<td>92.61%</td>
<td>98.61%</td>
<td>n/a</td>
</tr>
</tbody>
</table>

(b) Speech recognition results. The second column shows the misclassification using the clustering of all /ee/ sounding letters as described in the text.

Each digit was written with a stylus pen on a touch tablet, which recorded the $x$ and $y$ coordinates of the pen as a function of time. These data were re-sampled such that each written digit was represented by a series of eight $(x, y)$ points, spaced out by a constant arc length over the path of the digit. Then, for each digit, the set of $(x, y)$ points were scaled such that the largest axis, usually the $y$ axis, ranged from 0 to 1. By dividing the number line $[0, 1]$ into 23 “bins” we translated each of these coordinates into a pair of amino acids as shown in Figure 4-6 on page 179. We concatenated these amino acid pairs to obtain a protein sequence representation of each digit: a “digit–protein.”

Speech Recognition

For our speech recognition experiments, we used data from Deitterich and Bakiri, 1995, available in the University of California Irvine repository of machine learning databases [38]. This data set consisted of 7797 recordings of individuals speaking one of the letters $A–Z$. A total of 150 speakers each said every letter $A–Z$ twice (three recordings were discarded by the original authors). Then, each recording was processed into a set of 617 real–valued attributes in
This matrix was used in place of the default scoring matrix, Blosum. Amino acids are given "points", amino acids that are one off are given points, and so on. These were relatively insensitive to permutations about the form of the scoring matrix was found heuristically. Also, a few experiments indicated that the alignments are relatively insensitive to permutations about the form of $s_{ij}$ given above.

\[
A \quad R \quad N \quad D \quad C \quad Q \quad E \quad G \quad H \quad I \quad L \quad K \quad M \quad F \quad P \quad S \quad T \quad V \quad W \quad Y \quad Z \quad X
\]

|   | A | R | N | D | C | Q | E | G | H | I | L | K | M | F | P | S | T | V | W | Y | Z | X |
| A | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 | -11 | -12 |
| R | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 | -11 | -12 |
| N | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 | -11 | -12 |
| D | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 | -11 | -12 |
| C | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 | -11 |
| Q | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| E | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| Y | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| L | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| P | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| F | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| I | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| T | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| W | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| V | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| B | -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |
| X | -6 | -5 | -4 | -3 | -2 | -1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | -1 | -2 | -3 | -4 | -5 | -6 | -7 | -8 | -9 | -10 |

Table 4.3: The scoring matrix used for the handwriting and speech recognition FastA alignments. Each entry of the scoring matrix, $s_{ij}$, is given by $s_{ij} = 10 - (i - j)$. That is, matching amino acids are given 10 “points”, amino acids that are one off are given 9 points, and so on. This matrix was used in place of the default scoring matrix, Blosum$_{50}$ [109], for FastA. The scoring matrix was found heuristically. Also, a few experiments indicated that the alignments are relatively insensitive to permutations about the form of $s_{ij}$ given above.
the range \([-1, 1]\). A more detailed description of the database is available from Dietterich & Bakiri, 1995.

By dividing the number line \([-1, 1]\) into 23 bins we translated these real numbers into a series of amino acids. For example, the series \("-1.0, -0.5, 0.1, 0.65\)" was translated to “AQKY”. We concatenated these amino acids to make a protein representation of each recording: a “voice–protein”.

### 4.4.3 Results

**Handwriting Recognition**

We conducted two handwriting recognition experiments. In both experiments part of the digit–protein database was assumed to contain a “known” set of digits that was subsequently used to annotate, or classify, the remaining “unknown” digits. For our first experiment, we used for the known database containing the writing of 30 persons (7494 digits) and an unknown database with the writing of the remaining 14 persons (3498 digits). Using FastA, we searched each sequence from the unknown set in the known set and used the top scoring hits to annotate the unknown digits. Searches were carried out using the scoring matrix shown in Table 4.3 on the preceding page with FastA version 3.4.11 using the default gap open and extension penalties, and the following options: \(-p -Q -d0 -f8 -g1 -h -E1000 -b1\). An example alignment of two handwritten nines from different writers is shown in Figure 4-6 on page 179.

For our second experiment, we used 25% (2748 digits) of our digit–protein database, selected randomly, as the unknown set and the remaining 75% (8244 digits) as our known set. Alignments and annotations using FastA were performed as in the first experiment.

The results of the two handwriting recognition experiments are shown in Table 4.2 on page 182. In experiment 1, our results are about the same as the best k–means clustering results of Alimoglu and Alpaydin [6, 7]. This experiment simulates the user–independent handwriting recognition problem: the handwriting of one group of writers was used to classify digits from a different group. In the user–dependent problem, experiment 2, the database of known handwritten digits contains samples from all the writers, on average. Thus, for every unknown handwriting sample, there is often a close match in the database of known samples. As such,
the results of experiment 2 are significantly better than those of experiment 1 as shown in Table 4.2 on page 182.

In experiment 1, the average time for each alignment was 0.117 seconds per unknown sequence on a 1 gHz Pentium III processor. This is much shorter than the time required to write the digits. Thus sequence alignment could be used as a “real–time” method for handwriting recognition. This high speed, together with the high accuracy for user–dependent recognition makes sequence alignment good candidate for use on a Tablet PCs, or even PDAs.

Speech Recognition

Using the voice–protein database, we conducted two experiments, analogous to the two handwriting recognition experiments described previously. First, we used a known set consisting of 6238 recordings from 120 speakers and an unknown set with 1559 recordings from the remaining 30 speakers. Second, we used 25% (1949 recordings) of the voice–protein database, selected randomly, as the unknown set and the remaining 75% (5848 recordings) as the known set. Each of the speech recognition alignments was performed using the same scoring matrix and FastA parameters as the handwriting recognition experiments. An example alignment of two voice–proteins is shown in Figure 4–7 on page 180.

The results of the two speech recognition experiments are shown in Table 4.2 on page 182. Experiment 1 is compared to the best Error Correcting Output Code (ECOC) results of Deitterich and Bakiri [67], but there was no comparison available for experiment 2. The misclassification for experiment 1 was 6.16%, higher than the ECOC result of 3.27%. However, we observed that most of the errors were due to rhyming letters, and in particular all of the /ee/ sounding characters [B, C, D, E, G, P, T, V, Z]. This indicated that these characters were similar on a sequence level, so we constructed a phylogenetic tree of the sequences to study their relationship.

A phylogenetic tree of 26 voice–proteins from a single speaker is shown in Figure 4–8 on page 181. As the figure shows, the protein projections of phonetically similar letters tend to be homologous. Furthermore, letters such as A and H, which have the /ay/ sound at the beginning, are more closely related to each other than they are to J and K, which have the /ay/ sound at the end. Because the /ee/ sounding letters all have /ee/ at the end, they are particularly difficult to
distinguish from each other. These letters account for a disproportionate majority of the errors in our two experiments. By clustering these letters together such that they are considered the same for classification purposes, the error in experiment 1 was reduced to 1.09%. If the original error was evenly distributed between the classes, the error would have been reduced only to about 5.5%. This suggests that, although string alignment performs poorly for /ee/ sounding characters, it performs well for all other characters.

4.4.4 Conclusions

This work showed that sequence alignment can be a powerful classification tool for problems outside the domain of bioinformatics. In both the handwriting and speech recognition problems, we projected real–valued data into strings of amino acids and used FastA as a classification tool, in a manner analogous to protein annotation. In the case of handwriting recognition, we showed that sequence alignment is a viable alternative to traditional methods, such as k–means clustering, and is fast enough to be used as a real–time recognition method.

There are many ways to improve upon the results we presented here. First, we did not have any explicit training phase for either set of experiments. However, there are at least two sequence alignment parameters which can be trained: the gap open and extension penalties, and the scoring matrix. The optimization of these parameters for protein annotation is well documented \[9, 65, 109, 110, 112, 258\] and would be similar for alternative sequence alignment applications such as handwriting recognition. Second, intelligent projection of data into strings can greatly improve results. Here, we used bins of equal size to partition the real–valued data into amino acids; however, bins of unequal size may improve the resolution between closely related sequences and improve classification. Finally, more customizable sequence alignment tools would be very useful. These tools should take an arbitrary alphabet (Blast and FastA are restricted to 23 amino acids) and a user–defined scoring matrix (FastA allows user–defined matrices, but Blast does not).

The potential applications of sequence alignment tools outside of bioinformatics are boundless. Tools such as Blast and FastA can be used to quickly classify or search through any data that can be projected into a string of characters. Of course, these methods will work best with data that is of a low dimension. Our experiments with more complex data data, such as color
images, suggest that how the data are projected into a string is very important with large number of dimensions. However, for simple types of data, such as customer purchase histories, black and white images, or Internet chat transcripts, we have been able to use sequence alignment as a quick and effective classification tool.
4.5 Machine learning approaches to modeling the physiochemical properties of peptides

4.5.1 Introduction

In this section, I discuss the modeling of small peptide sequences using non-grammatical models. Most commonly, peptides and protein sequences are represented as a string of letters drawn from the alphabet of characters representing the twenty natural amino acids. Here, I present a series of experiments using a more meaningful representation of amino acids and test the ability of various machine learning techniques to predict peptide function. Specifically, I develop a set of three amino acid representation schemes and test these schemes combinatorially with a set of six machine learning techniques. All of the work described in this section is part of a publication appearing in the proceedings of the fourth Singapore–MIT Alliance Programme on Molecular Engineering of Biological and Chemical Systems, which was co-authored with Mark Styczynski and Gregory Stephanopoulos. Throughout this section, the use of the pronoun “we” refers to these authors.

4.5.2 Motivation and background

Amino acid representations

The most common representation of small peptides are as strings of letters representing the twenty amino acids, e.g. KWRAG, which is the five residue sequence lysine, tryptophan, arginine, alanine, and glycine. Notably, both amino acid names and their corresponding abbreviations are human constructs that carry no information about the underlying physiochemical characteristics of each amino acid. That is, the string KWRAG carries little information in and of itself, without some information about what a K is and how it is different from the other amino acids. In place of such physical descriptions, previous efforts have described the similarity of amino acids based on the tendency for one amino acid to substitute for another in homologous, similarly-functioning proteins across different species [65, 109]. That is, substitutions that are observed in nature can be construed in some sense as indicating similarity between certain amino acids. While such efforts have been extremely useful for tasks such as aligning more
distant protein homologs, they typically do not capture enough information to be practically useful in *de novo* design or prediction of protein activity.

Here we experiment with feature vector representations of small peptides using sets of amino acid physiochemical characteristics derived from the AAindex database [135, 176, 247]. The AAindex database lists 453 physiochemical parameters for each of the twenty amino acids. These parameters range from those that are very tangible and intuitive — for example, residue volume, which is AAindex parameter BIGC17017 [36] — to the abstract — for example, the normalized frequency of participation in an N-terminal beta–sheet, which is AAindex parameter CHOP170208 [57]. The parameters were culled from the scientific literature by the AAindex authors and might be considered the universe of what we, as the scientific community, know about each amino acid.

Thus, a very logical way of representing an amino acid is as a feature vector of these 453 attributes. In this sense each type of amino acid has a different feature vector of the same dimensionality. This might be considered the “maximally informative” representation of the amino acids since it incorporates an expansive set of features culled from the literature. Extending this, we could write an amino acid sequence as the concatenation of these vectors. That is, a three residue peptide could be represented as a $3 \times 453 = 1359$ feature vector. Intuitively, this representation retains more information than the string representation. Further, we would imagine that the physiochemical representation would be more useful for modeling the function of a peptide sequence, such as its propensity to fold in a certain manner or to react with a certain enzyme.

The representation of amino acids has received some previous attention in the literature. For example, Atchley et. al. [16] use the physiochemical parameters from the AAindex to create a low–dimensional projection of the characteristics of each of the twenty natural amino acids. Further, they used this low–dimensional progression to derive metrics of similarity between the amino acids, similar to popular amino acid scoring matrices such as Blosum [109] and PAM [65].
HIV–I Protease

In this work we will use the HIV–I protease as a model system for demonstrating the merits of different physiochemical amino acid representations. Specifically, we show the success of different representations and different machine learning methods at modeling substrate specificity of the protease.

The HIV–I protease is a proteolytic enzyme encoded by the HIV genome [44]. The protease plays a critical role in viral replication and the development of viral structure [262]. The protease recognizes specific eight–residue sequences in its substrates (see Figures 4–9 and 4–10 on the facing page). The protease's natural targets are subsequences of other HIV genes which must be cleaved for the virus to successfully replicate. Accordingly, small molecule inhibitors of the protease are a common therapy for HIV/AIDS [39].

![Structure of the HIV–I protease](image)

**Figure 4-9:** Structure of the HIV–I protease, derived from the Protein Data Bank (PDB) [34] entry 7HVP [241]. Over one hundred other structures of the protease have been solved since the first in 1989 and are available from the PDB's website. The protein is a dimer of two 99 amino acid chains. The regions of the protein at the top of the figure, the “flaps,” open up and accept a substrate protein, closing behind it. Two aspartate residues in the active site, aided by the presence of water, act to cleave the substrate.

In addition to the handful of sites that the protease cleaves to facilitate viral development, it can cleave a number of other “non–natural” substrates [28]. These substrates have been the focus of intense experimental study [18, 26, 27, 60]. In a recent manuscript, You et. al. collected a comprehensive set of 700+ eight–residue substrates that have been tested for cleavability by the HIV–I protease [276]. In addition, You et. al. developed a series of models for the protease's substrate selectivity that, in general, outperform previous computational models [48, 56, 177, 211], which relied on a much smaller dataset [49].
Figure 4-10: Schematic of the HIV–I protease active site. The active site comprises eight binding pockets (P1–P4 and P1′–P4′) into which eight residues from the target protein fall. The target protein is cleaved between the S1 and S1′ residues. One half of the catalytic unit is made up by chain A of the protease and the other by chain B (see Figure 4-9 on the facing page).

4.5.3 Methods

Amino acid representations and input data set

A set of 746 eight–residue peptides were generously provided by You et. al. [276], each with a class: cleavable by the HIV–I protease or not cleavable. In addition, the complete set of 453 physiochemical parameters for each of the 20 naturally occurring amino acids was downloaded from the AAindex database (release 7.0, July 2005).

From these 453 parameters, we removed redundant parameters for which the magnitude of the correlation coefficient with another parameter was greater than 0.80. The remaining 155 independent parameters were kept. Using these parameters, we made three different projections of the 746 experimentally tested protease substrates as detailed below.

Full physiochemical projection In this projection each eight–residue peptide was represented as a 1241–dimensional feature vector: 8 residues with 155 physiochemical features per residue plus the class — cleaved or not cleaved. Of our three representations, this one retains
the most information about the peptides.

**Feature–selected physiochemical projection**

Using the “FULL” projection (above) we performed a feature selection routine to select only those features that are most correlated to the class. (Throughout this manuscript, all modeling and feature selection were performed using the Waikato Environment for Knowledge Analysis, or WEKA [269]). Briefly, we evaluated the worth of a subset of features by considering the individual predictive ability of each feature with respect to the cleaved/uncleaved class, along with the degree of redundancy between the features. Using this method, we created a 54–dimensional projection of the peptide substrates (53 features plus the class).

Analysis of this lower–dimensional projection revealed that the features of the outer residues (S₄, S₄') are relatively unimportant, whereas the central residues (S₁, S₁') are quite important in determining cleavability. For the S₁ position, seven parameters were chosen:

- FASG₇₆₀₁₀₂: Melting point [79];
- FAUJ₈₈₀₁₀₅: Minimum width of the side chain [80];
- PALJ₈₁₀₁₁₁: Normalized frequency of beta–sheet in alpha+beta class [188];
- PRAM₉₀₀₁₀₁: Hydrophobicity [198];
- ROBB₇₆₀₁₀₇: Information measure for extended without H-bond [210];
- KOEP₉₉₀₁₀₁: Alpha–helix propensity derived from designed sequences [143]; and
- MITS₀₂₀₁₀₁: Amphiphilicity index [171].

**PCA projection of physiochemical properties**

Using the full, 155–dimensional representation of each of the 20 naturally occurring amino acids, we performed principal component analysis (PCA) to find linear combinations of features that capture the variation between different kinds of amino acids. More formally, PCA, or the Karhunen–Loève transform, is a linear transformation by which the 20 data points in a 155–dimensional space are projected onto a new coordinate system. The system is chosen such that the greatest variance is captured by the first axis, or the first “principal component.” Successive principal components (axes)
capture progressively less variance. Each component is a linear combination of some of the initial features; given appropriate uniform normalization, the weight of each feature in a given component indicates the relative importance of that feature in defining the component.

Using PCA, we derived 16 principal components that capture 95% of the variance in the amino acids, with the first PC capturing 30% of the variance. The set of 746 peptide 8–mers were projected into a reduced 129–dimensional space: 8 concatenated 16–dimensional residues plus the class of the peptide.

**Model creation and classification**

For each of the three peptide representations detailed above, we tested the ability of six machine learning techniques to classify the peptides as either cleaved or uncleaved. Each of these models is described below. For each model, we evaluated the performance using 10x10 cross–validation (see Conclusion): for each of ten runs, 10% of the peptide dataset was withheld for testing a classifier trained by the remaining 90% of the peptides. The sensitivity and specificity of each classifier’s predictions for all ten of its cross–validation runs can then be combined to determine the percentage of correctly classified peptides. This value is used to quantify the classifier’s overall accuracy and facilitates pairwise comparison of models and representation schemes.

**Decision tree model**  Decision trees are simple, intuitive classification schemes that use a series of questions (decisions) to place a sample in a class with low error rate. More specifically, a decision tree is a structure in which the internal branches represent conditions, such as “hydrophobicity index at S > 0.52”. Following these conditions leads to the leaves of the tree, which are classifications indicating whether the peptide is cleaved or not. Here, we use a particular variant of the decision tree, a C4.5 decision tree [203], which is notable for not being prone to overfitting of input data. An example decision tree from our experiments is shown in Figure 4-11 on page 198.

**Logistic regression model**  A logistic regression is just a non–linear transformation of a linear regression. In this model, each independent variable (the different dimensions of our various projections) are regressed to the class (cleaved or not cleaved). Here we use a variant of logistic regression that leads to automated feature selection and is described elsewhere [146].
Bayesian network model  Bayesian network models use directed acyclic graphs to model the joint probability distribution of each class over all input features. That is, the model captures conditional dependencies between the features with regards to how they impact the final classification of each sample. Bayesian networks can be used to find causality relationships, one of many features that make these models particularly well-suited to many applications in computational biology (see, for example, [85, 105, 221]). The method uses a Bayesian scoring metric that ranks multiple models based on their ability to explain data with the simplest possible method. The Bayesian metric is a function of the probability of the model being correct given a set of observed data; this is, in turn, correlated to the model’s prior probability and its physical likelihood. For a more detailed explanation of Bayesian networks, see Witten and Frank [269] or Heckerman [106].

Naive Bayes model  The naive Bayes model, or “Idiot’s” Bayes model [103], is a simple machine learning scheme that assumes naively that each feature has an independent effect on the classification of each sample [130]. In the case of the HIV–I protease substrates, this means that the physiochemical characteristics of the S1 residue contribute to the cleavability of the peptide in a way that is independent of the other residues: S1, S2, etc. The resulting network dependencies are less complex than one might otherwise obtain from a Bayesian network model but are frequently useful, particularly for unwieldy datasets or problems with physical characteristics that may warrant the assumption of conditional independence of features.

Support vector machine model with linear basis function  The support vector machine (SVM) is a machine learning technique posed as a quadratic programming (QP) problem [31]. The formulation can best be conceptualized by considering the problem of classifying two linearly separable groups of points. The first step is to define the “convex hull” of each group, which is the smallest–area convex polygon that completely contains a group. The SVM approach looks for the best linear classifier (single straight line) between the two groups of points, defined as either the line that bisects the two closest points on each convex hull or the two parallel planes tangent to each convex hull that are furthest apart. These alternative definitions provide two alternative formulations of a convex QP problem; notably, they both reduce to the same problem. (A rigorous mathematical treatment of these qualitative explanations can
be found elsewhere [30, 62].) Tried and true methods for solving QP problems can then be used to (relatively quickly) determine the best classifier. This method can be expanded to allow for linearly inseparable cases by altering the optimization problem to account for a weighted cost of misclassification when training the model. There is evidence in the literature that an SVM approach to defining the best classifier is less susceptible to overfitting and generalization error [63, 254, 255].

**Support vector machine model with radial basis function**  The above description of an SVM, despite accounting for the possibility of inseparability, does not address the need for non-linear classifiers. For instance, if the members of one class fall within a well-defined circle and the non-members fall outside of the circle, the above method will perform extremely poorly because it will try to form just one plane to separate the groups [31]. Rather than attempting to fit higher-order curves, it is easier to project the input attributes into a higher-dimensional space in which the groups are (approximately) linearly separable. The higher-dimensional spaces can be characteristic of any desired classifier (e.g., nonlinear terms generated by multiplying attributes or squaring attributes). The same method for computing the best linear classifier is then used. The result is mapped back into attribute space of the appropriate dimensions and constitutes a non-linear classifier. Though one may expect such a process to be prohibitively expensive for data with many attributes, there exists a computational shortcut using “kernel functions” to avoid calculating all possible higher-dimensional feature values. In this work, the basis function for the kernel gives us the ability to detect optimal classifiers that are based upon training points’ radius from some center point (as in the above example).

### 4.5.4 Conclusion

Our results show that the full, 1241-dimensional representation performed the best, followed by the PCA representation and, finally, the representation made via feature selection. (See Figure 4-12 on page 199 and Table 4.6 on page 197 & 4.7 on page 197. In these tables “FULL” is the full physiochemical, 1241-dimensional representation; “CFS” is the feature-selected, 55-dimensional representation; and “PCA” is the 129-dimensional representation created using principal component analysis.)
Table 4.4: Machine learning model comparison. Each \(i,j\) entry represents the number of representations, out of three, for which the \(i\) model performed \textit{worse} than the \(j\) model. Here “worse” means that the model had a statistically significant lower performance, based on a two-tailed \(t\)-test at the 0.05 confidence level.

<table>
<thead>
<tr>
<th></th>
<th>DT</th>
<th>LR</th>
<th>NB</th>
<th>BN</th>
<th>SVM</th>
<th>SVM–rbf</th>
</tr>
</thead>
<tbody>
<tr>
<td>DT</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>LR</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NB</td>
<td>0</td>
<td>3</td>
<td>-</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>BN</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SVM</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>SVM–rbf</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

Of the models tested, results show that logistic regression is the best, followed by (linear basis function) SVMs and Bayesian networks (See Figure 4-12 on page 199 and Table 4.4 & 4.5 on the next page.) The single best model/representation combination was the SVM model with radial basis function (SVM–rbf) and the FULL representation. It is worth noting that though this single combination was the best, the radial basis function SVM itself did not perform consistently well. Though this may not have been expected, it is definitely reasonable per the “No Free Lunch” theorem: no single machine–learning method should be expected to perform the best in all cases [271].

