

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

© Kyle L. Jensen, 2006.

Author
Department of Chemical Engineering
April 27, 2006

Certified by
Gregory N. Stephanopoulos
Bayer Professor of Chemical Engineering
Thesis Supervisor

Accepted by
William M. Deen
Chairman, Department Committee on Graduate Students

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

2.3.3	Antimicrobial sequence and grammar databases	84
2.3.4	Annotator design and validation	84
2.4	Preliminary strategy for the design of novel AmPs	88
2.4.1	Sequence design	88
2.4.2	Peptide synthesis and validation	92
2.4.3	Later experimentation	99
2.5	Focused design of AmPs	99
2.5.1	Derivation of highly conserved AmP grammars	99
2.5.2	Design of synthetic AmP sequences	101
2.5.3	Assay for antimicrobial activity	105
2.5.4	Results and conclusions	106
3	A generic motif discovery algorithm	111
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

3.5.3	Motif discovery in nucleotide sequences and the (l,d) -motif problem	143
3.6	Discussion	154
4	Other exercises in motif discovery	159
4.1	Introduction	159
4.2	Biogrep: a tool for matching regular expressions	160
4.2.1	Introduction	160
4.2.2	Implementation and results	160
4.3	The evolution of updated BLOSUM matrices and the Blocks database	162
4.3.1	Introduction	162
4.3.2	Motivation	163
4.3.3	Methods	164
4.3.4	Results	168
4.3.5	Discussion	174
4.4	Bioinformatics and handwriting/speech recognition: unconventional applications of similarity search tools	177
4.4.1	Introduction	177
4.4.2	System and Methods	178
4.4.3	Results	184
4.4.4	Conclusions	186
4.5	Machine learning approaches to modeling the physiochemical properties of peptides	188
4.5.1	Introduction	188
4.5.2	Motivation and background	188
4.5.3	Methods	191
4.5.4	Conclusion	195
4.6	Identifying functionally important mutations from phenotypically diverse sequence data	200
4.6.1	Introduction	200
4.6.2	Motivation	200

4.6.3	Materials and Methods	202
4.6.4	Results	204
4.6.5	Discussion	210
A	Abbreviations and reference data	215
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	225
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	421
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

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

2-9	Design space	93
2-10	Antimicrobial peptide action	95
2-11	Gel picture	97
2-12	Antimicrobial peptide hemolysis	98
2-13	Example grammar-based dot plot	103
2-14	Example grammar-based dot plot for computing Q	104
2-15	Activity of rationally designed AmPs against <i>S. aureus</i> and <i>B. anthracis</i>	108
3-1	Alignment representing the LexA cis-regulatory binding site	115
3-2	A sketch showing the flow of the Gemoda algorithm for an example input set of protein sequences.	118
3-3	Pseudo-code for the Gemoda convolution	129
3-4	A natural language example illustrating the steps that Gemoda takes	131
3-5	A second natural language example	134
3-6	Guanosine-3',5'-bis(diphosphate) 3'-pyrophosphohydrolase ((ppGpp)ase) (Penta-phosphate guanosine-3'-pyrophosphohydrolase) sequences	137
3-7	The RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences.	139
3-8	Logo representation of the RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences	140
3-9	Logo representation of the RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences	141
3-10	The similarity graph for the Gemoda 3.1.7.2 enzyme example	142
3-11	Alpha carbon trace projection used by Gemoda	143
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]	144
3-13	Structural motif in Gemoda's 3-D structure viewer	145
3-14	The sequence logo for a) the motif implanted in each sequence for the (<i>l,d</i>)–motif problem and b) the LexA binding site motif generated from the highest-scoring motif returned by Gemoda.	153

4-1	Characteristics of the BLOSUM matrices calculated from successive releases of the Blocks database	165
4-2	The relative performance of updated BLOSUM matrices	171
4-3	A complete set of Bayesian bootstrap replicates, with inset histogram of coverage difference	172
4-4	Plots of the differences in performance of updated RBLOSUM matrices . . .	173
4-5	Coverage of a cleaned RBLOSUM matrix compared to the RBLOSUM64 matrix	175
4-6	Projection of a digit written with a PDA stylus into protein space	179
4-7	A Voice alignment of the spoken–letter “X” recorded from two different speakers	180
4-8	A phylogenetic tree of voice–proteins	181
4-9	Structure of the HIV-I protease, derived from the Protein Data Bank (PDB) [34] entry 7HVP [241]	190
4-10	Schematic of the HIV-I protease active site	191
4-11	Decision tree calculated for modeling 8–mer peptides	198
4-12	Classification results for all amino acid representations and model types . . .	199
4-13	Structure of the PL–TET _{O1} promoter	205
4-14	Schematic of the experimental procedure for promoter mutagenesis	206
4-15	Statistical distribution of mutations and their effects on mutant fluorescence .	209
A-1	Amino acid structures and abbreviations	216
A-2	Nucleotide base structures and abbreviations	217
A-3	Gas and mass spectra for the synth-I peptide	224

List of Tables

1.1	“Omic” fields other than genomic and proteomic	22
1.2	Gene sequences from <i>Arabodopsis thaliana</i>	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 <i>S.</i> <i>aureus</i> and <i>B. anthracis</i>	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

4.7	Machine learning representation ranking	197
4.8	Summary of site-directed mutagenesis loci	213
4.9	Summary of double and triple mutants constructed by site-directed mutagenesis.	214
A.1	Standard codon table	217

Chapter I

Introduction to motif discovery

I.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 I-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 