In general, these results suggest that higher–dimensional physiochemical representations tend to have better performance than representations incorporating fewer dimensions selected on the basis of high information content. As such, it seems that as long as the training set is a reasonable size, more accurate classifiers can be constructed by keeping as many significant input attributes as possible. Though methods like principal components analysis help to reduce computational complexity for unwieldy datasets, it is better to avoid feature selection until a supervised method (like the models tested in this work) can determine which features are most important in classifying samples.
4.5. MACHINE LEARNING APPROACHES TO MODELING THE PHYSIOCHEMICAL PROPERTIES OF PEPTIDES

Table 4.5: Machine learning model ranking. Each row shows, for each model, how many other model/representation pairs that model (with any representation) “wins” against. (Thus, the max of the sum of the columns in any row is $18 - 3 = 15$; however, ties are not shown.) Here “win/loss” means that the model had a statistically significant higher/lower performance, based on a two–tailed t–test at the 0.05 confidence level.

<table>
<thead>
<tr>
<th></th>
<th>total wins</th>
<th>total losses</th>
<th>model</th>
</tr>
</thead>
<tbody>
<tr>
<td>LR</td>
<td>8</td>
<td>0</td>
<td>LR</td>
</tr>
<tr>
<td>SVM</td>
<td>7</td>
<td>1</td>
<td>SVM</td>
</tr>
<tr>
<td>BN</td>
<td>5</td>
<td>3</td>
<td>BN</td>
</tr>
<tr>
<td>SVM–rbf</td>
<td>5</td>
<td>5</td>
<td>SVM–rbf</td>
</tr>
<tr>
<td>NB</td>
<td>1</td>
<td>7</td>
<td>NB</td>
</tr>
<tr>
<td>DT</td>
<td>0</td>
<td>10</td>
<td>DT</td>
</tr>
</tbody>
</table>

Table 4.6: Machine learning representation comparison. Each $i, j$ entry represents the number of models, out of six, for which the $i$ representation performed worse than the $j$ representation. Here “worse” means that the representation had a statistically significant lower performance, based on a two–tailed t–test at the 0.05 confidence level.

<table>
<thead>
<tr>
<th></th>
<th>FULL</th>
<th>CFS</th>
<th>PCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>FULL</td>
<td>-</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CFS</td>
<td>3</td>
<td>-</td>
<td>4</td>
</tr>
<tr>
<td>PCA</td>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.7: Machine learning representation ranking. Each row shows, for each representation, how many other model/representation pairs that representation (with any model) “wins” against. (Thus, the max of the sum of the columns in any row is $18 - 6 = 12$; however, ties are not shown.) Here “win/loss” means that the representation had a statistically significant higher/lower performance, based on a two–tailed t–test at the 0.05 confidence level.

<table>
<thead>
<tr>
<th></th>
<th>FULL</th>
<th>PCA</th>
<th>CFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 4-11: The decision tree calculated for the CFS, a 54–dimensional representation of the 8–mer peptides. The branch points are in the form PARAMETER_RESIDUE. For example, \text{CHOP780207}_S2' represents the AAindex parameter CHOP780207 (normalized frequency of participation in a C–terminal non–helical region) at the S2' residue. Values for all AAindex parameters are normalized to 1 across all amino acids. The tree shows various questions about a peptide that, when followed, lead to a set of conclusions. For example, if a given peptide has \text{CHOP780207}_S2' \leq 0.41765 and \text{FAUJ880105}_S1 > 0.57778 and \text{QIAN880137}_S3 > 0 then the peptide is classified as uncleaved. As shown in the table, 37 of the 746 known peptides are correctly classified by this scheme and only one is incorrectly classified.
4.5. MACHINE LEARNING APPROACHES TO MODELING THE PHYSIOCHEMICAL PROPERTIES OF PEPTIDES

Figure 4-12: Classification results for all amino acid representations and model types. The three different amino acid representations are shown in shades of gray: “FULL” is the full physicochemical, 1241-dimensional representation; “CFS” is the feature-selected, 55-dimensional representation; and “PCA” is the 129-dimensional representation using created using principle component analysis (see text). Error bars show the standard deviation over the 10x10 cross-validation test (100 samples per representation/model combination with a total of 1800 tests.) The best performing model was the SVM with radial basis function (SVM–rbf in the figure) with the full 1241-dimensional feature vector representing each eight-residue sequence. Averaged over all representations, the logistic regression model is best (see Table 4.4 on page 196). The poorest performing model is the decision tree (DT) with the 129-dimensional feature vector created using the PCA projections created as described in the text. In general the full 1241-dimensional representation performed the best, followed by the PCA representation and finally the CFS representation, which was created by a feature selection process.
4.6 Identifying functionally important mutations from phenotypically diverse sequence data

4.6.1 Introduction

In the previous section, I departed from the use of grammar–based models of sequences and explored statistical modeling approaches. This section continues this line of work, but is focused on the identification of important mutations in nucleotide sequences, rather than global, physiochemical characteristics of small peptides. In particular, in this section I present a simple statistical method for parsing out the phenotypic contribution of a single mutation from libraries of functional diversity that contain a multitude of mutations and varied phenotypes. This work is part of a publication that is in press at Applied and Environmental Microbiology, which was co–authored with Hal Alper, Curt Fischer, and Gregory Stephanopoulos. Throughout this section, the use of the pronoun “we” refers to these authors.

4.6.2 Motivation

The engineering of functional nucleic acid sequences and other biomolecules is frequentlyhampered by a limited understanding of how specific mutations at a genotype level are manifested in the phenotype. For some well–studied, large protein families, these relationships can be inferred; however, such cases are rare. In the absence of these relationships, we resort to strategies that explore the genotype space in a random manner, such as directed evolution.

In many cases, directed evolution of genes and other functional DNA loci is an effective approach to sample the sequence space in search of biomolecules with desirable properties [98, 236]. However, the most successful examples employ a selectable fitness criterion that allows for high-throughput screening of the mutational space: sampling a large enough space eliminates the need to make rational mutations. For many proteins or functional nucleic acids, it may not be possible to link a desired phenotype with a selectable criterion, fit for high-throughput screening. In the absence of such a criterion, clonal populations of mutants must be assayed individually for the phenotype of interest. This scenario might be called “assay–based” directed evolution, a situation in which the upstream mutagenesis has a higher
throughput than the downstream characterization. In this scenario, there is a premium on information linking mutational changes to their phenotypic manifestations. Further, there is a strong incentive to “learn from” the (relatively small) mutational spectra of these mutants to determine sequence-phenotype interactions, and to use this information rationally in subsequent rounds of mutagenesis.

Here, we present a simple statistical method for analyzing a mutational spectrum to parse out the phenotypic manifestation of individual mutations, even when they are masked by the presence of many other mutations. Because assay-based directed evolution does not employ any pre-screening or selection of clones, as is the case when a selectable marker is available, mutants are expected to have a range of phenotypes, including both increased and decreased fitness. Here, we demonstrate our method by identifying mutations in a library of mutagenized PL–λ promoters [8] that result in either increased or decreased promoter activity and we show how to quantify the statistical confidence in these mutation-phenotype linkages.

The central premise of our method is that mutations that have no effect on mutant phenotype should partition randomly, following a multinomial distribution, between phenotypic classes. For example, consider a hypothetical experiment in which we mutagenize a protein that can fluoresce in one of three colors: red, blue, or green. After generating a library of 1000 mutants, each bearing many point mutations, our assay reveals that 600 have the red phenotype, 300 are blue, and 100 are green. If a particular point mutation has no effect on the color, then we expect that, by chance, mutants containing this modification will be distributed between the red, blue, and green classes in a ratio of 6:3:1. That is, the mutation should not be correlated to any particular phenotypic class. More rigorously, we say that the mutations are multinomially distributed between the three classes with background frequencies 0.6, 0.3, and 0.1.

Multinomial statistics and related combinatorial statistics commonly arise in the analysis of naturally–occurring mutational diversity [2, 196]. For example, similar statistical analyses have been used to find functional gene domains [157], important structural RNA sites [131], and genomic loci with an overabundance of single nucleotide polymorphisms (SNPs) [259]. Here we apply multinomial statistics to the analysis of an artificially generated mutational landscape to parse out critical residues controlling phenotypic behavior. We show that, based on this information, mutants with sets of individual mutations can be made, and we suggest that this can
be used as a method for improving directed evolution experiments by incorporating sequence information.

In what follows, we detail the construction of numerous PL–λ promoter variants, which were generated by error–prone PCR such that each mutant incorporated many point mutations. The activity of these promoters was assayed using flow cytometry to measure the fluorescence of a GFP reporter gene. We show how our statistical analysis revealed the phenotypic manifestation of numerous mutations. Finally, we present a validation of our method by constructing point mutations for several of the identified mutations and combinations of sites using site-directed mutagenesis. These mutations, we show, have the predicted effect on the promoter phenotype, even when removed from the background of other mutations.

4.6.3 Materials and Methods

Strains and Media

E. coli DH5α (Invitrogen) was used for routine transformations as described in the protocol. Assay strains were grown at 37°C with 225 RPM orbital shaking in M9-minimal media containing 5 g/L D-glucose and supplemented with 0.1% casamino acids. All other strains and propagations were cultured at 37°C in LB media. Media was supplemented with 68 µg/ml chloramphenicol. All PCR products and restriction enzymes were purchased from New England Biolabs and utilized Taq polymerase. M9 Minimal salts were purchased from US Biological and all remaining chemicals were from Sigma-Aldrich.

Library Construction

Nucleotide analogue mutagenesis was carried out in the presence of 20 µM 8–oxo–2′–deoxyguanosine (8–oxo–dGTP) and 6–(2–deoxy–β–D–ribofuranosyl)–3,4–dihydro–8H–pyrimido–[4,5–c][1,2]oxazin–7–one (dPTP) (TriLink Biotech), using plasmid pZE–gfp(ASV) kindly provided by M. Elowitz as template along with the primers PL_sense_AatII, TCCGACGTCTAAGAAACCAT-TATTATC and PL_anti_EcoRI, CCGGAATTCGTCAGTGCGTCCTGCTGAT. Ten and 30 amplification cycles with the primers mentioned above were performed. The 151 bp PCR products were purified using the GeneClean Spin Kit (Qbiogene). Following digestion with
AatII and EcoRI, the product was ligated overnight at 16°C and transformed into library efficiency *E. coli* DH5α (Invitrogen). About 30,000 colonies were screened by eye from minimal media–casamino acid agar plates and 200 colonies, spanning a wide range in fluorescent intensity, were picked from each plate. Selected mutants were sequenced using primers PL_Sense_Seq,

AGATCCTTGGGGCAAGAAA

and PL_Anti_Seq,

GCCATGGAACAGGTAGTTTTCCAG.

### Library Characterization

About 20 µL of overnight cultures of library clones growing in LB broth were used to inoculate 5mL M9G medium supplemented with 0.1% w/v casamino acid (M9G/CAA). The cultures were grown at 37°C with orbital shaking. After 14 h, roughly the point of glucose depletion, a culture sample was centrifuged at 18,000 g for 2 minutes, and the cells were resuspended in ice–cold water. Flow cytometry was performed on a Becton–Dickinson FACScan as described elsewhere [8], and the geometric mean of the fluorescence distribution of each clonal population was calculated.

Mean and standard deviation were calculated from the FL1–H distribution resulting after gating the cells based on a FSC–SSC plot. A total of 200,000 events were counted to gain statistical confidence in the results.

### Construction of designed promoters

Promoters with specific nucleotide changes were created using overlap–extension PCR and primers specifically designed to incorporate these changes. Primers were designed to divide the promoter region into thirds, and the proper primers were assembled piecewise in a PCR reaction consisting of 95°C for 4 minutes, 10 cycles with an annealing temperature of 44°C, followed by 30 cycles of PCR with an annealing temperature of 60°C, and a final extension for 3 minutes at 72°C. Fragments were gel extracted using 2.5% agarose gels and Qiagen MERmaid spin kit. The isolated fragment was then linked with the final primer using the same
PCR and extraction procedures. Finalized fragments were then digested using EcoRI and AatII and ligated into the digested plasmid backbone. Sequencing was performed to verify correct constructs.

4.6.4 Results

Generation of mutant library

Previously, we reported on the development of a promoter library generated through the random mutagenesis of the sequence space [8]. In that work, library diversity was created through error–prone PCR of the PL–TET01 promoter, a variant of the PL–λ promoter [59], which was placed upstream of a gfp gene. The promoter region contains two tandem promoters PL–1 and PL–2, each of which contains -10 and -35 sigma factor binding sites [95–97]. Furthermore, the promoter contains, at approximately the same location, an UP element that binds C–terminal domain of the alpha subunit and a binding site for integration host factor (IHF). In addition, the PL–TET01 promoter has two tetO2 operators from the Tn10 tetracycline resistance operon [158].

Mutants in the library were analyzed using flow–cytometry to measure the single–cell level of expression of GFP as a proxy for the activity of the mutagenized promoters. (A detailed schematic of the experimental procedure is shown in Figure 4-14 on page 206.) Promoters that had roughly log–normal fluorescence distributions (no obvious tails in the distribution or bimodal distributions) were sequenced and, from that set, those mutants that contained deletions or insertions were removed. The final set comprised 69 mutant promoters, with well–behaved fluorescence distributions (single distribution with a low standard deviation), that only contained transition and transversion mutations. Notably, our error–prone PCR method introduces predominantly transitions and not transversions, except in rare cases.

Identification of critical sites

Returning to the red, blue, green example introduced earlier, each of these N hypothetical mutants can be classified into one of M mutually–exclusive and collectively–exhaustive phenotypic classes — \( P_1, P_2, \ldots, P_M \) — such that there are \( n_1, n_2, \ldots, n_M \), mutants in each class and
Figure 4.13: Structure of the PL–TET01 promoter. There are numerous functional sites on the PL–TET01 promoter that are known to effect the rate of complex formation between the promoter and RNA polymerase [95–97]. The promoter region contains two tandem promoters PL–1 and PL–2, each of which contain -10 and -35 sigma factor binding sites. Furthermore, the promoter contains, at approximately the same location, an UP element that binds C-terminal domain of the alpha subunit and a binding site for integration host factor (IHF) a global regulator of gene expression in E. coli. The IHF site acts to bend the promoter region, bringing the alpha–CTD binding site in sufficient proximity to the beta subunit of the RNA polymerase. In addition, the PL–TET01 promoter has two tetO2 operators from the Tn10 tetracycline resistance operon [158].
Figure 4.14: Schematic of the experimental procedure. A variant of the constitutive bacteriophage PL–λ promoter (PL–TETO1) was mutated through error–prone PCR to create mutated fragments of promoters. These fragments were then ligated into plasmid constructs and used to drive the expression of gfp in E.coli. These cells were then analyzed using flow cytometry to quantify the fluorescence of GFP and output capacity of the promoter.
Consider a subset of mutants $B$ of size $X$, where $X < N$, comprising mutants with a particular mutation. If the mutation does not influence the phenotype of the mutants, we would expect, by chance, that there would be $x_i = X(n_i/N)$ mutants of type $P_i$. In general, the probability that the set $x_1, x_2, \ldots, x_M$ will take on the particular set of values $y_1, y_2, \ldots, y_M$ is

$$
Pr(x_1 = y_1, x_2 = y_2, \ldots, x_M = y_M) = \binom{X}{y_1, y_2, \ldots, y_M} \prod_{i=1}^{M} \frac{n_i}{N}
$$

where $\sum y_i = X$. In this equation, the term

$$
\binom{X}{y_1, y_2, \ldots, y_M} = \frac{X!}{y_1! y_2! \ldots y_M!}
$$

is the so-called multinomial coefficient, which can be equivalently written

$$
Pr(x_1 = y_1, x_2 = y_2, \ldots, x_M = y_M) = \frac{X!}{y_1! y_2! \ldots y_M!}
$$

The coefficient is the number of ways sets of size $y_1, y_2, \ldots, y_M$ could be chosen from a set of size $X$. (For example, in the case $X = 6, M = 2, y_1 = y_2 = 3$, the coefficient is 20 because there are 20 different ways to choose two subsets of size three from a set of six.)

The probability that $q$ or more (where $q < X$) of the $B$ mutants would be seen in a particular class, $P_i$, by chance is

$$
Pr(x_i \geq q) = \sum_{k=q}^{X} \binom{X}{q} \left( \frac{n_i}{N} \right)^k \left( 1 - \frac{n_i}{N} \right)^{N-k}
$$

Equivalently, this is the $p$–value for seeing $q$ of the $B$ mutants in class $P_i$. The lower the $p$–value, the more confident we are that the $B$ mutation is correlated with the $P_i$ phenotype.

For this study, we divided the mutants into two phenotypic classes based on their fluorescence (i.e. $M = 2$): the top 50th percentile and the lower 50th percentile. Figure 4-15 on page 209 shows a detailed schematic of the statistical analysis, which is greatly simplified in this case because there are only two phenotypes. As shown in the figure, applying our statistical method to the sequence data resulted in the identification of seven nucleotide positions that are correlated with one of the two phenotypic classes in a statistically significant manner. The figure
should be read clockwise from the top–left, progressively showing the fluorescence distribution, mutation distribution, statistical distribution of mutations, and finally, the identified important positions in part D in the lower left. In quadrant A, the vertical axis shows the mutant number, where the mutants are sorted in descending order by their relative fluorescence. In general, the single-cell fluorescence distribution for each mutant strain was log-normal distributed. The horizontal axis shows the mean of the log relative fluorescence for each mutant strain, where the error is the standard deviation of this distribution. Reading to the right from quadrant A into quadrant B reveals the point mutations present in each mutant. For each location in a mutant (where location is indicated on the horizontal axis) that was changed via the error-prone PCR, a black dot is indicated. With only a handful of exceptions, all of these changes are base transitions rather than transversions, so the sequence of each of the 69 clones can be inferred from the WT sequence shown in quadrant D. Reading down from quadrant B into quadrant C shows how mutations at a particular location partition between the two classes of mutants: the top and bottom 50th percentiles. Sites that have no effect on the fluorescence phenotype should partition equally between the two classes, i.e. they should follow a binomial distribution with $p = 0.5$. Sites that deviate from this distribution are labeled with a dot and are colored either green or red, corresponding to the apparent effect of a mutation at the site. For these sites, p–values are indicated, where this value is the probability of seeing a distribution at least as skewed to one side. Sites that were subsequently tested experimentally (see below) are indicated with an asterisk, where the color of the asterisk denotes the expected effect of a mutation at the site. We chose a range of sites to test experimentally, from those with high-confidence (low p–value) positive effects, to those with low-confidence (p–value ~ 0.5) negative effects (see Table 4.8 on page 213). These sites are also shown in quadrant D, which contains the WT nucleotide sequence of the promoter region that was subjected to mutation.

**Site–directed mutagenesis of predicted sites**

We selected 8 sites in the promoter region to test whether their phenotypic effects, as predicted by the statistical method, agreed with their observed effects when the mutations were introduced individually, without the background of other mutations. These 8 mutated positions are shown in Table 4.8 on page 213 and labeled in Figure 4-15 on the facing page, parts C&D. The sites
Figure 4-15: Statistical distribution of mutations and their effects on mutant fluorescence. See text for a description.
were chosen to span a range of characteristics. The -8 site was predicted to have a negative effect on promoter strength with high confidence, i.e. it was statistically significant (see Table 4.8). The -10, -28, and -123 sites were predicted to have negative effects, but had moderate p–values and, thus, medium–to–low confidence. Sites -14 and -21 were predicted to have positive effects with high confidence. The sites -82 and -96 were chosen because they had p–values of exactly 0.5. Notably, there are two ways that a position could have produced an insignificant p–value (i.e. a p–value close to 0.5): the mutation could partition equally between the two classes, or the mutation could have been observed very few times. Mutations at both the -82 and -96 sites were observed relatively few times and seemed to partition between the top–50th percentile and bottom–50th percentile classes with equal frequency. Thus, in the absence of a statistically significant correlation, we predicted they would have no effect on the phenotype. (These observations are summarized in Table 4.8 on page 213.)

For each of the sites listed in Table 4.8 on page 213, we created mutant strains incorporating transition SNPs at the specified location. Each of these mutants were analyzed using flow-cytometry to test the single-cell level of expression of GFP using the same protocols as for the parent mutant library. The fluorescence results for each mutant are shown in Table 4.8 on page 213 in the right-most columns. In addition, for certain combinations of sites in Table 4.8 on page 213, we created double and triple mutants (see Table 4.9 on page 214).

4.6.5 Discussion

As shown in Table 4.8 on page 213, the statistical method correctly predicts the phenotypic effects of 7/8 of the individual mutations that were tested. Furthermore, the phenotypic effects of the mutations with statistically significant p–values were correctly predicted. For these mutations, we showed that the effect of an individual mutation on the phenotype can be parsed out from a mutational spectrum, even when the effect is obscured by a background of other mutations.

It is interesting to note that while most of the statistically significant mutations are near the sigma factor binding sites, two are located further upstream of this region. The -123 site, which was not statistically significant, but was tested experimentally, showed that such distal sites are participating in the regulation of transcription.
There are a few caveats to the use of our statistical method. First, the method assumes independence between mutations. That is, we assume mutated sites cannot interact. As shown in Table 4.9 on page 214, 4/6 of the combination–mutations had the predicted effect. The two combination–mutants that had unintuitive phenotypes could be a result of interaction between sites. (Notably, the -82,-14,-21 triple mutant appeared to have a high fluorescence by visual inspection in a rich medium pre–culture; however, quantification of GFP activity by flow cytometry revealed consistently low measurements in the minimal medium used.)

The second caveat is that the method can require a significant number of mutants for each position: for a position to be statistically significant in our particular experiment, at least 4 observations were required. (This would be true for any two–phenotype mutational spectra, where each phenotype occurs with equal prior probability.) The number of observations required scales roughly with the number of mutation types. Our mutagenesis method introduced only transitions, not transversions, which allowed us to treat each site as "mutated" or "not mutated" without loss of information. The method can be applied to cases in which all four nucleotides are present; however, roughly 4 times as many observations would be required to make a statistically significant correlation between a particular nucleotide (at a single position) and a phenotype. Finally, the statistical method presented here is only applicable to situations in which the method used to introduce sequence diversity does not also introduce deletions or insertions. Ignoring relatively small insertions or deletions in the analysis would not significantly bias the results of identifying critical residues (data not shown). However, rigorously, alterations would be needed to differentiate between deletions and mutations in our statistical framework. In such cases, more complex models could be adapted, such as those used to describe the distribution and effects of naturally–occurring mutations over a fitness landscape for populations under positive and negative selective pressures [186, 212].

Despite its caveats, this method has a significant advantage when compared to deducing critical mutations using sequence data from only the best performing mutants. Intuitively, if we were to ignore the bottom–50th percentile in Figure 4–15 part C on page 209, we may mistakenly identify sites as associated with high fluorescence that are, in fact, evenly distributed between the two classes. That is, having sequence data for multiple phenotypes allowed us to determine, with quantifiable confidence, the effect of each individual mutation in a way that
discounts artifacts of the mutagenesis method, such as a bias for mutagenizing particular loci.
Table 4.8: Summary of site-directed mutagenesis loci. The selected sites, which span a range of p-values and predicted activities, were each mutated and assayed for fluorescence levels individually (see Figure 4.15 on page 209. As shown in the table, all sites but the -96 site were in the phenotypic class predicted by our statistical method.

<table>
<thead>
<tr>
<th>Site</th>
<th>Predicted activity</th>
<th>P-value</th>
<th>Observations</th>
<th>Confidence</th>
<th>Relative Fluorescence</th>
<th>Log Relative Fluorescence</th>
<th>Agreement?</th>
</tr>
</thead>
<tbody>
<tr>
<td>-8</td>
<td>Low</td>
<td>&lt;0.0001</td>
<td>22</td>
<td>High</td>
<td>0.036</td>
<td>-3.32</td>
<td>Yes</td>
</tr>
<tr>
<td>-10</td>
<td>Low</td>
<td>0.1094</td>
<td>6</td>
<td>Med</td>
<td>0.011</td>
<td>-4.52</td>
<td>Yes</td>
</tr>
<tr>
<td>-14</td>
<td>High</td>
<td>0.0625</td>
<td>4</td>
<td>High</td>
<td>1.428</td>
<td>0.35</td>
<td>Yes</td>
</tr>
<tr>
<td>-21</td>
<td>High</td>
<td>0.0625</td>
<td>4</td>
<td>High</td>
<td>1.585</td>
<td>0.46</td>
<td>Yes</td>
</tr>
<tr>
<td>-28</td>
<td>Low</td>
<td>0.3770</td>
<td>10</td>
<td>Low</td>
<td>0.756</td>
<td>-2.58</td>
<td>Yes</td>
</tr>
<tr>
<td>-82</td>
<td>No effect</td>
<td>0.5000</td>
<td>2</td>
<td>Low</td>
<td>0.926</td>
<td>-0.08</td>
<td>Yes</td>
</tr>
<tr>
<td>-96</td>
<td>No effect</td>
<td>0.5000</td>
<td>5</td>
<td>Med</td>
<td>0.046</td>
<td>-3.08</td>
<td>No</td>
</tr>
<tr>
<td>-123</td>
<td>Low</td>
<td>0.1938</td>
<td>12</td>
<td>Med</td>
<td>0.087</td>
<td>-2.45</td>
<td>Yes</td>
</tr>
</tbody>
</table>
# Table 4.9: Summary of double and triple mutants constructed by site-directed mutagenesis.

<table>
<thead>
<tr>
<th>Sites Predicted activity</th>
<th>Predicted Fluorescence</th>
<th>Log Relative Fluorescence</th>
<th>Agreement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>4.03</td>
<td>0.01</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>1.47</td>
<td>0.10</td>
<td>High</td>
</tr>
<tr>
<td>Yes</td>
<td>3.1</td>
<td>4.02</td>
<td>Low</td>
</tr>
<tr>
<td>Yes</td>
<td>3.69</td>
<td>3.4</td>
<td>High</td>
</tr>
<tr>
<td>No</td>
<td>0.4</td>
<td>4.95</td>
<td>High</td>
</tr>
<tr>
<td>Yes</td>
<td>6.69</td>
<td>1.94</td>
<td>High</td>
</tr>
</tbody>
</table>

-8.10-28  8.3-14.4  9.6-1  12  1.21-21  4.1-4.2  4.1-8.2  4.1-21
Appendix A

Abbreviations and reference data

A.1 Basic molecular biology data

• Figure A-1 on the following page shows structures and abbreviations for the 20 naturally occurring amino acids. The abbreviations shown in the figure are used consistently throughout this thesis.

• Figure A-2 on page 217 shows structures and abbreviations for the four nucleotides found in DNA and RNA, and urysil, which is found only in RNA.

• Table A.1 on page 217 shows the standard codon table that translates from three letter nucleotide sequences to the corresponding amino acid during the process of mRNA translation.
Figure A-1: Amino acid structures and abbreviations. The figure shows the chemical structure of the 20 naturally occurring amino acids and their three letter and one letter abbreviations.
Table A.1: Standard codon table. The table should be interpreted by reading the first and second nucleotides off of the vertical axis on the left, and reading the final nucleotide off of the horizontal axis at the top. For example, the amino acid corresponding to the three nucleotide sequence AAG is Arg, or arginine.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>C</th>
<th>G</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>Lys</td>
<td>Asn</td>
<td>Lys</td>
<td>Asn</td>
</tr>
<tr>
<td>F</td>
<td>Thr</td>
<td>Thr</td>
<td>Thr</td>
<td>Thr</td>
</tr>
<tr>
<td>i</td>
<td>Arg</td>
<td>Ser</td>
<td>Arg</td>
<td>Ser</td>
</tr>
<tr>
<td>r</td>
<td>Ile</td>
<td>Ile</td>
<td>MET</td>
<td>Ile</td>
</tr>
<tr>
<td>s</td>
<td>Gln</td>
<td>His</td>
<td>Gln</td>
<td>His</td>
</tr>
<tr>
<td>t</td>
<td>Pro</td>
<td>Pro</td>
<td>Pro</td>
<td>Pro</td>
</tr>
<tr>
<td>s</td>
<td>Arg</td>
<td>Arg</td>
<td>Arg</td>
<td>Arg</td>
</tr>
<tr>
<td>&amp;</td>
<td>Leu</td>
<td>Leu</td>
<td>Leu</td>
<td>Leu</td>
</tr>
<tr>
<td>t</td>
<td>Glu</td>
<td>Asp</td>
<td>Glu</td>
<td>Asp</td>
</tr>
<tr>
<td>S</td>
<td>Ala</td>
<td>Ala</td>
<td>Ala</td>
<td>Ala</td>
</tr>
<tr>
<td>e</td>
<td>Gly</td>
<td>Gly</td>
<td>Gly</td>
<td>Gly</td>
</tr>
<tr>
<td>c</td>
<td>Val</td>
<td>Val</td>
<td>Val</td>
<td>Val</td>
</tr>
<tr>
<td>o</td>
<td>.</td>
<td>Tyr</td>
<td>.</td>
<td>Tyr</td>
</tr>
<tr>
<td>n</td>
<td>Ser</td>
<td>Ser</td>
<td>Ser</td>
<td>Ser</td>
</tr>
<tr>
<td>d</td>
<td>.</td>
<td>Cys</td>
<td>Trp</td>
<td>Cys</td>
</tr>
<tr>
<td>U</td>
<td>Leu</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A.2 Supplementary data and analyses

A.2.1 Position weight matrix computation and matching

The code shown below is a simple Python script used to compute a position weight matrix. The script can be copied from this text and run on most personal computers. After the code, I present a brief example of how this should be run, using the yeast 3’ splice sites shown in Figure 1-9 on page 47.

#!/usr/bin/env python
import string
import sys
import math

# Usage
def usage():
    print sys.argv[0], "": make a weight matrix from an alignment file"
    print "Usage: ", sys.argv[0], "<alignment file>"

# a PWM class, basically just a list of dictionaries
# with a few methods for building/accessing
class Pwm:
    """A position weight matrix class""
    def __init__(self, aln):
        # find out what characters occurs in seqs
        self.chars = self.getChars(aln)
        self.length = aln.support()
        self.width = aln.wid()

        # initialize list of dicts
        # cm = count matrix
        # fm = frequency matrix
        self.cm = []
        self.fm = []
        for j in range(aln.width):
            self.cm.append( dict([(char, 0)for char in self.chars]) )
            self.fm.append( dict([(char, 0.0)for char in self.chars]) )

        # fill in the counts
        for j in range(self.width):
            for i in range(self.length):
c = aln.getChar(i, j)
self.cm[j][c] = self.cm[j][c] + 1

# calculate the frequency matrix
for c in self.chars:
    for j in range(self.wid()):
        self.fm[j][c] = float(self.cm[j][c]) / float(self.len())

self.calcEntropy()

def len(self):
    return self.length
def wid(self):
    return self.width
def getChars(self, aln):
    chars = []
    for j in range(aln.width):
        for i in range(aln.support()):
            c = aln.getChar(i, j)
            if chars.count(c) == 0:
                chars.append(aln.getChar(i, j))
    return chars
def getFreq(self, pos, char):
    return self.fm[pos][char]
def getCount(self, pos, char):
    return self.cm[pos][char]
def printPwm(self):
    print "Frequency Matrix:"
    for char in self.chars:
        s = "%s" % (char)
        for pos in range(self.wid()):
            s = s + (" %1.3f" % ( self.getFreq(pos, char) ))
        print s

def printPwmDNA(self):
    print "Frequency Matrix:"
    for char in ['A', 'T', 'G', 'C']:
        s = "%s" % (char)
        for pos in range(self.wid()):
            s = s + (" %1.3f" % ( self.getFreq(pos, char) ))
        print s
    print ""

# returns a vector, each member of which is
# the entropy at a pos in the pwm
def calcEntropy(self):
    self.entropy = []
base = 2
for pos in range(self.wid()):
    self.entropy.append(0)
    for char in self.chars:
        freq = self.getFreq(pos, char)
        if (freq > 0):

def getEntropy(self, pos):
    return self.entropy[pos]

# returns a vector, each member of which is
# the info content at a pos in the pwm
# take a dictionary containing background
# frequencies of various characters
def calcInfo(self, bg):
    bgInfo = 0.0
    base = 2
    for char, prior in bg.iteritems():
        if prior > 0:
            bgInfo = bgInfo - prior * math.log(prior, base)

    self.info = []
    self.ic = 0
    for pos in range(self.wid()):
        self.info.append(bgInfo - self.getEntropy(pos))
        self.ic = self.ic + bgInfo - self.getEntropy(pos)

def getInfo(self, pos):
    return self.info[pos]

def getIC(self):
    return self.ic

def printInfo(self):
    for pos in range(self.wid()):
        print "Position %d: entropy = %1.3f, information = %1.3f" \\
            % (pos+1, self.getEntropy(pos), self.getInfo(pos))

    print "\nTotal Information Content = %1.3f" % (self.getIC())

# compute a bitScore match of the pwm against a sequence
def score(self, seq, bg):
    total = 0
    for pos in range(self.wid()):
        char = seq[pos]
        # potential need for error checking here!
freq = self.getFreq(pos, char)
prior = bg[char]
if freq != 0 and prior != 0:
    total = total + math.log(freq/prior,2)
return total

# A simple sequence alignment class
class Alignment:
    """A simple sequence alignment class""
    def __init__(self, label):
        self.label = label
        self.seqs = []
        self.width = 0
    def addSeq(self, seq):
        self.seqs.append(seq)
        if(len(seq) > self.width):
            self.width = len(seq)
    def printAlignment(self):
        for seq in self.seqs:
            print seq
    def support(self):
        return len(self.seqs)
    def wid(self):
        return self.width
    def getChar(self, seq, pos):
        return self.seqs[seq][pos]
    def makePWM(self):
        self.pwm = Pwm(self)
    def printPWM(self):
        self.pwm.printPwm()
    def printPWMDNA(self):
        self.pwm.printPwmDNA()
    def calcInfo(self, bg):
        self.pwm.calcInfo(bg)
    def printInfo(self):
        self.pwm.printInfo()
    def selfScore(self, bg):
        mean = 0
        for seq in self.seqs:
            score = self.pwm.score(seq, bg)
            print "Sequence %s has score s = %0.3f" % (seq, score)
            mean = mean + score
        mean = mean / self.support()
print "Mean score = %.3f" % (mean)

def readAlignFile(alignFH, label):
    aln = Alignment(label)
    for line in alignFH:
        line = string.strip(line)
        aln.addSeq(line)
    return aln

# Main
def main():

    # see if we got the right number of command line args
    if len(sys.argv) != 2:
        usage()
        sys.exit(2)

    # get command line paramters
    fname1 = sys.argv[1]

    # open the alignment file
    try:
        alignFH = open(fname1, "r")
    except IOError, (errno, strerror):
        print "Error %s: %s" % (errno, strerror)
        sys.exit()

    aln = readAlignFile(alignFH, "My alignment")

    # close alignment file
    try:
        alignFH.close()
    except IOError, (errno, strerror):
        print "Error %s: %s" % (errno, strerror)
        sys.exit()

    # do all our fancy shizzle
    #aln.printAlignment()
    aln.makePWM()
    aln.printPWMDNA()
    aln.calcInfo({'A': 0.25, 'T': 0.25, 'G': 0.25, 'C': 0.25})
    aln.printInfo()
    print ""
    aln.selfScore({'A': 0.25, 'T': 0.25, 'G': 0.25, 'C': 0.25})
# execute main
if __name__ == "__main__":
    sys.exit(main())

>cat motifs.txt
AGCTGAC
GACTAAT
GGCTAAT
TTCTAAC
....edited for space...
TACTAAC
TACTAAC
TTTTAAC

>motif.py motifs.txt
Frequency Matrix:
A 0.067 0.627 0.000 0.000 0.893 1.000 0.000
T 0.773 0.240 0.120 1.000 0.027 0.000 0.133
G 0.093 0.120 0.000 0.000 0.080 0.000 0.000
C 0.067 0.013 0.880 0.000 0.000 0.000 0.867

Position 1: entropy = 1.127, information = 0.873
Position 2: entropy = 1.367, information = 0.633
Position 3: entropy = 0.529, information = 1.471
Position 4: entropy = 0.000, information = 2.000
Position 5: entropy = 0.576, information = 1.424
Position 6: entropy = 0.000, information = 2.000
Position 7: entropy = 0.567, information = 1.433

Total Information Content = 9.834

Sequence AGCTGAC has score s = 2.999
Sequence GACTAAT has score s = 6.650
Sequence GGCTAAT has score s = 4.266
Sequence CATTAAC has score s = 5.991
....edited for space...
Sequence TACTAAC has score s = 12.401
Sequence TACTAAC has score s = 12.401
Sequence TTTTAAC has score s = 8.142
Mean score = 9.834
A.2.2 Antimicrobial design data

Figure A-3 shows the gas and mass spectra for a peptide designed in Chapter 2. See Section 2.4 on page 88.

A)

![Gas and mass spectra for a peptide designed in Chapter 2.](image)

B)

![Mass fraction vs. mass for a peptide designed in Chapter 2.](image)

**Figure A-3:** Gas and mass spectra for the synth–1 peptide. As the figure shows, the peptide appears well above the 85% purity threshold. This peptide was designed using our preliminary, sensitive approach for designing antimicrobial peptides. However, the peptide was shown to have undetectable activity under good experimental conditions, prompting the more focused, specific approach for designing AmPs.
Appendix B

Gemoda file documentation

B.1 Introduction

This chapter contains detailed documentation of the source code implementation of the Gemoda algorithm described in Chapter 3 on page 111. The Gemoda software is written in the C programming language and segmented in such a way as to allow the extension of the algorithm to varieties of sequential data that were not anticipated by the authors. Furthermore, where possible the code was crafted to be “object–oriented like” for maximum readability. The software makes extensive use of the GNU Scientific Library [1] and the popular Basic Linear Algebra Subprograms (BLAS) [37, 68, 69] to speed–up computationally intensive operations associated with the discovery of motifs in three–dimensional protein structures and other real–valued data.

The Gemoda source code is available from http://web.mit.edu/bamel/gemoda. The software includes a number of “helper” applications for 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.
To install Gemoda, use the following recipe:

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`.

In the following sections of this appendix, I describe in detail the organization and design of the Gemoda software. These sections are organized by file and are designed to show the dependencies and interactions of different functions. As described in Chapter 3 on page 111, Gemoda operates in three steps: comparison, clustering, and convolution. The software keeps the steps clearly segmented.

**B.2 align.c File Reference**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include "FastaSeqIO.fastaIO.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

B.2.0.1 \#define ALIGN_ALPHABET 256

Definition at line 24 of file align.c.
Function Documentation

B.2.0.2 \texttt{int alignMat (char \ast s1, char \ast s2, int L, int \ast mat[]) \textbf{[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 \texttt{aaOrder}, and \texttt{mat}.

Referenced by \texttt{alignWordsMat\_bit()}.  

\begin{verbatim}
{  
  int i;
  int points = 0;
  int x, y;

  // Go over each character in the L-length window
  for (i = 0; i < L; i++)
  {
    // The integer corresponding to the character in
    // the first string, so that we can look it up
    // in one of our scoring matrices.
    x = aaOrder[(int) s1[i]];

    // And for the second character
    y = aaOrder[(int) s2[i]];

    // If the characters aren't going to be in the scoring
    // matrix, they get a -1 value...which we'll give zero
    // points to here.
    if (x != -1 && y != -1)
    {
      // Otherwise, they get a score that is looked up
      // in the scoring matrix
      points += mat[x][y];
    }
  }

  return points;
}
\end{verbatim}

B.2.0.3 \texttt{bitGraph\_t* alignWordsMat\_bit (sPat\_t \ast words, int wc, int \ast mat[]) \textbf{[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().

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

**Variable Documentation**

B.2. ALIGN.C FILE REFERENCE

B.2.0.4 const int aaOrder[]

Definition at line 32 of file matrices.h.

Referenced by alignMat().
B.3 bitSet.c File Reference

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

Include dependency graph for bitSet.c:

![Dependency graph](image)

Functions

- `bit_t * newBitArray (int bytes)`
- `bitSet_t * newBitSet (int size)`
- `int setTrue (bitSet_t *s, int x)`
- `int setFalse (bitSet_t *s, int x)`
- `int flipBits (bitSet_t *s)`
- `int fillSet (bitSet_t *s)`
- `int emptySet (bitSet_t *s)`
- `int checkBit (bitSet_t *s, int x)`
- `int deleteBitSet (bitSet_t *s)`
- `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 *s)`
- `int nextBitBitSet (bitSet_t *s, int start)`
- `int countBitGraphNonZero (bitGraph_t *bg)`
- `int printBitSet (bitSet_t *s)`
- `int bitGraphRowUnion (bitGraph_t *bg, int row1, int row2, bitSet_t *s)`
- `int bitGraphRowIntersection (bitGraph_t *bg, int row1, int row2, bitSet_t *s)`
- `int printBinaryBitSet (bitSet_t *s)`
Detailed Description

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

Definition in file `bitset.c`.

Function Documentation

B.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`.

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

```c
int bitcount32_precomp (unsigned int n)
{
    // works only for 32-bit ints
    return bits_in_char[n & 0xffu] + bits_in_char[(n >> 8) & 0xffu] + bits_in_char[(n >> 16) & 0xffu] + bits_in_char[(n >> 24) & 0xffu];
}
```

### B.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.

```c
int bitcount64 (unsigned int n)
{
    n = PCCOUNT (n, 0);
    n = PCCOUNT (n, 1);
    n = PCCOUNT (n, 2);
    n = PCCOUNT (n, 3);
    n = PCCOUNT (n, 4); // for 64-bit integers
    return n;
}
```

### B.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().

```c
int bitGraphCheckBit (bitGraph_t * bg, int x, int y)
{
    return checkBit (bg->graph[x], y);
}
```
B.3. BITSET.C FILE REFERENCE

B.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 }

B.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 }

B.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 }
\textbf{B.3.0.12} \textit{int bitGraphSetFalseDiagonal (bitGraph_t * bg)}

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

Definition at line 714 of file \texttt{bitSet.c}.

References \texttt{bitGraph_t::graph}, and setFalse().

Referenced by \texttt{convolve()}.

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

\textbf{B.3.0.13} \textit{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 \texttt{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 \texttt{bitGraph_t} object).

Definition at line 669 of file \texttt{bitSet.c}.

References \texttt{bitGraph_t::graph}, and setFalse().

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

\textbf{B.3.0.14} \textit{int bitGraphSetTrue (bitGraph_t * bg, int x, int y)}

Sets a specific bit in a bitGraph true. Input: a \texttt{bitGraph_t} object, the index of the row of the \texttt{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 \texttt{bitGraph_t} object).

Definition at line 641 of file \texttt{bitSet.c}.

References \texttt{bitGraph_t::graph}, and setTrue().

\begin{verbatim}
 642 { 
 643    setTrue (bg->graph[x], y); 
 644    return 0; 
 645 }
\end{verbatim}
B.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 }

B.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 }

B.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.
APPENDIX B. GEMODA FILE DOCUMENTATION

B.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.

B.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 }

B.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 }

B.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     {

B.3.0.22 int checkBit (bitSet_t * sI, 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 }
```

B.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 }
```
### B.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().

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

### B.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.

```c
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     sum += countSet (bg->graph[i]);
544 }
545 return sum;
546 }
```

### B.3.0.26  int countSet (bitSet_t * s)

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

B.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 }
B.3. BITSET.C FILE REFERENCE

B.3.0.28 int deleteBitSet (bitSet_t * s1)


Definition at line 159 of file bitSet.c.

References bitSet_t::tf.

Referenced by convolve(), deleteBitGraph(), filterGraph(), findCliqued(), 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 }

B.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 }

B.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.
B.3.0.31  **int emptySet (bitSet_t * s)**

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()`.

```
842 {
843  emptySet (bg->graph[row]);
844  return 0;
845 }
```

B.3.0.32  **int fillBitGraph (bitGraph_t * bg)**

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 }
```

B.3.0.33  **int fillSet (bitSet_t * s)**

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 }
```
B.3.1.34  int flipBits (bitSet_t * s)

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       s1->tf[i] = ~s1->tf[i];
113   return 0;
114 }
```

B.3.1.35  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             bitSetIntersection (bg1->graph[i], bs, bg1->graph[i]);
759       } else
760       {
761         emptySet (bg1->graph[i]);
762       }
763   return 0;
764 }
```

B.3.1.36  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().

```c
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
%s
", strerror (errno));  
27       fflush (stderr);  
28       exit (0);  
29     }  
30   // Set them all false  
31   memset (b, 0, bytes);  
32   return b;  
33 }
```

**B.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().

```c
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 }
```
**B.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().

```c
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    }
```

**B.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().

```c
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    // s1->max is the number of bits in s1
481    // test to see if we’re looking too high
482    if (start >= s1->max)
```
APPENDIX B. GEMODA FILE DOCUMENTATION

B.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().
B.3. BITSET.C FILE REFERENCE

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().

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().

printf ("%d", (BSTEST (s1->tf, i) ? 1 : 0));
return 0;

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 }

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 for (i = 0; i < s1->max; i++)
565 {
566 if (BSTEST (s1->tf, i))
567 {
568 printf (" %d", i);
569 }
570 }
571 printf ("\n");
572 return 0;
573 }
**B.3.0.43** int setFalse (bitSet_t * sI, 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().

```c
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 }
```

**B.3.0.44** int setTrue (bitSet_t * sI, 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().

```c
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 }
```
# B.4 convll.c File Reference

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

Include dependency graph for convll.c:

![Dependency Graph](image)

## 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 *cliqueSet, 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)`
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

B.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, "\n\nNULL set for clique! - addToStacks\n");
504 fflush (stderr);
505 exit (0);
506 }
507 return memberStacks;
508 }

B.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 213 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, "\n\nMemory Error - bitSetToCSet - [1]\n\n\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, "\n\nMemory Error - bitSetToCSet - [2]\n\n",
231 ...
APPENDIX B. GEMODA FILE DOCUMENTATION

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 }

B.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, "\nMemory error - checkCliquecSet\n\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  
285  if (cliquecSet->size < 1)
286  {
287    fprintf (stderr, "\nClique of zero size! - checkCliquecSet\n\n");  
288    fflush (stderr);
289    exit (0);
291   }
292   // Find the first sequence number.
293   seqNums[0] = indexToSeq[cliqueSet->members[0]];
294   // Iterate over the remaining size of the clique
295   for (i = 1; i < cliqueSet->size; i++)
296   {
297     // Find the next sequence number.
298     thisSeq = indexToSeq[cliqueSet->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   // Now just see what the value of the last number in the array is;
318   // if it’s the sentinel, then we didn’t find instances in p
319   // unique sequences. If it’s not the sentinel, then we’ve met
320   // the -p criterion.
321   if (seqNums[p - 1] == -1)
322   {
323     free (seqNums);
324     return (0);
325   }
326   else
327   {
328     free (seqNums);
329     return (1);
330   }
331 
B.4. CONVLL.C FILE REFERENCE

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

Referenced by convolve().
APPENDIX B. GEMODA FILE DOCUMENTATION

B.4.0.49  **emptyMemberStacks** (mll_t ** members, 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().

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

### B.4.50 `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().

```c
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     return memberStacks;
527 }
```

### B.4.51 `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().

```c
173 {
174     // Check to make sure that the head is not already initialized.
```
B.4.0.52  \texttt{mll\_t} * \texttt{mergeIntersect} (\texttt{cll\_t} * \texttt{first}, \texttt{cll\_t} * \texttt{second}, \texttt{mll\_t} * \texttt{intersection}, \texttt{bitSet\_t} * \texttt{printstatus}, \texttt{int} * \texttt{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 \texttt{cSet\_t::members}, \texttt{pushMemStack()}, and \texttt{cnode::set}.

Referenced by singleCliqueConv().

```c
761 {
762    int i = 0, j = 0, status = 0;
763
764    // Make sure we are still in-bounds, otherwise we bail out
765    // We'll refer to the offset currently being analyzed from the
766    // first clique as the 'first offset' and the offset currently
767    // being analyzed from the second clique as the 'second offset'
768    while ((i < first->set->size) && (j < second->set->size))
769    {
770        // If the second offset is earlier than the first offset plus
771        // one, then we move on to the next possible second offset
772        if ((first->set->members[i] + 1) > second->set->members[j])
773        {
774            j++;
775        }
776        // If the second offset is later than the first offset plus
777        // one, then we move on the next possible first offset
778        else if ((first->set->members[i] + 1) < second->set->members[j])
779        {
780            i++;
781        }
782        // Otherwise, the second offset is equal to the first offset
783        // plus one, so we have an extendable node. Push that on
784        // to the intersection stack, move both the first and second
785        // offsets to their respective next possible offsets, and
786        // increment the support counter for the new clique (status)
787        else
788        {
789            intersection = pushMemStack (intersection, first->set->members[i]);
790            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 }

B.4. CONVLL.C FILE REFERENCE

B.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().
B.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().

```c
while (head != NULL)
{
    head = popCll (head);
}
return head;
```

B.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().

```c
  if (head == NULL)
  {
    fprintf (stderr, "Can't pop a null linked list\n");
    fflush (stderr);
    exit (0);
  }
  // unless this is the end of the linked list, set the new head
  // to the next member of the list. Otherwise, since by default the
  // new head is NULL, it will properly return an empty list
  if (head->next != NULL)
  {
    newNode = head->next;
  }
  // Check to see if there is a set. If there is, and there are members,
  // then first free the members. And if there is a set, then free it.
  if (head->set != NULL)
  {
    if (head->set->members != NULL)
B.4. CONVLL.C FILE REFERENCE

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 }

B.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, "Can’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 }

B.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)  

APPENDIX B. GEMODA FILE DOCUMENTATION

B.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.

B.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.
B.4. CONVLL.C FILE REFERENCE

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     printf ("\nSupport = %d (id = %d) \n", node->set->size, node->id);
1486     printf ("Members = \n");
1487     for (i = 0; i < node->set->size; i++)
1488     {
1489         printf ("\t%d\n", node->set->members[i]);
1490     }
1491     return 1;
1492 }
1493

B.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 }
1088

B.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.
APPENDIX B. GEMODA FILE DOCUMENTATION

558 {  
559   int i = 0;  
560   mll_t *curr = NULL;  
561   for (i = 0; i < size; i++)  
562   {  
563     curr = memberStacks[i];  
564     printf ("Offset %d: ", i);  
565     while (curr != NULL)  
566     {  
567       printf ("%, curr->cliqueMembership);  
568       curr = curr->next;  
569     }  
570   }  
571   printf ("\n");  
572 }

B.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   // We’ll do this similar to the pruneBitGraph function... we will  
522   // keep track of which source sequence each motif occurrence was in.  
523   // Again, since the occurrences are listed monotonically, we only  
524   // need to compare the last non-sentinel index to the current  
525   // sequence number.  
526   seqNums = (int *) malloc (p * sizeof (int));  
527   if (seqNums == NULL)  
528     {  
529       fprintf (stderr, "Memory error - pruneCll\n\ns\n", strerror (errno));  
530       fflush (stderr);  
531       exit (0);  
532     }  
533   while (curr != NULL)  
534   {  
535     {  
536       // First make sure the set size is at least p.  
537       // This is redundant, but extremely simple and not expensive,  
538       // so we’ll leave it in just as a check.  
539       if (curr->set->size < p)  
540         {  
541           if (prev != NULL)  
542             {  
543               prev->next = curr->next;  
544             }  
545           }  
546         }  
547       }  
548   }  
549   free (seqNums);  
550   for (i = 0; i < p; i++)  
551   {  
552     seqNums[i] = indexToSeq[i];  
553   }  
554   return head;  
555 }
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   // Note, we’ve checked to make sure size > p, and we know
564   // p must be 2 or greater, so we can start at 1 without
565   // worrying about segfaulting
566   for (i = 1; i < curr->set->size; i++)
567       {
568           thisSeq = indexToSeq[curr->set->members[i]];
569           if (thisSeq != seqNums[j])
570               {
571                   j++;
572                   seqNums[j] = thisSeq;
573                   if (j == p - 1)
574                       {
575                           break;
576                       }
577               }
578       }
579   // Same story as before... if the last number is -1,
580   // then we didn’t have enough to fill up the <p> different
581   // slots, so this doesn’t meet our criterion.
582   if (seqNums[p - 1] == -1)
583       {
584           if (prev != NULL)
585               {
586                   prev->next = curr->next;
587               }
588           else
589               {
590                   head = curr->next;
591               }
592       storage = curr->next;
593       free (curr->set->members);
594       free (curr->set);
595       free (curr);
596       curr = storage;
597       }
598   else
599       {
600           prev = curr;
601           curr = curr->next;
602       }
603   free (seqNums);
604   return (head);
B.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 memory
353 // 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 }

B.4.0.64  cll_t* pushClI (cll_t * head)

Pushes a new, empty head onto a linked list of cliques. Note: this should always be followed by a call to initheadClI, 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().

```c
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      printf (stderr, "$\nMemory Error - pushCll
%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 }
```

B.4. CONVLL.C FILE REFERENCE

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().

```c
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 }

B.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 }

B.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().

```c
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 }
```

**B.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().

```c
953 {
```
int foundStatus = 0;
cll_t *curr = head;
cll_t *prev = NULL;
int i = 0, j = 0, breakFlag = 0;

while (curr != NULL)
{
    if (curr->id == node)
    {
        foundStatus = 1;
        break;
    }
    curr = curr->next;
}

if (foundStatus == 0)
{
    fprintf (stderr, "\nFirst clique not found! (removeSupers)\n");
    fflush (stderr);
    exit (0);
}

// Now this is trickier, to remove nodes from the middle of a linked
// list; this means that we need to remember which node we were just
// at so that we can connect it to the node after the one we are
// about to delete.
prev = curr;
curr = curr->next;

// This code is similar to that in uniqClique.
// Descend through all members of the next level’s linked list.
while (curr != NULL)
{
    i = 0;
    j = 0;
    breakFlag = 0;

    // The proposed convolved clique will be referred to as the
    // 'first' clique, and the current clique being analyzed
    // in the next level is the 'second' clique.
    // Continue if we have more members in both cliques. We will
    // have already broken out if it is not possible for this
    // second clique to be a subset of the first.
    while ((i < newClique->size) && (j < curr->set->size))
    {
        // If the current member of the first clique is
        // less than the current member of the second clique
        // then it is still possible that the first is a
        // superset of the second, so move on to the next
        // member.
        if (newClique->members[i] < curr->set->members[j])
        {
            i++;
        }
        else if (newClique->members[i] > curr->set->members[j])
        {
            breakFlag = 1;
            break;
        }
        else
        {
            breakFlag = 1;
            break;
        }
    }
}
// If the breakflag is 1, then we know
// that there is a member of the second clique not in the
// first, and so the second is not a subset. If the breakflag
// is 0 but j is less than the second clique’s size, then
// we must have broken because we ran out of members in the
// first clique... thus, there is a member of the second
// clique not in the first. Thus, only if the breakflag is
// 0 and j is equal to the size of the second clique do we
// know that every member of the second clique is in the first
// and that the second clique can thus be removed.
if ((breakFlag == 0) && (j == curr->set->size))
{
    // Make the previous clique point to the next one
    // instead of the current one.
    prev->next = curr->next;
    // Free all of the memory used by the current clique.
    free (curr->set->members);
    free (curr->set);
    free (curr);
    curr = prev->next;
}
else
{
    // Otherwise, the current second clique is not a
    // subset of the first, and we advance the prev and
    // curr pointers.
    prev = curr;
    curr = curr->next;
}

return head;

B.4. CONVLL.C FILE REFERENCE

612 {
613     int i = 0;
614     emptySet (queue);
615 }
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

B.4.0.69  

bitSet_t* searchMemsWithList (int *list, int listszie, 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().
B.4.0.70  **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 \( \geq \) the number of cliques in the current clique linked list. Output: 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     // Set the bit in queue corresponding to the current clique
592     // membership true
593     setTrue (queue, curr->cliqueMembership);
594     curr = curr->next;
595   }
596
597   return queue;
598 }
```

B.4.0.71  **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, "Nonexistent 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, "First clique not found! - singleCliqueConv\n");
685                     fflush (stderr);
686                     exit (0);
687                 }
688             first = *firstGuess;
689         }
690     // Our guesses depend on monotonic traversal. If we don’t find
691     // the second clique, then bail out.
692     while ((*secondGuess)->id != secondClique)
693         {
694             if ((*secondGuess)->next != NULL)
695                 {
696                     (*secondGuess) = (*secondGuess)->next;
697                 }
698             else
699                 {
700                     fprintf (stderr, "Second clique not found! - singleCliqueConv\n");
701                     fflush (stderr);
702                     exit (0);
703                 }
704             second = *secondGuess;
705         }
706     // 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 }

B.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;
B.4. CONVLL.C FILE REFERENCE

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 < cliquetcSet->size) && (j < head->set->size) &&
\begin{verbatim}
((asubbflag == 1) || (bsubaflag == 1))
{
  // If the current member of the first clique is less
  // than the current member of the second clique,
  // it is impossible for the first clique to be a
  // subset of the second (since the members are
  // traversed in ascending order.
  if (cliquecSet->members[i] < head->set->members[j])
  {
    i++;
    asubbflag = 0;
  }
  else if (cliquecSet->members[i] > head->set->members[j])
  {
    j++;
    bsubaflag = 0;
  }
  // Otherwise, they matched this time, so move them
  else
  {
    i++;
    j++;
  }

  // If the proposed clique is a subset of some other clique
  // in the next level, then return -2, and it won't be added.
  // (Note, this also is how exact duplicates are handled.)
  if ((asubbflag == 1) && (i == cliquecSet->size))
  {
    return (-2);
  }
  // If the proposed clique is a superset of some other clique(s)
  // in the next level, then return the id of the first clique
  // of which it is a superset.
  if ((bsubaflag == 1) && (j == head->set->size))
  {
    return (head->id);
  }
  // If we've gotten here, we've checked all cliques in the previous
  // level and haven't found the proposed clique to be a superset or
  // a subset... if so, then we're all good, so return a -1.
  return (-1);
}
\end{verbatim}

\textbf{B.4.0.74} \texttt{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().

```c
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
%s
",
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     // Note that we start from the end of the cSet member list so that
1251     // we can conolve the highest-id cliques first, which are at the
1252     // beginning of our stack of cliques.
1253     for (i = cliquesToSearch->size - 1; i >= 0; i--)
1254     {
1255         nextPhase = singleCliqueConv (head, node->id, firstGuess,
1256             cliquesToSearch->members[i], secondGuess,
1257             nextPhase, printStatus, support);
1258     }
1259     // And then we free everything that we created
```
B.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().

```c
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
%s
", strerror (errno));
1297 fflush (stderr);
1298 exit (0);
1299 }
1300 // Start off at the head.
1301 *firstGuess = *head;
1302 // Convolve a whole clique at a time, traversing the linked list.
1303 // Note that firstGuess gets altered within the function.
1304 while (curr != NULL)
1305 {
1306 nextPhase =
1307 wholeCliqueConv (*head, curr, firstGuess, memList, numOffsets,
1308 nextPhase, printStatus, support);
1309 curr = curr->next;
1310 }
1311 // Now go back to the head for printing output
1312 curr = *head;
1313 // printf("%n
****************************************************
1314 // printf("Length = %d", length);
```
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().
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

B.4.0.77 int cliquecounter = 0

Definition at line 335 of file convll.c.
Referenced by pushClique().
B.5  convll.h File Reference

#include <stdio.h>
#include <stdlib.h>
#include "bitSet.h"

Include dependency graph for convll.h:

```
convll.h
+++
```

This graph shows which files directly or indirectly include this file:

```
convll.h
+++
```

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)
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**

**B.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.

**B.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**

**B.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().

**B.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().

**B.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().

B.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().

B.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().

B.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().
B.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().

B.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().

B.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().

B.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().
B.5.0.90 \textbf{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().

B.5.0.91 \textbf{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().

B.5.0.92 \textbf{int printCll (cll\_t * head)}

Prints the members (cliques) of a linked list in the format: \textit{id} = unique id number of clique within linked list; \textit{Length} = number of members of clique, if available; \textit{Size} = length of each member of clique; \textit{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.

B.5.0.93 \textbf{int printCllPattern (cll\_t * node, int length)}

Prints out the contents of a clique linked list node in this format: \textit{support} = number of motif occurrences (\textit{id} = some id number); \textit{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.

B.5.0.94 \textbf{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.

**B.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().

**B.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().

**B.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().

**B.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().

B.5.0.99  

```c
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().

B.5.0.100  

```c
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().

B.5.0.101  

```c
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().

B.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. Output: the updated bitSet_t object.

Definition at line 516 of file convll.c.
References mnode::cliqueMembership, mnode::next, and setTrue().
Referenced by searchMemsWithList().

B.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().

B.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().

B.5.0.105  int uniqClique (cSet_t * cliqueSet, 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().

B.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().
B.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().

B.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().
B.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

B.6.0.109  
#define BIG_BUFFER 1000000

Definition at line 11 of file fastaSeqIO.c.

B.6.0.110  
#define BUFFER 100000

Definition at line 10 of file fastaSeqIO.c.

Function Documentation

B.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 }

B.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       }
81       if (newLine == 1 && myChar == '>') {
82           count++;
83       }
84       if (myChar == '\n') {
85           newLine = 1;
86       } else {
87           newLine = 0;
88       }
89       myChar = fgetc(INPUT);
90   }
91   fseek(INPUT, start, SEEK_SET);
92   return count;
93 }
B.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().

B.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().
B.6. FASTASEQIO/FASTASEQIO.C FILE REFERENCE

B.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 }

B.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 }

B.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!"
115         "\n\n");
116         fflush(stderr);
117         return NULL;
118     }
119     buf = (char **) malloc ( (int)(nl+1) * sizeof(char *));
if (buf == NULL) {
    fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
    fflush(stderr);
    exit(0);
}

// measure the first line
tls = measureLine(INPUT) + 1;
if(tls != 0) {
    buf[i] = (char *) malloc ( tls * sizeof(char));
    if (buf[i] == NULL) {
        fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
        fflush(stderr);
        exit(0);
    }
    fgets(buf[i], tls, INPUT);
    do {
        if(buf[i][strlen(buf[i])-1] == '\n') {
            buf[i][strlen(buf[i])-1] = '\0';
        }
        tls = measureLine(INPUT) + 1;
        if(tls != 0) {
            i++;
            buf[i] = (char *) malloc ( tls * sizeof(char) );
            if (buf[i] == NULL) {
                fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
                fflush(stderr);
                exit(0);
            }
        }
    } while (fgets(buf[i], tls, INPUT) != NULL);
    free(buf[i]);
    buf = (char **) realloc ( buf, i * sizeof(char *) );
    if (buf == NULL) {
        fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
        fflush(stderr);
        return NULL;
    }
    // I think that 'i' might actually be the # of lines
    // plus one here? somehow line 131 isn't being freed,
    // or at least 2 bytes of it.
    *n = i;
    return buf;
}

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().
206    buf = ReadFile(INPUT, &nl);
207    if(buf == NULL){
208       return NULL;
209    }
210
211    // Count how many sequences we have
212    for( j=0 ; j<nl ; j++ ){
213       if(buf[j][0] == '>'){
214          ns++;
215       }
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
225    ll = (sSize_t *) malloc ( ns * sizeof(sSize_t) );
226    if(ll == NULL){
227       fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
228       fflush(stderr);
229       exit(0);
230    }
231
232    // find the first sequence
233    k=0;
234    while( buf[k][0] != '>' ){
235       k++;
236    }
237
238    // record how large each sequence is
239    i = -1;
240    for( j=k ; j<nl ; j++ ){
241       if(buf[j][0] == '>'){
242          i++;
243          ll[i].start = j;
244          ll[i].stop = j;
245          ss[i].start = j+1;
246          ss[i].size = strlen(buf[j]) ;
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
260    initAofFSeqs(aos, ns);
261
262    for ( i=0 ; i<ns ; i++ ){
263       if( ll[i].size > 0 ){
264          aos[i].label = (char *) malloc ( (ll[i].size+1) * sizeof(char) );
265          if( aos[i].label == NULL){
266             fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
267             fflush(stderr);
268             exit(0);
269          }
270          aos[i].label[0] = '\0';
271          for ( j=ll[i].start ; j<ll[i].stop ; j++ ){
272             // both instances of strcat here are using
273             // .label/.seq's that are NULL and that is
274             //
APPENDIX B. GEMODA FILE DOCUMENTATION

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, "Memory Error\n\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 for ( i=0 ; i<nl ; i++ ){
294    free(buf[i]);
295 }
296 free(buf);
297 free(buf);
298 *numberOfSequences = ns;
299 return aos;
300 }

B.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    buf = ReadFile(INPUT, &nl);
178    if(buf == NULL){
179        return NULL;
180    }
181    aos = (fSeq_t *) malloc ( nl * sizeof(fSeq_t) );
182    if( aos == NULL){
183        fprintf(stderr, "Memory Error\n\n", strerror(errno));
184        fflush(stderr);
185        exit(0);
186    }
187    initAofFSeqs(aos, nl);
188    for ( i=0 ; i<nl ; i++ ){
189        aos[i].seq = buf[i];
190    }
191    free(buf);
192    *numberOfSequences = nl;
193    return (aos);
194 }
B.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 }
B.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

B.7.0.121  long CountFSeqs (FILE * INPUT)

Definition at line 44 of file fastaSeqIO.c.
B.7.0.122  long countLines (FILE * INPUT)

Definition at line 69 of file fastaSeqIO.c.
Referenced by ReadFile().

B.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().

B.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().

B.7.0.125  long measureLine (FILE * INPUT)

Definition at line 25 of file fastaSeqIO.c.
Referenced by ReadFile().

B.7.0.126  int printFSeqSubSeq (fSeq_t * seq, int start, int stop)

Definition at line 14 of file fastaSeqIO.c.
References fSeq_t::seq.

B.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().

B.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.
int WriteFSeqA (FILE * MY_FILE, fSeq_t * arrayOfSequences, int start, int stop)

Definition at line 332 of file fastaSeqIO.c.
B.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)
**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**

**B.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 629 of file `newConv.c`.

Referenced by main().

```c
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  } else
661      {
662          singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
663                              p);
664      }
665      fprintf (OUTPUT_FILE, "Clusters found! Now filtering clusters (if option set)...\n");
666      fflush (NULL);
667      if (p > 1)
668      {
669          elemPats = pruneCll (elemPats, indexToSeq, p);
670      }
671      deleteBitSet (cand);
672      deleteBitSet (mask);
673      deleteBitSet (Q);
674
675      // Now let's convolve what we made.
676      if (noConvolve == 0)
677      {
678          fprintf (OUTPUT_FILE, "Now convolving...\n");
679          fflush (NULL);
680          allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
681      }
682      else
683      {
684          curr = elemPats;
685          while (curr != NULL)
686          {
687              yankCll (&elemPats, NULL, &curr, &allCliques, 0);
688          }
689      }
690      return allCliques;
691  }

B.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++)
if (s[i] == ',')
    numParams++;
}
return numParams;

---

### B.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(), popAllCl(), readRealData(), realComparison(), bitGraph_t::size, rdh_t::size, sortByStats(), and usage().

```c
int inputOption = 0;
char *sequenceFile = NULL;
FILE *SEQUENCE_FILE = NULL;
char *outputFile = NULL;
FILE *OUTPUT_FILE = NULL;
int L = 0;
int status = 0;
double g = 0;
int sup = 2;
int R = 1;
int P = 0;
int compFunc = 0;
double *extraParams = NULL;
int numExtraParams = 0;
int i = 0, j = 0;
/*
   int j, k, i, l;
*/
int noConvolve = 0;
int samp = 1;
int supportDim = 0, lengthDim = 0;
bitGraph_t *oam = NULL;
unsigned int **d = NULL;
oamSize = 0;
cll_t *allCliques = NULL;
/*
   cll_t *curClique = NULL;
*/
/*
   int curSeq;
*/
/*
int curPos;
*/
int clusterMethod = 0;
int joelOutput = 0;

// gemoda-r new stuff
rdh_t *data = NULL;

Get command-line options */
while ((inputOption = getopt(argc, argv, "p:m:e:i:o:l:g:k:c:njs:")) != EOF)
{
  switch (inputOption)
  {
    // Comparison metric
    case 'm':
      compFunc = atoi(optarg);
      break;
    // Input file
    case 'i':
      sequenceFile = optarg;
      break;
    // Output file
    case 'o':
      outputFile = (char *) malloc((strlen(optarg) + 1) * sizeof(char));
      if (outputFile == NULL)
        {
          fprintf(stderr, "Error allocating memory for options.\n");
          exit(EXIT_FAILURE);
        }
      else
        {
          strcpy(outputFile, optarg);
        }
      break;
    // Minimum motif length
    case 'l':
      L = atoi(optarg);
      break;
    // Minimum motif similarity score
    case 'g':
      g = atof(optarg);
      status++;
      break;
    // Minimum support (number of motif occurrences)
    case 'k':
      sup = atoi(optarg);
      break;
  }
}

Recursive initial pruning: an option for clique finding.
* It takes all nodes with less than the minimum
* number of support and removes all of their nodes, and does this
* recursively so that nodes that are connected to many sparsely connected
* nodes will be removed and not left in the
* This option is deprecated as it is at worst no-gain and at best useful.
* It will be on by default for clique-finding, but can be turned
* back off with some
* minor tweaking. For almost all cases in which it does not speed
* up computations, it will have a trivial time to perform. Thus, if
* clique-finding is turned on, then R is set to 1 by default.
  case 'r':
    R = 1;
  break;

Optional pruning parameter to require at motif occurrences
263     // in at least P distinct input sequences
264     case 'p':
265         P = atoi(optarg);
266         break;
267     // Clustering method.
268     case 'c':
269         clusterMethod = atoi(optarg);
270         break;
271     // Extra parameters for comparison function
272     case 'e':
273         numExtraParams = countExtraParams(optarg);
274         extraParams = parseExtraParams(optarg, numExtraParams);
275         break;
276     case 'n':
277         noConvolve = 1;
278         break;
279     case 'j':
280         joelOutput = 1;
281         break;
282     case 's':
283         samp = atoi(optarg);
284         break;
285     // Catch-all.
286     case '?':
287         fprintf(stderr, "Unknown option '-%c'.\n", optopt);
288         usage(argv);
289         return EXIT_SUCCESS;
290     default:
291         usage(argv);
292         return EXIT_SUCCESS;
293     }
294 }
295 // Require an input file, a nonzero length, and a similarity threshold
296 // to be set.
297 if (sequenceFile == NULL || L == 0 || status < 1)
298 {
299     usage(argv);
300     return EXIT_SUCCESS;
301 }
302 // Open the sequence file
303 if ((SEQUENCE_FILE = fopen(sequenceFile, "r")) == NULL)
304 {
305     fprintf(stderr, "Couldn't open file %s; %s\n", sequenceFile,
306            strerror(errno));
307     exit(EXIT_FAILURE);
308 }
309 // Open the output file
310 if (outputFile != NULL)
311 {
312     if ((OUTPUT_FILE = fopen(outputFile, "w")) == NULL)
313     {
314         fprintf(stderr, "Couldn't open file %s; %s\n", outputFile,
315                 strerror(errno));
316         exit(EXIT_FAILURE);
317     }
318 } else
319 {
320     OUTPUT_FILE = stdout;
321 }
322 // Verbosity in output helps to distinguish output files.
323 fprintf(OUTPUT_FILE, "Input file = %s\n", sequenceFile);
324 fprintf(OUTPUT_FILE, "l = %d, k = %d, g = %f\n", L, sup, g);
if (P > 1)
{
    fprintf (OUTPUT_FILE, "Minimum # of sequences with motif = %d\n", P);
}

if (R > 0)
{
    fprintf (OUTPUT_FILE, "Recursive pruning is ON.\n");
}

data = readRealData (SEQUENCE_FILE);
fclose (SEQUENCE_FILE);

data = readRealData (SEQUENCE_FILE);
fclose (SEQUENCE_FILE);

 data = readRealData (SEQUENCE_FILE);
fclose (SEQUENCE_FILE);

if ((samp > 0) && (clusterMethod == 0))
{
    // We are currently using one gap per sequence, as done in
    // realCompare.c's call to initRdhIndex in realComparison.
    // Note that this is data->size, NOT oam->size.
    d =
        getStatMat (oam, sup, L, &supportDim, &lengthDim, data->size, samp,
        OUTPUT_FILE);
    allClique s =
        convolve (oam, sup, R, data->indexToSeq, P, clusterMethod,
        data->offsetToIndex, data->size, noConvolve, OUTPUT_FILE);

    oamSize = oam->size;
    // Do some early memory cleanup since this is so big.
    deleteBitGraph (oam);
    if ((samp > 0) && (clusterMethod == 0))
    {
        cumDMatrix (d, allClique s, supportDim, lengthDim, oamSize, data->size);
        calcStatAllClique s (d, allClique s, oamSize - data->size);
        allClique s = sortByStats (allClique s);
    }

    if (joelOutput == 0)
    {
        outputRealPats (data, allClique s, L, OUTPUT_FILE, d);
    }
    else
    {
        outputRealPatsWCentroid (data, allClique s, L, OUTPUT_FILE, extraParams,
            compFunc);
    }

    freeD (d, supportDim);
    freeRdh (data);
    allClique s = popAllCll (allClique s);
    fclose (OUTPUT_FILE);
    return 0;
}
B.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 extraParams = (double *) malloc (numParams * sizeof (double));
117 if (extraParams == NULL)
118 {
119 fprintf (stderr, "Can’t allocate extra params!\n");
120 exit (0);
121 }
122 j = 0;
123 k = 0;
124 startLength = strlen (s);
125 for (i = 0; i < startLength; i++)
126 {
127 if (s[i] == ‘,’)
128 {
129 // We’ve found an end. So point the pointer to
130 // the beginning of the previous string.
131 paramString = &s[k];
132 // Terminate the string where the comma used to be
133 s[i] = ‘\0’;
134 // Update the location for the next string beginning
135 k = i + 1;
136 // Convert to a double and update the param number.
137 extraParams[j] = atof (paramString);
138 j++;
139 }
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 }

B.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().

```c
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           // and put its sequence number in the array of ints.  
448           seqNums[0] = indexToSeq[nextBit];  
449         }  
450       
451       // If it’s the last sequence, then bail out so that we don’t  
452       // segfault in the next step.  
453       if (seqNums[0] >= numOffSeqs - 1)  
454         {  
455           emptySet (bg->graph[i]);  
456           continue;  
457         }  
458       
459       // Find the next neighbor of this bit, STARTING AT the first  
460       // bit in the next sequence.  
461       nextBit =  
462         nextBitBitSet (bg->graph[i],  
463           offsetToIndex[indexToSeq[nextBit] + 1][0]);  
464       
465       // And iterate this until we run out of neighbors.  
466       while (nextBit >= 0)  
467         {  
468         }
```
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().

Usage: %s -i <Fasta sequence file> 
	-l <word size> 
	-k <support> -g <threshold> 
	-m <matrix name> [-z] 
	-c <cluster method [0|1]> [-p <unique support>] 

Required flags and input:

-i <Fasta sequence file>: File containing all sequences to be searched, in Fasta format.
-l <word size>: Minimum length of motifs; also the sliding window length
-k <support>: Minimum number of motif occurrences.
-g <threshold>: Similarity threshold. Two windows, when scored with the similarity matrix defined by
-m <matrix name>: similarity matrix defined by the -m flag, must have at least
"this score in order to be deemed 'connected'. This criterion\n" must be met over all sliding windows of length \n" 
"-c <cluster method [0|1]>: \n" The clustering method to be used after evaluating the 
"similarity of the unique words in the input. Note that the "clustering method will have a significant impact on both the "results that one obtains and the computation time.\n
"0: clique-finding\n"Uses established methods to find all maximal cliques in the "data. This will give the most thorough results (that are "provably exhaustive), but will also give less-significant "results in addition to the most interesting and most "significant ones. The results are deterministic but may take some "time on data sets with high similarity or if the similarity "threshold is set extremely low.\n"1: single-linkage clustering\n"Uses a single-linkage-type clustering where all nodes that "are connected are put in the same cluster. This method is "also deterministic and will be faster than clique-finding, "but it loses guarantees of exhaustiveness in searching the "data set.\n
"-p <unique support>: \n"A pruning parameter that requires the motif to occur in "at least <unique support> different input sequences. Note "that this parameter must be less than or equal to the total "support parameter set by the -k flag, argv[0]);

```c
```
B.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:

![Dependency Graph]

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)
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

B.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 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().

```c
90 {
91     bitGraph_t * sg = NULL;
92     int score;
93     int i, j;
94     // Assign a new bitGraph_t object, with (wc x wc) possible
95     // true/false values
96     sg = newBitGraph (wc);
97     for (i = 0; i < wc; i++)
98         for (j = i; j < wc; j++)
99             // Get the score for the alignment of word i and word j
```
score = 
    alignMat (words[i].string, words[j].string, words[i].length, mat);

    // If that score is greater than threshold, set
    // a bit to 'true' in our bitGraph_t object
    if (score >= threshold)
    {
        // We use 'bitGraphSetTrueSym' because, if i=j,
        // then j-i for most applications. However, this
        // can be relaxed for masochists.
        bitGraphSetTrueSym (sg, i, j);
    }

    // Return a pointer to this new bitGraph_t object
    return sg;

B.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(), findClique(), newBitSet(), pruneBitGraph(), pruneCll(), singleLinkage(), bitGraph_t::size, and yankCll().
// Now we set the main diagonal false for clustering and filtering.
bitGraphSetFalseDiagonal (bg);
filterGraph (bg, support, R);
fprintf (OUTPUT_FILE, "Graph filtered! Now clustering...\n");
fflush (NULL);
if (clusterMethod == 0)
{
    findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
}
else
{
    singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
}
printf (OUTPUT_FILE,
    "Clusters found! Now filtering clusters (if option set)...\n");
if (p > 1)
{
    elemPats = pruneCll (elemPats, indexToSeq, p);
}
deleteBitSet (cand);
deleteBitSet (mask);
deleteBitSet (Q);

// Now let's convolve what we made.
if (noConvolve == 0)
{
    printf (OUTPUT_FILE, "Now convolving...\n");
    fflush (NULL);
    allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
}
else
{
    curr = elemPats;
    while (curr != NULL)
    {
        yankCll (&elemPats, NULL, &curr, &allCliques, 0);
        curr = elemPats;
    }
    return allCliques;
}

B.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(), 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().
int i, j;
int totalChars = 0;
int hashSize;
sHashEntry_t newEntry;
sHashEntry_t *ep;
sHash_t wordHash;
sPat_t *words = NULL;
int wc = 0;
int prev = -1;
l;

// Count the total number of characters. This
// is the upper limit on how many words we can have
for (i = 0; i < numSeq; i++)
{
    totalChars += strlen(seq[i].seq);
}

// Get a prime number for the size of the hash table
hashSize = sieve3 ((long) (2 * totalChars));
wordHash = initSHash (hashSize);

// Chop up each sequence and hash out the words of size L
for (i = 0; i < numSeq; i++)
{
    prev = -1;

    // skip sequences that are too short to have
    // a pattern
    if (strlen(seq[i].seq) < L)
    {
        continue;
    }
    for (j = 0; j < strlen(seq[i].seq) - L + 1; j++)
    {
        // Make a hash table entry for this word
        newEntry.key = &(seq[i].seq[j]);
        newEntry.data = 1;
        newEntry.idx = wc;
        newEntry.L = L;

        // Check to see if it’s already in the hash table
        ep = searchSHash (&newEntry, &wordHash, 0);
        if (ep == NULL)
        {
            // If it’s not, create an entry for it
            ep = searchSHash (&newEntry, &wordHash, 1);
            // Increase the size of our word array
            words = (sPat_t *) realloc (words, (wc + 1) * sizeof (sPat_t));
            if (words == NULL)
            {
                fprintf (stderr, "Error!\n");
                fflush (stderr);
            }
            // Add the new word
            words[wc].string = &(seq[i].seq[j]);
            words[wc].length = L;
            words[wc].support = 1;
            words[wc].offset =
                (sOffset_t *) malloc (1 * sizeof (sOffset_t));
            if (words[wc].offset == NULL)
            {
                fprintf (stderr, "\nMemory Error\n", strerror (errno));
            }
        }
    }
}
```c
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    if (prev != -1)
450        {
451            words[prev].offset[words[prev].support - 1].next = wc;
452        }
453    prev = wc;
454    wc++;
455    }
456    else
457        {
458            // If it is, increase the count for this word
459            ep->data++;
460            // add a new offset to the word array
461            l = words[ep->idx].support;
462            words[ep->idx].offset = (sOffset_t *) realloc (words[ep->idx].offset,
463                (l + 1) * sizeof (sOffset_t));
464            words[ep->idx].offset[l].seq = i;
465            words[ep->idx].offset[l].pos = j;
466            words[ep->idx].offset[l].prev = prev;
467            words[ep->idx].offset[l].next = -1;
468            // Update the next/prev
469            if (prev != -1)
470                {
471                    words[prev].offset[words[prev].support - 1].next = ep->idx;
472                }
473            prev = ep->idx;
474            // Have to put this down here for cases when we create
475            // a word and it is immediately followed by itself!!
476            words[ep->idx].support += 1;
477        }
478    }
479    destroySHash (&wordHash);
480    *numWords = wc;
481    return words;
482 }
```

B.9.0.139  void getMatrixByName (char name[], int mat[][MATRIX_SIZE])

Referenced by main().

B.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(), calcStatAll-
Clqs(), convolve(), countWords2(), cumDMatrix(), deleteBitGraph(), FreeFSeqs(), getMatrix-
ByName(), 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 R = 1;
203 int sup = 2;
204 int P = 0;
205 int noConvolve = 0;
206 int j, k, i, l;
207 bitGraph_t * bg = NULL;
208 bitGraph_t * oam = NULL;
209
210 // new
211 int **offsetToIndex = NULL;
212 int *indexToSeq = NULL;
213 int *indexToPos = NULL;
214 int numberOfOffsets = 0;
215 int pos1, pos2;
216
217 // int *prevRowArray;
218 sOffset_t * offset1, *offset2;
219 cll_t * allClls = NULL;
220 cll_t * curCll = NULL;
221 int curSeg;
222 int curPos;
223 int clusterMethod = 0;
224
225 // patStats
226 int samp = 1;
227 unsigned int **d = NULL;
228 int supportDim = 0, lengthDim = 0;
229 int oamSize = 0;
230
231 /*
232     Get command-line options
233 */
while ((inputOption = getopt (argc, argv, "i:o:l:g:k:m:p:zc:ns:")) != EOF)
{
    switch (inputOption)
    {
    // Input file
    case 'i':
        sequenceFile = optarg;
        seqReadFunct = &ReadFSeqs;
        break;
    // Output file
    case 'o':
        outputFile = (char *) malloc ((strlen (optarg) + 1) * sizeof (char));
        if (outputFile == NULL)
        {
            fprintf (stderr, "Error allocating memory for options.
*");
            exit (EXIT_FAILURE);
        }
    else
    {
        strcpy (outputFile, optarg);
    }
    break;
    // Minimum motif length
    case 'l':
        L = atoi (optarg);
        break;
    // Minimum motif similarity score
    case 'g':
        g = atoi (optarg);
        status++;
        break;
    // Minimum support (number of motif occurrences)
    case 'k':
        sup = atoi (optarg);
        break;
    // Similarity matrix used to find similarity score
    case 'm':
        getMatrixByName (optarg, &mat);
        matName = (char *) malloc (strlen (optarg) * sizeof (char));
        if (matName == NULL)
        {
            fprintf (stderr, "Error allocating memory for options.
*");
            exit (EXIT_FAILURE);
        }
    else
    {
        strcpy (matName, optarg);
    }
    break;
/*
 * Recursive initial pruning: an option for clique finding.
 * It takes all nodes with less than the minimum
 * number of support and removes all of their nodes, and does this
 * recursively so that nodes that are connected to many sparsely connected
 * nodes will be removed and not left in the
 * This option is deprecated as it is at worst no-gain and at best useless.
 * It will be on by default for clique-finding, but can be turned
 * back off with some
 * minor tweaking. For almost all cases in which it does not speed
up computations, it will have a trivial time to perform. Thus, if clique-finding is turned on, then R is set to 1 by default.

```c
    case 'r'
    R = 1;
    break;
```

---

Optional pruning parameter to require at least P distinct input sequences

```c
    case 'p'
    P = atoi(optarg);
    break;
```

---

Clustering method.

```c
    case 'c'
    clusterMethod = atoi(optarg);
    break;
    case 'n'
    noConvolve = 1;
    break;
    case 's'
    samp = atoi(optarg);
    break;
```

---

Catch-all.

```c
    case '?'
    fprintf(stderr, "Unknown option '-%c'.\n", optopt);
    usage(argv);
    return EXIT_SUCCESS;
    case 'z'
    matrixlist();
    return EXIT_SUCCESS;
```

---

Require a similarity matrix

```c
    if (mat == NULL) {
        usage(argv);
        return EXIT_SUCCESS;
    }
```

---

Require an input file, a nonzero length, and a similarity threshold to be set.

```c
    if (sequenceFile == NULL || L == 0 || status < 1) {
        usage(argv);
        return EXIT_SUCCESS;
    }
```

---

Open the sequence file

```c
    if ((SEQUENCE_FILE = fopen(sequenceFile, "r")) == NULL) {
        fprintf(stderr, "Couldn't open file \s; \s\n", sequenceFile, strerror(errno));
        exit(EXIT_FAILURE);
    }
```

---

Open the output file

```c
    if (outputFile != NULL) {
        if ((OUTPUT_FILE = fopen(outputFile, "w")) == NULL) {
            fprintf(stderr, "Couldn't open file \s; \s\n", outputFile, strerror(errno));
            exit(EXIT_FAILURE);
        }
    }
```
else
{
    OUTPUT_FILE = stdout;
}
else

// Allocate some sequences
mySequences = seqReadFunct (SEQUENCE_FILE, &numberOfSequences);
if (mySequences == NULL)
{
    fprintf (stderr, "Error reading your sequences/text.");
    fprintf (stderr, "Check the format/size of the file.");
    fprintf (stderr, "ERROR: %s
", strerror (errno));
    return EXIT_FAILURE;
}

// Close the input files
fclose (SEQUENCE_FILE);

// Verbosity in output helps to distinguish output files.
fprintf (OUTPUT_FILE, "Matrix used = %s
", matName);
fprintf (OUTPUT_FILE, "Input file = %s
", sequenceFile);
fprintf (OUTPUT_FILE, "l = %d, k = %d, g = %d
", L, sup, g);
if (P > 1)
{
    fprintf (OUTPUT_FILE, "Minimum # of sequences with motif = %d
", P);
}
if (R > 0)
{
    fprintf (OUTPUT_FILE, "Recursive pruning is ON.
");
}

// Find the unique words in the input.
words = countWords2 (mySequences, numberOfSequences, L, &wc);

/*
fprintf(stderr, "Counted %d words
", wc);
*/
/*
fflush(stderr);
*/

// Align the words that we just found by applying the similarity
// matrix to each pair of them. Note that
// bg is the adjacency matrix of words, but we
// need an adjacency matrix of offsets instead.
bg = alignWordsMat_bit (words, wc, mat, g);

/*
fprintf(stderr, "Aligned! Creating offset matrix...
");
*/
fflush (NULL);

// Create an intermediate translation matrix
// to store the offset number of each sequence number/position.
// Note that
// this matrix is better called "Index to offset", and
// the other matrices are better called "offset to Seq" and
// "offset to Pos"
offsetToIndex = (int **) malloc (numberOfSequences * sizeof (int *));
if (offsetToIndex == NULL)
{
    fprintf (stderr, "Unable to allocate memory - offsetToIndex in gemoda.c\n\%s\n", strerror (errno));
    fflush (stderr);
    exit (0);
}
for (i = 0; i < numberOfSequences; i++)
{
439  // MPS 5/23/05: Added in "-L+2" to make there only be one
440  // blank between sequences.
441  offsetToIndex[i] =
442  malloc ((strlen (mySequences[i].seq) - L + 2) * sizeof (int));
443  if (offsetToIndex[i] == NULL)
444  {
445      fprintf (stderr,
446         "Unable to allocate memory - offsetToIndex[%d] in gemoda.c\n%s\n",
447             i, strerror (errno));
448      fflush (stderr);
449      exit (0);
450  }
451  
452  // MPS 5/23/05: Added in "-L+2" to make there only be one
453  // blank between sequences.
454  for (j = 0; j < (strlen (mySequences[i].seq) - L + 2); j++)
455  {
456      offsetToIndex[i][j] = numberOfOffsets;
457      numberOfOffsets++;
458  }
459
460  // Now create translation matrices such that we can get the sequence
461  // or position number of a given offset.
462  indexToSeq = (int *) malloc (numberOfOffsets * sizeof (int));
463  if (indexToSeq == NULL)
464  {
465      fprintf (stderr,
466         "Unable to allocate memory - indexToSeq in gemoda.c\n%s\n",
467             strerror (errno));
468      fflush (stderr);
469      exit (0);
470  }
471
472  indexToPos = (int *) malloc (numberOfOffsets * sizeof (int));
473  if (indexToPos == NULL)
474  {
475      fprintf (stderr,
476         "Unable to allocate memory - indexToPos in gemoda.c\n%s\n",
477             strerror (errno));
478      fflush (stderr);
479      exit (0);
480  }
481  
482  // Now make an offset adjacency matrix!
483  //
484  oam = newBitGraph ( numberOfOffsets );
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  // Now make an offset adjacency matrix!
496  //
497  oam = newBitGraph ( numberOfOffsets );
498
499  // Go through each unique word
500  for (i = 0; i < wc; i++)
501  {
502      offset1 = words[i].offset;
503  
504  // Go through each occurrence
505  for (k = 0; k < words[i].support; k++)
506  {
( // Use the offsetToIndex translation to get the offset
  // of the first occurrence
  pos1 = offsetToIndex[offset1[k].seq][offset1[k].pos];
  // And go through each word in the first offset to
  // find words that meet the similarity threshold
  for (j = 0; j < wc; j++)
  { if (bitGraphCheckBit (bg, i, j))
    offset2 = words[j].offset;
    // And find all of their occurrences,
    // using offsetToIndex to get the
    // offsets, and then setting those
    // locations in the offset adjacency
    // matrix true.
    for (l = 0; l < words[j].support; l++)
    { pos2 = offsetToIndex[offset2[l].seq][offset2[l].pos];
      bitGraphSetTrueSym (oam, pos1, pos2); }
  }
}
fprintf (OUTPUT_FILE, "Offset matrix created...");
deleteBitGraph (bg);
if ((samp > 0) && (clusterMethod == 0))
{ fprintf (OUTPUT_FILE, " taking preliminary statistics.\n");
  fflush (NULL);
  d = getStatMat (oam, sup, L, &supportDim, &lengthDim, numberOfSequences,
                   samp, OUTPUT_FILE);
  fprintf (OUTPUT_FILE, "Now filtering...\n");
  fflush (NULL);
}
else
{ fprintf (OUTPUT_FILE, " now filtering.\n");
  fflush (NULL);
  d = NULL;
  supportDim = 0;
}
// Now we're convolving on offsets
allCliques = convolve (oam, sup, R, indexToSeq, P, clusterMethod, offsetToIndex,
                       numberOfSequences, noConvolve, OUTPUT_FILE);
// Do some early memory cleanup to limit usage
oamSize = oam->size;
deleteBitGraph (oam);
fprintf (OUTPUT_FILE, "Convolved! Now making output...\n");
flush (NULL);
if ((samp > 0) && (clusterMethod == 0))
{ cumDMatrix (d, allCliques, supportDim, lengthDim, oamSize,
             numberOfSequences);
  calcStatAllClims (d, allCliques, numberOfOffsets - numberOfSequences);
  allCliques = sortByStats (allCliques);
}
// walk over the cliques and give some output in the format:
// pattern <pattern id num>: len=<motif length> sup=<motif instances>
575 // <sequence num> <position num> <motif instance>
576 // ...
577 curCliq = allCliques;
578 i = 0;
579 while (curCliq != NULL)
580 {
581     fprintf (OUTPUT_FILE, "pattern %d: len=%d sup=%d", i,
582             curCliq->length + L, curCliq->set->size);
583     if (d != NULL)
584         fprintf (OUTPUT_FILE, " signif=%le\n", curCliq->stat);
585     else
586         fprintf (OUTPUT_FILE, "\n");
587 }
588
589 for (j = 0; j < curCliq->set->size; j++)
590 {
591     pos1 = curCliq->set->members[j];
592     curSeq = indexToSeq[pos1];
593     curPos = indexToPos[pos1];
594     fprintf (OUTPUT_FILE, "%d\t%d\t", curSeq, curPos);
595     for (k = curPos; k < curPos + curCliq->length + L; k++)
596         fprintf (OUTPUT_FILE, "%c", mySequences[curSeq].seq[k]);
597     fprintf (OUTPUT_FILE, "\n");
598 }
599 curCliq = curCliq->next;
600 i++;
601 }
602
603 // And do some memory cleanup
604 // And cleanup of probability stuff...
605 /*
606 free(letterfreqs); delete_augmented_matrix(augmat);
607 */
608 allCliques = popAllCll (allCliques);
609 free (indexToSeq);
610 indexToSeq = NULL;
611 free (indexToPos);
612 indexToPos = NULL;
613 for (i = 0; i < numberOfSequences; i++)
614 {
615     free (offsetToIndex[i]);
616     offsetToIndex[i] = NULL;
617 }
618 // Free’ing added by MPS, 6/4
619 for (i = 0; i < wc; i++)
620 {
621     free (words[i].offset);
622 }
623 free (words);
624 // End free’ing added by MPS
625 free (offsetToIndex);
626 offsetToIndex = NULL;
627 // -------------------------------------------
628 // Free up fastaSequences
629 FreeFSeqs (mySequences, numberOfSequences);
630 fclose (OUTPUT_FILE);
631 return 0;
B.9. GEMODA-S.C FILE REFERENCE

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().

B.9.0.141 void matrixlist (void)

100 {
101     fprintf (stdout, "The following similarity matrices are installed ");
102     "with the default Gemoda installation. Most of these ");
103     "were obtained from publically available BLAST distributions. ");
104     "Identity matrix for DNA: returns 1 when A,C,G,T are ");
105     "compared to themselves, 0 otherwise. ");
106     "Identity matrix for amino acids: returns 1 when any ");
107     "letter but J,O,U are compared to themselves, and 0 ");
108     "Similar to identity_aa, but it returns 10 in place ");
109     "Similar to idmat, but it returns -10 in place of 0. ");
110     "PAM matrices for various evolutionary distances. ");
111     "Vanilla-flavored' pam250, very similar to pam250. ");
112     "BLOSUM matrix of unknown origin. ");
113     "BLOSUM-clustered scoring matrix with target frequency ");
114     "Persson-Argos clustering = (73,78,82)percent. ");
115     "From Ng, Henikoff, & Henikoff, Bioinformatics 16: 760. ");
116     "coil_mat: ");
117     "alpha_mat: ");
118     "beta_mat: ");
119     "Three structure-specific matrices described by Luthy. ");
120     fprintf (stdout, "");
121 }

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().

```c
int i = 0, j = 0, nextBit = 0;
int *seqNums = NULL;

// Since we don’t immediately know which node is in which source sequence, we can’t just count them up regularly. Instead, we’ll need to keep track of which sequences they come from and // increment _something_. What we chose to do here is just make // an array of integers of length = <p>. Then, we try to put the // source sequence number of each neighbor (including itself, since // the main diagonal is still true at this time) into the next slot // Since we will monotonically search the bitSet, we can just // move on to the first bit in the next sequence using the // offsetToIndex structure so that we know the next sequence number // to be put in is always unique.
seqNums = (int *) malloc (p * sizeof (int));
if (seqNums == NULL)
{
    fprintf (stderr, "Memory error - pruneBitGraph
%s
", strerror (errno));
    fflush (stderr);
    exit (0);
}

// So, for each row in the bitgraph...
for (i = 0; i < bg->size; i++)
{
    // Make sure the whole array is -1 sentinels.
    for (j = 0; j < p; j++)
    {
        seqNums[j] = -1;
    }
    j = 0;

    // Find the first neighbor of this bit.
    nextBit = nextBitBitSet (bg->graph[i], 0);
    if (nextBit == -1)
    {
        continue;
    }
    else
    {
        // and put its sequence number in the array of ints.
        seqNums[0] = indexToSeq[nextBit];
    }
```

// If it’s the last sequence, then bail out so that we don’t
// segfault in the next step.
if (seqNums[0] >= numOfSeqs - 1)
{
    emptySet (bg->graph[i]);
    continue;
}

// Find the next neighbor of this bit, STARTING AT the first
// bit in the next sequence.
nextBit =
nextBitBitSet (bg->graph[i],
    offsetToIndex[indexToSeq[nextBit] + 1][0]);

// And iterate this until we run out of neighbors.
while (nextBit >= 0)
{
    j++;
    seqNums[j] = indexToSeq[nextBit];

    // Or until this new neighbor will fill up the array
    if (j == p - 1)
    {
        break;
    }

    // Or until this new neighbor is in the last sequence.
    if (seqNums[j] >= numOfSeqs - 1)
    {
        break;
    }

    // Get the next neighbor!
    nextBit =
    nextBitBitSet (bg->graph[i],
        offsetToIndex[indexToSeq[nextBit] + 1][0]);
}

// If we didn’t have enough unique sequences, and either a) we
// were in the nth-to-last sequence and there were no
// neighbors after it, or b) we were in the last sequence,
// then the last number will still be our sentinel, -1. If
// the last number is not a sentinel, then we have at least
// p distinct sequence occurrences, so we’re OK.
if (seqNums[p - 1] == -1)
{
    emptySet (bg->graph[i]);
}
free (seqNums);
return (bg);

B.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.
fprintf (stdout, "Usage: %s -i <Fasta sequence file> 
"  "-1 <word size> \n\t-k <support> -g <threshold>\n"  "-m <matrix name> [-z] \n\t-c <cluster method [0|1]>\n"  "\t-p <unique support>\n\n"  "Required flags and input:\n\n"  "-i <Fasta sequence file>:\n\n"  "-l <word size>:\n\n"  "Minimum length of motifs; also the sliding window length\n\n"  "over which all motifs must meet the similarity criterion\n\n"  "-k <support>:\n\n"  "Minimum number of motif occurrences.\n\n"  "-g <threshold>:\n\n"  "Similarity threshold. Two windows, when scored with the\n\n"  " similarity matrix defined by the -m flag, must have at least\n\n"  " this score in order to be deemed 'connected'. This criterion\n\n"  " must be met over all sliding windows of length l.\n\n"  "-m <matrix name>:\n\n"  "Name of the similarity matrix to be used to compare windows.\n\n"  "Use -z to see a list of matrices installed by default.\n\n"  "Optional flags and input:\n\n"  "-z:\n\n"  "Lists all of the similarity matrices available with the\n\n"  " initial installation of Gemoda. Note that this overrides\n\n"  " all other options and will only give this output.\n\n"  "-c <cluster method [0|1]>:\n\n"  "The clustering method to be used after evaluating the\n\n"  " similarity matrix of the unique words in the input. Note that the\n\n"  " clustering method will have a significant impact on both the\n\n"  " results that one obtains and the computation time.\n\n"  "0: clique-finding\n\n"  " USES established methods to find all maximal cliques in the\n\n"  " data. This will give the most thorough results (that are\n\n"  " probably exhaustive), but will also give less-significant\n\n"  " results in addition to the most interesting and most\n\n"  " significant ones. The results are deterministic but may take some\n\n"  " time on data sets with high similarity or if the similarity\n\n"  " threshold is set extremely low.\n\n"  "1: single-linkage clustering\n\n"  " USES a single-linkage-type clustering where all nodes that\n\n"  " are connected are put in the same cluster. This method is\n\n"  " also deterministic and will be faster than clique-finding,\n\n"  " but it loses guarantees of exhaustiveness in searching the\n\n"  " data set.\n\n"  "-p <unique support>:\n\n"  " A pruning parameter that requires the motif to occur in\n\n"  " at least <unique support> different input sequences. Note\n\n"  " that this parameter must be less than or equal to the total\n\n"  " support parameter set by the -k flag.\n\n"  argv[0]);

}
B.10  matdata.h File Reference

This graph shows which files directly or indirectly include this file:

![Graph showing file inclusion]

**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**

B.10.0.144  `#define MATRIX_SIZE 23`

Definition at line 10 of file `matdata.h`.

Referenced by `main()`.
# B.11 matrices.c File Reference

```c
#include <stdio.h>
#include <string.h>
#include "matdata.h"
#include "matrixmap.h"
```

Include dependency graph for matrices.c:

![Dependency Graph](image)

## 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

### B.11.0.145 #define DEFAULT_MATRIX blosum62

Definition at line 7 of file matrices.c. Referenced by getMatrixByDate().
Function Documentation

B.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       break;
41   }
42   if (matrix_map[i].name != NULL)
43     *matp = (matrix_map[i].mat);
44   else
45     *matp = (DEFAULT_MATRIX);
46 }
50 }
B.12 matrices.h File Reference

This graph shows which files directly or indirectly include this file:

```
matrices.h -> matrixmap.h -> matrices.c
```

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]
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 $256 \times 256$ matrices. By changing aaOrder, you can easily make matrices for comparing ANSI 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

**B.12.0.147 const int aaOrder[]**
Definition at line 32 of file matrices.h. Referenced by alignMat().

**B.12.0.148 const int alpha_mat[MATRIX_SIZE][MATRIX_SIZE]**
Definition at line 1398 of file matrices.h.

**B.12.0.149 const int beta_mat[MATRIX_SIZE][MATRIX_SIZE]**
Definition at line 1422 of file matrices.h.

**B.12.0.150 const int blosum100[MATRIX_SIZE][MATRIX_SIZE]**
Definition at line 126 of file matrices.h.

**B.12.0.151 const int blosum30[MATRIX_SIZE][MATRIX_SIZE]**
Definition at line 150 of file matrices.h.

**B.12.0.152 const int blosum35[MATRIX_SIZE][MATRIX_SIZE]**
Definition at line 174 of file matrices.h.

**B.12.0.153 const int blosum40[MATRIX_SIZE][MATRIX_SIZE]**
Definition at line 198 of file matrices.h.

**B.12.0.154 const int blosum45[MATRIX_SIZE][MATRIX_SIZE]**
Definition at line 222 of file matrices.h.
B.12.0.155  const int blosum50[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 246 of file matrices.h.

B.12.0.156  const int blosum55[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 270 of file matrices.h.

B.12.0.157  const int blosum60[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 294 of file matrices.h.

B.12.0.158  const int blosum62[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 318 of file matrices.h.

B.12.0.159  const int blosum65[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 342 of file matrices.h.

B.12.0.160  const int blosum70[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 366 of file matrices.h.

B.12.0.161  const int blosum75[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 390 of file matrices.h.

B.12.0.162  const int blosum80[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 414 of file matrices.h.

B.12.0.163  const int blosum85[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 438 of file matrices.h.

B.12.0.164  const int blosum90[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 462 of file matrices.h.
B.12.0.165  const int blosumn[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 486 of file matrices.h.

B.12.0.166  const int coil_mat[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1446 of file matrices.h.

B.12.0.167  const int dayhoff[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 510 of file matrices.h.

B.12.0.168  const int dna_idmat[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 50 of file matrices.h.

B.12.0.169  const int identity_aa[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 76 of file matrices.h.

B.12.0.170  const int idmat[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 101 of file matrices.h.

B.12.0.171  const int pam100[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 534 of file matrices.h.

B.12.0.172  const int pam110[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 558 of file matrices.h.

B.12.0.173  const int pam120[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 582 of file matrices.h.

B.12.0.174  const int pam130[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 606 of file matrices.h.
B.12. MATRICES.H FILE REFERENCE

B.12.0.175  const int pam140[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 630 of file matrices.h.

B.12.0.176  const int pam150[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 654 of file matrices.h.

B.12.0.177  const int pam160[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 678 of file matrices.h.

B.12.0.178  const int pam190[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 702 of file matrices.h.

B.12.0.179  const int pam200[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 726 of file matrices.h.

B.12.0.180  const int pam210[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 750 of file matrices.h.

B.12.0.181  const int pam220[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 774 of file matrices.h.

B.12.0.182  const int pam230[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 798 of file matrices.h.

B.12.0.183  const int pam240[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 822 of file matrices.h.

B.12.0.184  const int pam250[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 846 of file matrices.h.
B.12.0.185  const int pam260[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 870 of file matrices.h.

B.12.0.186  const int pam280[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 894 of file matrices.h.

B.12.0.187  const int pam290[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 918 of file matrices.h.

B.12.0.188  const int pam300[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 942 of file matrices.h.

B.12.0.189  const int pam310[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 966 of file matrices.h.

B.12.0.190  const int pam320[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 990 of file matrices.h.

B.12.0.191  const int pam330[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1014 of file matrices.h.

B.12.0.192  const int pam340[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1038 of file matrices.h.

B.12.0.193  const int pam360[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1062 of file matrices.h.

B.12.0.194  const int pam370[MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1086 of file matrices.h.
B.12.0.195  const int pam380 [MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1110 of file matrices.h.

B.12.0.196  const int pam390 [MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1134 of file matrices.h.

B.12.0.197  const int pam400 [MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1158 of file matrices.h.

B.12.0.198  const int pam430 [MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1182 of file matrices.h.

B.12.0.199  const int pam440 [MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1206 of file matrices.h.

B.12.0.200  const int pam450 [MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1230 of file matrices.h.

B.12.0.201  const int pam460 [MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1254 of file matrices.h.

B.12.0.202  const int pam490 [MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1278 of file matrices.h.

B.12.0.203  const int pam500 [MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1302 of file matrices.h.

B.12.0.204  const int phat_t75_b73 [MATRIX_SIZE][MATRIX_SIZE]
Definition at line 1326 of file matrices.h.
B.12.0.205 \texttt{const int phat\_t80\_b78}[\texttt{MATRIX\_SIZE}][\texttt{MATRIX\_SIZE}]

Definition at line 1350 of file matrices.h.

B.12.0.206 \texttt{const int phat\_t85\_b82}[\texttt{MATRIX\_SIZE}][\texttt{MATRIX\_SIZE}]

Definition at line 1374 of file matrices.h.
B.13 matrixmap.h File Reference

#include "matdata.h"
#include "matrices.h"

Include dependency graph for matrixmap.h:

```
matrixmap.h
matdata.h
matrices.h
```

This graph shows which files directly or indirectly include this file:

```
matrixmap.h
matrices.c
```

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

B.13.0.207 const int(* mat)[MATRIX_SIZE]

Definition at line 15 of file matrixmap.h. Referenced by alignMat(), alignWordsMat_bit(), and main().

B.13.0.208 struct { ... } matrix_map[]

This data structure maps the names of common matrices to the names of their variables. Referenced by getMatrixByName().
B.13.0.209  char* name

Definition at line 14 of file matrixmap.h.
B.14 newConv.c File Reference

#include "bitset.h"
#include <errno.h>
#include "convll.h"

Include dependency graph for newConv.c:

```
newConv.c -> bitSet.h -> errno.h -> convll.h
```

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 numSeqs, 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

B.14.0.210 cll_t* convolve (bitGraph_t * bg, int support, int R, int * indexToSeq, int p, int clusterMethod, int **ffsetToIndex, 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().

```c
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, p);
664     }
665   fprintf (OUTPUT_FILE,
```
Clusters found! Now filtering clusters (if option set)...

```
667 fflush (NULL);
668 if (p > 1)
669   {  
670     elemPats = pruneCll (elemPats, indexToSeq, p);
671   }
672 deleteBitSet (cand);
673 deleteBitSet (mask);
674 deleteBitSet (Q);
675 // Now let’s convolve what we made.
676 if (noConvolve == 0)
677   {  
678     fprintf (OUTPUT_FILE, "Now convolving...\n");
679     fflush (NULL);
680     allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
681   }  
682 else
683   {  
684       curr = elemPats;
685       while (curr != NULL)
686         {  
687           yankCll (&elemPats, NULL, &curr, &allCliques, 0);
688         }
689       return allCliques;
690   }
90  }

B.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 1 (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);
```
emptySet (work);

// Iteratively call the filtering by copying the previous "work" into
// "changed" after each iteration step.
if (R == 1) {
    do {
        filterIter (graph, support, changed, work);
        copySet (work, changed);
    } while (countSet (changed) > 0);
} else {
    // Otherwise, just do it once.
    filterIter (graph, support, changed, work);
}

deleteBitSet (changed);
defineBitSet (work);
return 0;

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 set-
true().

Referenced by filterGraph().

{ int i = 0, j = 0;
  int lastBit = 0, nextBit = 0, lastRow = 0, nextRow = 0;
  int numNodes = 0;
  int changedSize = countSet (changed);
  emptySet (work);

  // Note the convention that the first time the function is called,
  // it is done with an empty "changed" bitSet as a sentinel. It is
  // the responsibility of the master function calling the iterator
  // to catch future empty changed sets to know that convergence has
  // been achieved.
  //
  // 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 // Otherwise, zero it out, but going one by
257 // one so that you can also zero out the
258 // symmetric bit.
259 lastBit = 0;
260 for (j = 0; j < numNodes; j++)
261 {
262 nextBit = nextBitBitSet (graph->graph[i], lastBit);
263 if (nextBit == -1)
264 {
265 fprintf (stderr, "\nEnd of bitSet reached! - initial\n");
266 fflush (stderr);
267 exit (0);
268 }
269 setFalse (graph->graph[i], nextBit);
270 setFalse (graph->graph[nextBit], i);
271 // And set that corresponding bit true
272 // in the work bitSet so that we
273 // know we changed it for the next
274 // round.
275 setTrue (work, nextBit);
276 lastBit = lastBit + 1;
277 }
278 else
279 {
280 // Otherwise, we’ve been here before, so just follow what
281 // the changed bitSet says to do... only those bitSets that
282 // were changed could possibly have gone under the minimum
283 // support requirement.
284 lastRow = 0;
285 for (i = 0; i < changedSize; i++)
286 {
287 nextRow = nextBitBitSet (changed, lastRow);
288 if (nextRow == -1)
289 {
290 fprintf (stderr, "\nEnd of bitSet reached! - iter,row\n");
291 fflush (stderr);
292 exit (0);
293 }
294 else
295 {
296 // So now we’ve found the row that needs to be checked.
297 // We do the same thing we did above... either move
298 // on if it has enough possible support, or zero
299 // it out (with its symmetric locations) one by one.
300 numNodes = countSet (graph->graph[nextRow]);
301 if (numNodes >= support - 1)
302 {
303 lastRow = nextRow + 1;
304 continue;
305 }
306 }
307 }
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 indicating 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 1 (but more importantly, the elemPats clique linked list is expanded to contain all elementary (minimum-length) motif cliques.

Definition at line 36 of file newConv.c.

References bitSetIntersection(), checkBit(), countSet(), deleteBitSet(), bitGraph_t::graph, newBitSet(), nextBitBitSet(), pushClique(), setFalse(), setTrue(), and bitGraph_t::size.

Referenced by convolve().

```c
int findClique(bitSet_t * Q, bitSet_t * cand, bitSet_t * mask, bitGraph_t * oG, int support, int qCount, cll_t ** elemPats, int * indexToSeq, int p)
```
int numBits;
int u = 0;
int newMaskCount;
int candQCount;

graphSize = oG->size;

// Find which vertex in subg maximizes \[|cand \cap \gamma(u)|\]
gammaOG = oG->graph;
for (i = 0; i < graphSize; i++) {
    if (!(checkBit (mask, i)))
        continue;
    bitSetIntersection (gammaOG[i], cand, candQ);
    numBits = countSet (candQ);
    if (numBits > max)
        u = i;
    max = numBits;
}

// Then do the extension of the q's
qCount++;

// This loop iterates over all possible values of cand - gamma() by
// iterating over all possible values of cand but immediately
// "continue"ing if the node is also in gamma(u)
q = nextBitBitSet (cand, 0);
while (q != -1) {
    if (checkBit (gammaOG[u], q))
        continue;
    bitSetIntersection (mask, gammaOG[q], newMask);
    newMaskCount = countSet (newMask);
    setTrue (Q, q);
    if (newMaskCount > 0 && qCount + newMaskCount >= support)
        findCliques (Q, candQ, newMask, oG, support, qCount, elemPats, indexToSeq, p);
}
APPENDIX B. GEMODA FILE DOCUMENTATION

115 }  
116 else if (qCount >= support)  
117 {  
118 // This should be done when:  
119 // 1. countSet(newMask) == 0 [connected subgraph is maximal]  
120 // 2. Qcount >= minCount [connected subgraph has enough nodes]  
121 *elemPats = pushClique (Q, *elemPats, indexToSeq, p);  
122 }  
123  
124 // Remove q from Q, and remove q from cand  
125 setFalse (Q, q);  
126 setFalse (cand, q);  
127 q = nextBitBitSet (cand, q + 1);  
128 }  
129 qCount--;  
130 deleteBitSet (candQ);  
131 deleteBitSet (newMask);  
132 return 0;  
133 }

B.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",  
423 strerror (errno));  
424 fflush (stderr);  
425 exit (0);  
426 }
// So, for each row in the bitgraph...
for (i = 0; i < bg->size; i++) {
    // Make sure the whole array is -1 sentinels.
    for (j = 0; j < p; j++)
        seqNums[j] = -1;
    j = 0;

    // Find the first neighbor of this bit.
    nextBit = nextBitBitSet (bg->graph[i], 0);
    if (nextBit == -1)
        continue;
    else
        seqNums[0] = indexToSeq[nextBit];

    // If it’s the last sequence, then bail out so that we don’t
    // segfault in the next step.
    if (seqNums[0] >= numOfSeqs - 1)
        emptySet (bg->graph[i]);
    continue;
}

// Find the next neighbor of this bit, STARTING AT the first
// bit in the next sequence.
nextBit = nextBitBitSet (bg->graph[i],
                        offsetToIndex[indexToSeq[nextBit] + 1][0]);

// And iterate this until we run out of neighbors.
while (nextBit != 0)
    j++;
    seqNums[j] = indexToSeq[nextBit];

    // Or until this new neighbor will fill up the array
    if (j == p - 1)
        break;

    // Or until this new neighbor is in the last sequence.
    if (seqNums[j] >= numOfSeqs - 1)
        break;

    // Get the next neighbor!
    nextBit = nextBitBitSet (bg->graph[i],
                            offsetToIndex[indexToSeq[nextBit] + 1][0]);

    // If we didn’t have enough unique sequences, and either a) we
    // were in the nth-to-last sequence and there were no
    // neighbors after it, or b) we were in the last sequence,
    // then the last number will still be our sentinel, -1. If
    // the last number is not a sentinel, then we have at least
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().
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 deference 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, nextBitBit-
Set(), 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 }
else
{
    return 0;
}
return 0;


B.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**

**B.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 }
```

**B.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     // OK, so support is at least two. So we make the connections all
639     // on the first level, knowing that each node being connected has
640     // at least zero in common. There are [(size of cluster) - 1] of
641     // these connections to be made.
642     // 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 }

B.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 for (i = 0; i < currSupport + 1; i++)

// We no longer need to do that, since now we sum across both
// the support and the length dimensions. Now, d[support][0] will
// necessarily include d[support][1] being added to it. We don’t
// want to add this anymore, otherwise we would be underestimating
// the probability of making that first connection. For instance,
// if there were no nodes with 20 in common that weren’t also
// connected, and no nodes whatsoever with more than 20 in common,
// we’d want the p[20][0] to be 1, which would be
// d[20][1]/d[20][0]. When summing across length directions,
// this happens naturally, whereas before we needed to do it
// artificially as per above. If we did above, we’d have the
// probability of each node being 1/2 instead of 1.

// Rather than storing doubles and doing lots of multiplications,
// we’re going to limit the number of operations done in the actual
// probability calculation by only storing cumulative sums in d.
// Now remember, what we’re storing at each location is the
// number of nodes with [i] or more nodes in common (including
// each other and selves) that can be extended [j] times (with
// their initial similarity counting as 1).

// We go up to the last possible index in the length direction, which
// means going up to [maxLen]. We know that this is legitimate
// because maxLen is less than or equal to the longest possible
// diagonal, and the longest possible diagonal will be less
// than or equal to currLength. Since we have allotted
// (currLength + 1) integers, we know we’re OK to access [currLength].
for (j = 0; j < currLength + 1; j++)
{

// We start at currSupport - 1, because currSupport will
// clearly not be changed, and this makes it a much easier
// loop to read.
for (i = currSupport - 1; i >= 0; i--)
{
    d[i][j] += d[i + 1][j];
}
}

// Now we need to forcibly set d[0][0] to its correct value... it’s
// just the total number of comparisons, not including comparisons
// to delimiter 0’s meant to separate sequences. The number of
// windows is equal to the number of offsets minus the number
// of sequences (assuming one delimiter per sequence). We don’t count
// the main diagonal, so the first row has one less, and we want to
// sum over all the subsequent rows in the upper half of the matrix.
// So it’s (numWins - 1)*(numWins - 1 + 1)/2 to sum that up.
numWins = bgSize - numSeqs;
d[0][0] = numWins * (numWins - 1) / 2;

/*
for (i = 0; i <= maxSup; i++) {
    printf("support = %d:\t",i);
    for (j = 0; j <=
        maxLen; j++) {
        printf("%d\t",d[i][j]);
    } printf("n");
} */

return 1;
}
B.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      { // Still, it’s supportDim + 1, because we have an extra
693       if (d == 0)
694         return 0;
695          }
696    }
697    else
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

B.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          { len = curCliq->length;
53            curCliq = curCliq->next;
54          }
55      }
56    }
57    // We return (len + 1) because the length of the shortest streak
58    // is one, but is stored in the cluster data structure as being
59    // zero (number of extensions that have been made).
60    return (len + 1);
B.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().

B.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().
Q = (int *) malloc (bg->size * sizeof (int));
if (Q == NULL)
{
    fprintf (stderr,
        "\nMemory error --- couldn't allocate array!" "\n%s\n",
        strerror (errno));
    fflush (stderr);
    exit (0);
}
for (i = 0; i < bg->size; i++)
{
    Q[i] = 0;
}
d = (unsigned int **) malloc ((currSupport + 1) * sizeof (unsigned int *));
if (d == NULL)
{
    fprintf (stderr,
        "\nMemory error --- couldn't allocate array!" "\n%s\n",
        strerror (errno));
    fflush (stderr);
    exit (0);
}
for (i = 0; i < currSupport + 1; i++)
{
    d[i] = (unsigned int *) malloc ((currLength + 1) * sizeof (unsigned int));
    if (d[i] == NULL)
    {
        fprintf (stderr, "\nMemory error --- couldn't allocate array!
" "\n%s\n",
            strerror (errno));
        fflush (stderr);
        exit (0);
    }
    for (j = 0; j < currLength + 1; j++)
    {
        d[i][j] = 0;
    }
}
// printf("size=%d\n",bg->size);
time (&probStart);
for (i = 0; i < bg->size; i++)
{
    if (i == 200)
    {
        time (&probEnd);
        timeNeeded = ((double) (probEnd - probStart)) / 
            ((double) 60) * ((double) bg->size) / ((double) 200);
        if (timeNeeded > 2)
        {
            fprintf (OUTPUT_FILE,
"\nMax total time to calculate probability: \n\n");  
            fprintf (OUTPUT_FILE, "\n%d minutes\n", timeNeeded);
            fprintf (OUTPUT_FILE, "\nActual time will be less than this, \n\n" "but at least half of it.\n\n"");
            fprintf (OUTPUT_FILE, "\nTo bypass excessive probability calculations,\n" "cancel and use a different value\n" "for the '-s' flag (samples every \n" "'s' points).\n\n"");
            fflush (NULL);
        }
    }
    j = nextBitBitSet (bg->graph[i], 0);
    while (j >= 0)
    {
        k = nextBitBitSet (bg->graph[i], j + 1);
        }
while (k >= 0)
{
    if (checkBit (bg->graph[j], k) == 0)
    {
        if (sampleCounter == s)
        {
            bitGraphRowIntersection (bg, j, k, X);
            // visitCounter++;
            if (nextBitBitSet (X, 0) >= i)
            {
                // uniqCounter++;
                x = countSet (X);
                while (x > currSupport)
                {
                    d =
                    increaseMem (d, 1, currSupport, currLength,
                    currSupport +
                    support * multiplier);
                    currSupport += support * multiplier;
                }
                d[x][0] += 1;
                sampleCounter = 0;
                sampleCounter++;
            }
            sampleCounter = 0;
        }
        sampleCounter++;
        k = nextBitBitSet (bg->graph[i], k + 1);
    }
    if (j <= i)
    {
        j = nextBitBitSet (bg->graph[i], j + 1);
        continue;
    }
    bitGraphRowIntersection (bg, i, j, X);
    x = countSet (X);

    // Note, now we’re using "diagonals" rather than
    // location in a horizontal array. So you always
    // start from the main diagonal at 0 and move out.
    diagonal = j - i;

    // We change this to greater-than-one because
    // after Q[diagonal] is reduced to one, it isn’t
    // visited again until we reach a new streak, (because
    // the next bit in the diagonal is a zero), and at
    // that point we want to start with a new diagonal
    // measure.
    if (Q[diagonal] > 1)
    {
        y = Q[diagonal] - 1;
        Q[diagonal]--;
    }
    else
    {
        y = measureDiagonal (bg, i, j);
        Q[diagonal] = y;
    }
    while (x > currSupport)
    {
        d = increaseMem (d, 1, currSupport, currLength,
        currSupport + support * multiplier);
        currSupport += support * multiplier;
    }
    while (y > currLength)
    {
        d =
increaseMem (d, 2, currSupport, currLength,
    currLength + length * multiplier);
currLength += length * multiplier;
}

d[x][y]++;
j = nextBitBitSet (bg->graph[i], j + 1);
/*
  if(x != 0){ printf("%d:	%d %d
", j, x, y); fflush(stdout); }
*/
*/
printf("done\n"); fflush(stdout);
*/
}

/*
printf("done\n"); fflush(stdout);
*/

// We need to rescale by the sampling factor for all i>0 in d[i][0].
//
for (i = 1; i < currSupport; i++)
{
    d[i][0] *= s;
}

// Now we only need to assign the correct value for d[0][0]...
// but rather than figuring that out, we will just assign it in the
// cumulative function, since there it is merely the number of unique
// non-self comparisons and is easy to calculate.
deleteBitSet (X);
free (Q);
*supportDim = currSupport;
*lengthDim = currLength;
return (d);
}

B.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().

  int i = 0, j = 0;
  if (dimToChange == 1)
  {
      d = (unsigned int **) realloc (d, (newVal + 1) * sizeof (unsigned int *));
      if (d == NULL)
      {
          fprintf (stderr, "\nMemory error --- couldn’t allocate array!
          "\n\n", strerror (errno));
          fflush (stderr);
          exit (0);
      }
      for (i = currSupport + 1; i < newVal + 1; i++)
B.15. PATSTATS.C FILE REFERENCE

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\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   return d;
124 }
125 else if (dimToChange == 2)
126     {   
127       for (i = 0; i < currSupport + 1; i++)
128         {   
129           d[i] =
130           (unsigned int *) realloc (d[i],
131           (newVal + 1) * sizeof (unsigned int));
132           if (d[i] == NULL)
133             {   
134               fprintf (stderr,
135                  "\nMemory error --- couldn’t allocate array!"
136                  "\n\n", strerror (errno));
137               fflush (stderr);
138               exit (0);
139             }
140           for (j = currLength + 1; j < newVal + 1; j++)
141             {   
142               d[i][j] = 0;
143             }
144         }
145       return d;
146     }
147 else
148     {   
149       fprintf (stderr, "Invalid arguments to increaseMem!\n\n");
150       fflush (stderr);
151       exit (0);
152     }
153 }

B.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().
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~20 and K~20, the memory used shoots down to ~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>. 
```c
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", strerror (errno));
219        fflush (stderr);
220        exit (0);
221      }
222      for (i = 0; i < bg->size; i++)
223      {
224        Q[i] = 0;
225      }
226  d =
227     (unsigned int **) malloc ((currSupport + 1) * sizeof (unsigned int *));
228  if (d == NULL)
229      {
230        fprintf (stderr,
231               "\nMemory error --- couldn't allocate array!\n", strerror (errno));
232        fflush (stderr);
233        exit (0);
234      }
235      for (i = 0; i < currSupport + 1; i++)
236      {
237        d[i] =
238        (unsigned int *) malloc ((currLength + 1) * sizeof (unsigned int));
239        if (d[i] == NULL)
240            {
241              fprintf (stderr, "\nMemory error --- couldn't allocate array!\n", strerror (errno));
242              fflush (stderr);
243              exit (0);
244            }
245            for (j = 0; j < currLength + 1; j++)
246            {
247              d[i][j] = 0;
248            }
249      }
250  time (&probStart);
251  for (i = 0; i < bg->size; i++)
252      {
253        if (i == 200)
254            {
255                time (&probEnd);
256                timeNeeded = ((double) (probEnd - probStart)) / }
257                ((double) 60) * ((double) bg->size) / ((double) 200);
258        if (timeNeeded > 2)
259            {
260                printf ("Max total time to calculate probability:\n");
261                printf ("\n\t%d minutes
", timeNeeded);
262                printf ("Actual time will be less than this, but at",
263                       "least half of it.\n");
264                printf ("To bypass excessive probability calculations,",
265                "cancel and use the '-d' flag.\n");
266                fflush (NULL);
267            }
268            for (j = bg->size - 1; j > i; j--)
269            {
270                bitGraphRowIntersection (bg, i, j, X);
271            }
275 \( x = \text{countSet}(X); \)
276 if (Q[j - 1] != 0)
277 {
278     y = Q[j - 1] - 1;
279     Q[j] = Q[j] - 1;
280 } else {
281     y = \text{measureDiagonal}(bg, i, j);
282     Q[j] = y;
283 } while (x > currSupport)
284 {
285     d = \text{increaseMem}(d, 1, currSupport, currLength,
286         currSupport + support * multiplier);
287     currSupport += support * multiplier;
288 } while (y > currLength)
289 {
290     d = \text{increaseMem}(d, 2, currSupport, currLength,
291         currLength + length * multiplier);
292     currLength += length * multiplier;
293 } d[x][y]++;
294 /*
295     if (x != 0) { printf("%d:	%d %d
", j, x, y); fflush(stdout); } */
296 /*
297     printf("done\n"); fflush(stdout); */
298 */
299 // We know that the "blanks", inserted to delimit unique sequences
300 // and prevent convolution through them, will skew our statistics,
301 // so we subtract them. We know that they will never be similar to
302 // any others, so will only add to the d[0][0] number. Furthermore,
303 // we know how many they add. Since d never hits the main diagonal
304 // and only does the upper half of the matrix, the first one
305 // contributes bsize - 1 to d[0][0], the next bsize - 2, etc.
306 for (i = 0; i < numBlanks; i++)
307 {
308     d[0][0] -= bg->size - 1 - i;
309 } deleteBitSet (X);
310 free (Q);
311 \*supportDim = currSupport;
312 \*lengthDim = currLength;
313 return (d);
314 }

**B.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().
cll_t * curCliq = NULL;
cll_t ** arrayOfCliqs = NULL;
int numOfCliqs = 0;
int i = 0;
curCliq = allCliqs;
if (curCliq != NULL)
{
    numOfCliqs = curCliq->id + 1;
}
elsereturn (NULL);
arrayOfCliqs = (cll_t **) malloc (numOfCliqs * sizeof (cll_t *));
for (i = 0; i < numOfCliqs; i++)
{
    arrayOfCliqs[i] = curCliq;
curCliq = curCliq->next;
}
qsort (arrayOfCliqs, numOfCliqs, sizeof (cll_t *), statCompare);
for (i = 0; i < numOfCliqs - 1; i++)
{
    arrayOfCliqs[i]->next = arrayOfCliqs[i + 1];
}
arrayOfCliqs[numOfCliqs - 1]->next = NULL;
return (arrayOfCliqs[0]);

---

int statCompare (const cll_t ** first, const cll_t ** second)

Definition at line 709 of file patStats.c.

Referenced by sortByStats().

double difference = (*first)->stat - (*second)->stat;
if (difference < 0)
    return (-1);
else if (difference > 0)
    return (1);
else
    return (0);
B.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

B.16.0.229  int calcStatAllCliqs (unsigned int ** d, cll_t * allCliqs, int numWindows)

Definition at line 623 of file patStats.c.
References calcStatClique(), cnode::next, and cnode::stat.
Referenced by main().

B.16.0.230  int cumMDMatrix (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().

B.16.0.231  int freeD (unsigned int ** d, int supportDim)

Definition at line 637 of file patStats.c.
Referenced by main().

B.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().

B.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().
B.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 *extraParams)
- 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 *extra-Params)
- 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

**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
int i, j;
double numerator = 0.0;

/*
   double denominator=0.0;
*/
double xsum;
double ysum;
double ldenom = 0.0;
double rdenom = 0.0;
int dim;
int seq1, pos1;
int seq2, pos2;
gsl_matrix_view view1;
gsl_matrix_view view2;
gsl_matrix * mat1;
gsl_matrix * mat2;
dim = getRdhDim (data);

// Find out which seq,pos pairs these two windows correspond to
getRdhIndexSeqPos (data, win1, &seq1, &pos1);
getRdhIndexSeqPos (data, win2, &seq2, &pos2);

// Get a reference to a submatrix. That is, 'chop out' the window.
view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);

// Some error checking here would be nice!
// Did we get the matrices we wanted?
mat1 = &view1.matrix;
mat2 = &view2.matrix;

// Loop over each position
for (i = 0; i < mat1->size1; i++)
{
xsum = 0.0;
ysum = 0.0;

// Loop over each dimension at each position
for (j = 0; j < dim; j++)
{
xsum += gsl_matrix_get (mat1, i, j);
ysum += gsl_matrix_get (mat2, i, j);
}
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 }

B.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 }

B.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 Spec-
Connect.

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;
double ldenom = 0.0;
double rdenom = 0.0;
int dim;
int seq1, pos1;
int seq2, pos2;
double weight = 2.0;
gsl_matrix_view view1;
gsl_matrix_view view2;
gsl_matrix * mat1;
gsl_matrix * mat2;
double maxElut = -1;
if (extraParams != NULL)
{
    maxElut = extraParams[0];
}
dim = getRdhDim (data);
// Find out which seq, pos pairs these two windows correspond to
getRdhIndexSeqPos (data, win1, &seq1, &pos1);
getRdhIndexSeqPos (data, win2, &seq2, &pos2);
// Get a reference to a submatrix. That is, 'chop out' the window.
view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);
// Some error checking here would be nice!
// Did we get the matrices we wanted?
// This just makes it easier to handle the views
mat1 = &view1.matrix;
mat2 = &view2.matrix;
cum = 1.0;
// Loop over each position
for (i = 0; i < mat1->size1; i++)
{
xsum = 0.0;
ysum = 0.0;
// First take the first dimension for elution time
if (maxElut >= 0)
{
    if (fabs (gsl_matrix_get (mat1, i, 0) - gsl_matrix_get (mat2, i, 0)) >
        maxElut)
    {
        cum = 0;
        break;
    }
}
// printf("\n");
// Loop over each subsequent dimension at each position
for (j = 1; j < dim; j++)
{
    // printf("mat1val=\%lf,mat2val=\%lf\n",gsl_matrix_get(mat1,i,j)),
    // gsl_matrix_get(mat2,i,j));
    numerator += pow (j, weight) * sqrt (gsl_matrix_get (mat1, i, j)
        *gsl_matrix_get (mat2, i, j));
    ldenom += pow (j, weight) * gsl_matrix_get (mat1, i, j);
    rdenom += pow (j, weight) * gsl_matrix_get (mat2, i, j);
    // printf("numer=\%lf,ldenom=\%lf,rdenom=\%lf\n",numerator,
B.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().
else if ((compFunc == 1) || (compFunc == 2))
{
    if (score >= g)
    {
        bitGraphSetTrueSym (bg, i, j);
    }
}
else
{
    fprintf (stderr, "Comparison function undefined in ",
"realComparison function,\n located in ","realCompare.c. Exiting.\n\"");
    fflush (stderr);
    exit (0);
}
}
return bg;

### B.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().

```c
int trans = 1;
int rot = 1;
int dim;
double result = 0;
int seq1, pos1;
int seq2, pos2;
gsl_matrix_view view1;
gsl_matrix_view view2;
gsl_matrix * mat1;
gsl_matrix * mat2;
gsl_matrix * mat1copy;
gsl_matrix * mat2copy;

// The "rint" function is in math.h and rounds a number to the
// nearest integer. It raises an "inexact exception" if the
// number initially wasn’t an integer.
```
if (extraParams != NULL) {
    trans = rint (extraParams[0]);
    rot = rint (extraParams[1]);
}

dim = getRdhDim (data);

// Find out which seq,pos pairs these two windows correspond to
getRdhIndexSeqPos (data, win1, &seq1, &pos1);
getRdhIndexSeqPos (data, win2, &seq2, &pos2);

// Get a reference to a submatrix. That is, 'chop out' the window.
view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);

// This just makes it easier to handle the views
mat1 = &view1.matrix;
mat2 = &view2.matrix;

// Create copies of the windows, because our comparison will require altering the matrices
mat1copy = gsl_matrix_alloc (mat1->size1, mat1->size2);
mat2copy = gsl_matrix_alloc (mat2->size1, mat2->size2);
gsl_matrix_memcpy (mat1copy, mat1);
gsl_matrix_memcpy (mat2copy, mat2);

/*
 printf("matrix1:\n"); gsl_matrix_pretty_fprintf(stdout, mat1copy, "%f ");
 printf("\nmatrix2:\n"); gsl_matrix_pretty_fprintf(stdout, mat2copy, "%f ");
 */

// Are we going to do a translation?
if (trans == 1) {
    moveToCentroid (mat1copy);
    moveToCentroid (mat2copy);
}

// Are we going to do a rotation?
if (rot == 1) {
    // Rotate mat2copy to have a minimal rmsd with mat1copy
    rotateMats (mat1copy, mat2copy);
}

// Compute the rmsd between mat2copy and mat2copy
result = gsl_matrix_rmsd (mat1copy, mat2copy);
gsl_matrix_free (mat1copy);
gsl_matrix_free (mat2copy);
return result;
B.18 realCompare.h File Reference

```c
#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 *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

B.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()`.

B.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()`.
B.18. REALCOMPARE.H FILE REFERENCE

**B.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().

**B.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**

**B.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().
### B.19 realIo.c File Reference

```c
#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)`
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
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 *numSeq_p and *dim_p! Also, you must call this routine before calling parseRealData. Otherwise, parseRealData is guaranteed 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;
     }
     else if (numSeq > 1)
     {
       // 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;
     }
     else
     {
       if (fieldCount != width)
       {
         status = 0;
         break;
       }
     }
     thisDim++;
   }
 } // end loop

// End of file
B.19. REALIO.C FILE REFERENCE

B.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 separators.

Definition at line 90 of file realio.c.

References wordToDouble().

Referenced by checkRealDataFormat(), countTotalFields(), and pushOnRdhSeq().
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().

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\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       if (cliqueAdjMat[i] < min)
1146         {
1147           indmin = i;
1148           min = cliqueAdjMat[i];
1149           flagmin = 0;
1150         }
1151       else if (cliqueAdjMat[i] == min)
1152             flagmin = 1;
1153     }
1154
1155   // If we had a duplicate on the minimum, we locate all duplicates.
1156   if (flagmin == 1)```
counter = 0;
for (i = 0; i < curCliq->set->size; i++) {
    if (cliqueAdjMat[i] == min) {
        counter++;
        candidates[counter] = i;
    }
}

// Store the number of candidates at the array’s beginning
candidates[0] = counter;
free (cliqueAdjMat);
return (-1);
}
else {
    free (cliqueAdjMat);
    return (indmin);
}
}

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().

int i;
if (data != NULL) {
    if (data->indexToPos != NULL) {
        free (data->indexToPos);
        data->indexToPos = NULL;
    }
    if (data->indexToSeq != NULL) {
        free (data->indexToSeq);
        data->indexToSeq = NULL;
    }
    if (data->offsetToIndex != NULL) {
        for (i = 0; i < data->size; i++) {
            free (data->offsetToIndex[i]);
            data->offsetToIndex[i] = NULL;
        }
        free (data->offsetToIndex);
        data->offsetToIndex = NULL;
    }
    for (i = 0; i < data->size; i++) {
        if (data->seq[i] != NULL)
B.19. REALIO.C FILE REFERENCE

490  
491     gsl_matrix_free (data->seq[i]);
492     data->seq[i] = NULL;
493  }
494  if (data->label[i] != NULL)
495     free (data->label[i]);
496     data->label[i] = NULL;
497  }
498  }
499  }
500  if (data->seq != NULL)
501     free (data->seq);
502     data->seq = NULL;
503  }
504  }
505  if (data->label != NULL)
506     free (data->label);
507     data->label = NULL;
508  }
509  }
510     free (data);
511     data = NULL;
512  }
513 return data;
514 }

B.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 }

B.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().

```c
if (data == NULL || data->indexToSeq == NULL || data->indexToPos == NULL
    || index > data->indexSize)
{
    fprintf (stderr, "Passed bad data to getRdhIndexSeqPos -- error!\n");
    fflush (stderr);
    exit (0);
}
```

```c
*seq = data->indexToSeq[index];
*pos = data->indexToPos[index];
return 0;
```

---

**B.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().

```c
if (data == NULL || data->label == NULL || data->label[seqNo] == NULL)
{
    fprintf (stderr, "Passed bad data to getRdhLabel -- error!\n");
    fflush (stderr);
    exit (0);
}
return data->label[seqNo];
```

---

**B.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().
B.19. REALIO.C FILE REFERENCE

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 }

B.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 }

B.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   // Allocate space for our structure  
282   data = (rdh_t *) malloc (sizeof (rdh_t));  
283   if (data == NULL)  
284   {  

B.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().
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\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] = (int *) malloc (seqLen - wordSize + 1) * sizeof (int));  
388 if (data->offsetToIndex[i] == NULL)  
389 {  
390 fprintf (stderr, "\nMemory Error\n\n", strerror (errno));  
391 fflush (stderr);  
392 exit (0);  
393 }  
394 }  
395  
396 // One index for each word plus seqGap between each sequence  
397 // and a gap at the end  
398 data->indexSize = numWindows + numSeq * seqGap;  
399  
400 // Allocate indexToSeq  
401 // NOTE that it should be size of int, not int *... I think we got  
402 // fortunate in the previous revision because they are the same  
403 // size  
404 data->indexToSeq = (int *) malloc (data->indexSize * sizeof (int));  
405 if (data->indexToSeq == NULL)
APPENDIX B. GEMODA FILE DOCUMENTATION

```c
fprintf (stderr, "\n\nMemory Error\n%s\n", strerror (errno));
fflush (stderr);
exit (0);
}

// Allocate indexToPos
// See above for int vs. int* argument.
data->indexToPos = (int *) malloc (data->indexSize * sizeof (int));
if (data->indexToPos == NULL)
{
    fprintf (stderr, "\n\nMemory Error\n%s\n", strerror (errno));
    fflush (stderr);
    exit (0);
}

// Fill in the values
k = 0;
for (i = 0; i < numSeq; i++)
{
    // How many windows are in this sequence?
    thisNumWindows = getRdhSeqLength (data, i) - wordSize + 1;
    // For each window, make an entry in the indexToSeq
    // and indexToPos and offsetToIndex
    for (j = 0; j < thisNumWindows; j++)
    {
        data->indexToSeq[k] = i;
        data->indexToPos[k] = j;
        data->offsetToIndex[i][j] = k;
        k++;
    }
    // Add gaps between sequences in the index.
    // Usually seqGap is just 1;
    for (j = 0; j < seqGap; j++)
    {
        // -1 means no sequence and no position
        data->indexToSeq[k] = -1;
        data->indexToPos[k] = -1;
        k++;
    }
}
return 0;
```

B.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().

```c
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     // We use less-than-or-equal here because we're starting at 1,
1211     // so we want 1 to end. The length of candidates is one more than
1212     // the maxSup, so we know we can reach candidates[maxSup] without
1213     // a segfault.
1214     for (i = 2; i <= numCandidates; i++)
1215     {
1216         getRdhIndexSeqPos (data, curCliq->set->members[candidates[i]], &curSeq,
1217                         &curPos);
1218         if (curPos < min)
1219             {
1220                 indmin = candidates[i];
1221                 min = curPos;
1222             }
1223     }
1224     return (indmin);
1225 }
```

**B.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().

```c
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\ttsup=%d\t", i,
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().

B.19.0.259
if (candidates == NULL)
{
    fprintf (stderr, "Memory Error\n\n", strerror (errno));
    fflush (stderr);
    exit (0);
}
for (i = 0; i <= maxSup; i++)
{
    candidates[i] = 0;
}
comparisonFunc = getCompFunc (compFunc);
curClique = allPats;
i = 0;
while (curClique != NULL)
{
    fprintf (OUTPUT_FILE, "pattern %d: len=%d sup=%d
", i, curClique->length + L, curClique->set->size);
    centroid = findCliqueCentroid (data, curClique, L, comparisonFunc, extraParams, candidates);
    if (centroid < 0)
    {
        centroid = makeAlternateCentroid (data, curClique, candidates);
    }
    else
    {
        fprintf (OUTPUT_FILE, "WARNING: No single node in"
        " cluster has non-zero similarity to all other\n nodes"
        " in cluster; centroid set to first node.\n");
        centroid = 0;
    }
    for (j = 0; j < curClique->set->size; j++)
    {
        pos1 = curClique->set->members[j];
        getRdhIndexSeqPos (data, pos1, &curSeq, &curPos);
        fprintf (OUTPUT_FILE, " %d\t%d\t", curSeq, curPos);
        mfToCentroid = comparisonFunc (data, curClique->set->members[j],
        curClique->set->members[centroid], L, extraParams);
        fprintf (OUTPUT_FILE, "%lf\t", mfToCentroid);
    }
    fprintf (OUTPUT_FILE, 
    for (k = curPos; k < curPos + curClique->length + L; k++)
        { fprintf (OUTPUT_FILE, "%c", mySequences[curSeq].seq[k]); }
    /
    fprintf (OUTPUT_FILE, "\n");  
    fprintf (OUTPUT_FILE, "\n\n");
    curClique = curClique->next;
    i++;
    for (k = 0; k <= maxSup; k++)
    { 
        candidates[k] = 0;
    }
}
free (candidates);
return 0;

B.19. REALIO.C FILE REFERENCE

This function is used to parse a single line of a fastA formatted input buffer containing real
data. The function takes
parameters: a pointer to an array of pointers to characters, which stores the sequences that we will read from \textit{buf}; an integer, which is the line in the buffer on which we should start \textit{nl}; a single character, which is used to delimit the input data \textit{sep}; an integer which is the number of the sequence that we are currently reading in \textit{numSeq}; an integer that is the dimensionality of the input data \textit{dim};

Definition at line 933 of file realIo.c.

References \texttt{countTotalFields()}, \texttt{initRdh()}, and \texttt{pushOnRdhSeq()}. Referenced by \texttt{readRealData()}. 

```c
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     // Hit a new sequence
955     if (buf[i][0] == '>') {
956       
957         seqNo++; // Note that seqNo started at -1!
958         pushOnRdhSeq (data, buf, i, dim, sep);
959         i += dim + 1;
960       }
961     else
962       {
963         i++;
964       }
965     }
966   }
967
968   /*
969    printRdhSeq(data, 0, stdout);
970    */
971   return data;
972 }
```

\textbf{B.19.0.26} \texttt{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 \texttt{data}; an integer which is the sequence to be printed out \texttt{seqNo}; and a pointer to a file handle which is where the output will be printed \texttt{FH}.

Definition at line 710 of file realIo.c.
References getRdhDim(), getRdhLabel(), getRdhSeqLength(), and getRdhValue().

```c
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 \n", getRdhValue (data, seqNo, i, j));
723                 }
724                 fprintf (FH, "\n");
725         }
726     return 0;
727 }
```

**B.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().

```c
864 {
865     int i, j, k;
866     int numFields;
867     // NOTE THAT THESE ARE STATIC VARIABLES!!!!!
868     // That is, they retain their last value on
869     // each call to this function!
870     static int seqNo = 0;
871     static int indexNo=0;
872     /*
873     static int indexNo=0;
874     */
875     i = startLine;
876     // Assume that the sequence has at least
877     // one row (should have called checkRealDataFormat!
878     numFields = countFields (buf[i + 1], sep);
879     numFields = countFields (buf[i + 1], sep);
880     // Initialize the gsl_matrix object for this
881     // sequence in 'data'
882     //
```
initRdhGslMat (data, seqNo, numFields, dim);

// Set the sequence label
setRdhLabel (data, seqNo, buf[i]);

// Read in 'dim' rows
for (j = i + 1, k = 0; j < i + 1 + numFields; j++, k++)
{

  /*
  printf("%d\n", countFields(buf[j], sep));
  */

  // Set the k-th dimension of this sequence
  // STILL NOTE THE TRANSPOSE!
  setRdhColFromString (data, seqNo, k, buf[j], sep);
}

/*
for ( l=0 ; l<numFields ; l++ ){ setRdhIndex(data, seqNo, l, indexNo); indexNo++; }
*/
seqNo++;

// Augment indexNo once more to have a -1 between each sequence!
/*
indexNo++;
*/
return 0;

B.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().
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\n");  
1009        fprintf (stderr,  
1010            "Each sequence needs to be the same dimensionality! QUITTING!\n\n");  
1011        fprintf (stderr,  
1012            "numSeq = \%d, dimensions = \%d, status = \%d\n", numSeq,  
1013            dimensions, status);  
1014        exit (EXIT_FAILURE);  
1015    }  
1016  
1017  // From here on, we assume that the sequence file is well-formatted  
1018  // to make the code more simple.  
1019  data = parseRealData (buf, nl, sep, numSeq, dimensions);  
1020  
1021  // Free up our buffer  
1022  for (i = 0; i < nl; i++)  
1023    {  
1024        if (buf[i] != NULL)  
1025            free (buf[i]);  
1026    }  
1027  if (buf != NULL)  
1028    {  
1029        free (buf);  
1030    }  
1031  return data;  
1032}  

B.19. REALIO.C FILE REFERENCE

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
if (s == NULL || data->seq == NULL || data->seq[seqNo] == NULL) {
    fprintf(stderr, "Passed bad data to setRdhColFromString -- error!");
    fflush(stderr);
    exit(0);
}

// Loop over the length of the string
for (i = 0; i < strlen(s); i++) {
    // The previous state was space
    if (status == 0) {
        // We hit a word
        if (s[i] != sep) {
            begin = i;
            status = 1;
        } else { // We hit more space
            continue;
        }
    } else { // The previous state was word
        if (s[i] != sep) {
            continue;
        } else { // We hit a space
            end = i - 1;
            status = 0;
            val = wordToDouble(s, begin, end);
            // Go to the gsl_matrix object data->seq[seqNo]
            // and set the (fieldCount, colNo) = val;
            setRdhValue(data, seqNo, fieldCount, colNo, val);
            fieldCount++;
        }
    }
    // At the end, if we were in a word, we have
    if (status == 1) { // We're in a word
        val = wordToDouble(s, begin, strlen(s));
        // Added in, MPS 5/3/05 ---
        // And don't forget to set the RdhValue!
        setRdhValue(data, seqNo, fieldCount, colNo, val);
        fieldCount++;
    }
    return fieldCount;
}
B.19. REALIO.C FILE REFERENCE

B.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.

```c
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!\n");
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}
```

B.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().

```c
544 {
545  if (data->seq == NULL || data->label == NULL)
546   {
547      fprintf (stderr, "Passed bad data to setRdhLabel -- error!\n");
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);
```
B.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 data should be set.

Definition at line 575 of file realIo.c.
References getRdhDim(), getRdhSeqLength(), and rdh_t::seq.
Referenced by setRdhColFromString().

B.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-formatted data files.
double

Definition at line 30 of file realIo.c.
Referenced by countFields(), and setRdhColFromString().
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 }
**B.20 realIo.h File Reference**

```c
#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

B.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().

B.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().

B.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().
APPENDIX B. GEMODA FILE DOCUMENTATION

B.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().

B.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().

B.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().

B.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().
B.2.1 spat.h File Reference

This graph shows which files directly or indirectly include this file:

Data Structures

- struct sOffset_t
- struct sPat_t
B.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 hashpjaw (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**

B.22.0.276  `#define SHASH_MAX_KEY_SIZE 1000`

Definition at line 192 of file `words.c`.

Referenced by `printSHash()`, and `searchSHash()`.

**Function Documentation**

B.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()`.

```c
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 // Count the total number of characters. This
387 // is the upper limit on how many words we can have
388 for (i = 0; i < numSeq; i++) {
389 ```
totalChars += strlen (seq[i].seq);
}

// Get a prime number for the size of the hash table
hashSize = sieve3 ((long) (2 * totalChars));
wordHash = initSHash (hashSize);

// Chop up each sequence and hash out the words of size L
for (i = 0; i < numSeq; i++)
{
    prev = -1;
    // skip sequences that are too short to have
    // a pattern
    if (strlen (seq[i].seq) < L)
    {
        continue;
    }
    for (j = 0; j < strlen (seq[i].seq) - L + 1; j++)
    {
        // Make a hash table entry for this word
        newEntry.key = &(seq[i].seq[j]);
        newEntry.data = 1;
        newEntry.idx = wc;
        newEntry.L = L;
        // Check to see if it’s already in the hash table
        ep = searchSHash (&newEntry, &wordHash, 0);
        if (ep == NULL)
        {
            // If it’s not, create an entry for it
            ep = searchSHash (&newEntry, &wordHash, 1);
            // Increase the size of our word array
            words = (sPat_t *) realloc (words, (wc + 1) * sizeof (sPat_t));
            if (words == NULL)
            {
                fprintf (stderr, "Error!
");  
                fflush (stderr);
                exit (0);
            }
            // Add the new word
            words[wc].string = &(seq[i].seq[j]);
            words[wc].length = L;
            words[wc].support = 1;
            words[wc].offset = (sOffset_t *) malloc (1 * sizeof (sOffset_t));
            if (words[wc].offset == NULL)
            {
                printf (stderr, "%nMemory Error
", strerror (errno));
                fflush (stderr);
                exit (0);
            }
            words[wc].offset[0].seq = i;
            words[wc].offset[0].pos = j;
            words[wc].offset[0].prev = prev;
            words[wc].offset[0].next = -1;
            if (prev != -1)
            {
                words[prev].offset[words[prev].support - 1].next = wc;
            }
            prev = wc;
            wc++;
        }
    }
else
APPENDIX B. GEMODA FILE DOCUMENTATION

459   
460   // If it is, increase the count for this word
461   ep->data++;
462   
463   // add a new offset to the word array
464   l = words[ep->idx].support;
465   words[ep->idx].offset =
466   (sOffset_t *) realloc (words[ep->idx].offset,
467   (l + 1) * sizeof (sOffset_t));
468   words[ep->idx].offset[l].seq = i;
469   words[ep->idx].offset[l].pos = j;
470   words[ep->idx].offset[l].prev = prev;
471   words[ep->idx].offset[l].next = -1;
472   
473   // Update the next/prev
474   if (prev != -1)
475   {
476     words[prev].offset[words[prev].support - 1].next = ep->idx;
477   }
478   prev = ep->idx;
479   
480   // Have to put this down here for cases when we create
481   // a word and it is immediately followed by itself!!
482   words[ep->idx].support += 1;
483   }
484   
485   destroySHash (&wordHash);
486   *numWords = wc;
487   return words;
488   }
489   

B.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 }
B.22. WORDS.C FILE REFERENCE

B.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.

```c
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       free (words);
363       words = NULL;
364       return 0;
365     }
```n

B.22.0.280  unsigned long hash1 (unsigned char * str)

A hashing function that returns an integer, given a pointer to a null character terminated string.

Definition at line 73 of file words.c.

Referenced by searchSHash().

```c
74 {
75   unsigned long hash = 5381;
76   int c;
77   while ((c = *str++))
78     hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
79   return hash;
80 }
```n

B.22.0.281  int hashpjw (char * s)

A hashing function that returns an integer, given a pointer to a null character terminated string.

Definition at line 89 of file words.c.

```c
90 {
91   char *p;
92   unsigned int h, g;
93   h = 0;
94   for (p = s; *p != '\0'; p++)
95     {
96         h = (h << 4) + *p;
```n
B.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().

B.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.

```c
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            strncpy (string, thisHash->hash[i][j].key, thisHash->hash[i][j].L);
308            string[thisHash->hash[i][j].L] = '\0';
309            fprintf (FH, "%s %d\n", string, thisHash->hash[i][j].data);
310        }
311    }
312    return 0;
313 }
314
```

B.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.

```c
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)\n", a[i].offset[j].seq, a[i].offset[j].pos, a[i].offset[j].prev, a[i].offset[j].next);
338        }
339    }
340    free (s);
341    return 0;
342 }
343```
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().

```c
{ 
  char string[SHASH_MAX_KEY_SIZE];
  unsigned long (*hashFunction) () = &hash1;
  int i, thisIndex;
  int status = 0;

  // A string to store the key
  strncpy (string, newEntry->key, newEntry->L);
  string[newEntry->L] = '\0';

  // The index that this key hashes to
  thisIndex = hashFunction ((unsigned char *) string) % thisHash->totalSize;

  // For each member that has this index, check to see
  // if the key is the same
  if (status == 0 && create != 0)
  { 
    // Allocate space for the new entry at this index
    if (thisHash->iHashSize[thisIndex] == 0)
    { 
      thisHash->hash[thisIndex] =
      (sHashEntry_t *) malloc (sizeof (sHashEntry_t));
    } else
    { 
      thisHash->hash[thisIndex] =
      (sHashEntry_t *) realloc (thisHash->hash[thisIndex],
      (thisHash->iHashSize[thisIndex] +
      1) * sizeof (sHashEntry_t));
    }
    if (thisHash->hash[thisIndex] == NULL)
    {
      printf("\t\%s already in hash table!\n"),
      return &
    
  // If we didn't find the key and we're told to create it,
  // then allocate new memory for the hashEntry and put it in
  if (status == 0 && create == 0)
  { 
    if (thisHash->iHashSize[thisIndex] == 0)
    { 
      thisHash->hash[thisIndex] =
      (sHashEntry_t *) malloc (sizeof (sHashEntry_t));
    } else
    { 
      thisHash->hash[thisIndex] =
      (sHashEntry_t *) realloc (thisHash->hash[thisIndex],
      (thisHash->iHashSize[thisIndex] +
      1) * sizeof (sHashEntry_t));
    }
    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     // Return a pointer to this entry
262     return &(thisHash->hash[thisIndex][i]);
263 
264 }
265 return NULL;
266 }

B.22.0.286  int sieve3 (long n)

Prime number generator: returns first prime number equal or less than

Parameters:

  n.

Definition at line /uniF645/uniF64A 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         } while (j <= n);
53         p = p + 1;
54     } while (p * p < 2 * n);
55     for (i = n; i > 2; i--)
56     {
57         if (a[i])

```c
60 |
61 free (a);
62 return i;
63 |
64 |
65 free (a);
66 return 0;
67 }
```
Appendix C

Gemoda data structure documentation

C.1 Introduction

This appendix describes in detail the data structures used in the Gemoda software, which is described in the appendix on page 225. Although C is not an object–oriented programming language, we have tried where possible to use a similar philosophy in our programming.

C.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**

C.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().

C.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
C.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

C.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().

C.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().

C.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().
C.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
C.4  cnode Struct Reference

#include <convll.h>

Collaboration diagram for cnode:

```
cSet_t
   set
```

<table>
<thead>
<tr>
<th>Data Fields</th>
</tr>
</thead>
<tbody>
<tr>
<td>• cSet_t * set</td>
</tr>
<tr>
<td>• int id</td>
</tr>
<tr>
<td>• int length</td>
</tr>
<tr>
<td>• cnode * next</td>
</tr>
<tr>
<td>• double stat</td>
</tr>
</tbody>
</table>

**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**

C.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().

C.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().

C.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().

C.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().

C.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
C.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

C.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().

C.5.0.299 int cSet_t::size

Number of members in this set.

Definition at line 24 of file convll.h.

Referenced by bitSetToCSet(), calcStatClique(), 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
C.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

C.6.0.300  char* fSeq_t::label

Definition at line 14 of file fastaSeqIO.h.
Referenced by FreeFSeqs(), initAofFSeqs(), and ReadFSeqs().

C.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
C.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

C.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().

C.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
C.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

C.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().

C.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().
C.8.0.306 \textbf{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().

C.8.0.307 \textbf{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().

C.8.0.308 \textbf{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().

C.8.0.309 \textbf{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().

C.8.0.310 \textbf{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
C.9 sHash_t Struct Reference

Collaboration diagram for sHash_t:

![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

C.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().

C.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().
C.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().

C.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
C.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

C.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().

C.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().

C.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().

C.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 countWords(), printSHash(), and searchSHash().

The documentation for this struct was generated from the following file:

- *words.c*
### C.11 sOffset_t Struct Reference

```c
#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

##### C.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().

##### C.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().

##### C.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().
C.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


## C.12  sPat_t Struct Reference

```c
#include <spat.h>
```

Collaboration diagram for `sPat_t`:

```
  sOffset_t
    offset
```

### 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

**C.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()`.

**C.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()`.
C.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().

C.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
C.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

C.13.0.327 int sSize_t::size

Definition at line 168 of file fastaSeqIO.c.
Referenced by ReadFSeqs().

C.13.0.328 int sSize_t::start

Definition at line 166 of file fastaSeqIO.c.
Referenced by ReadFSeqs().

C.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


1008–12, Aug 2005. 109

[118] K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch. The PROSITE database, its status


Letter. 14, 140, 144


[123] P. Horton. Tsukuba BB: a branch and bound algorithm for local multiple alignment of

[124] J. D. Hughes, P. W. Estep, S. Tavazoie, and G. M. Church. Computational identifi-
cation of cis-regulatory elements associated with groups of functionally related genes in

[125] C. G. Hunter and S. Subramaniam. Protein fragment clustering and canonical local

J. D. Aitchison, L. Hood, A. F. Siegel, and H. Bolouri. A data integration methodology

[127] T. Ideker and D. Lauffenburger. Building with a scaffold: emerging strategies for high-


[246] E. Tiozzo, G. Rocco, A. Tossi, and D. Romeo. Wide-spectrum antibiotic activity of


[269] I. H. Witten and E. Frank. *Data mining: practical machine learning tools and techniques*


covery (DMKD’98), Seattle, Washington, 1998. URL citeseer.ist.psu.edu/zaki98theoretical.html. 116


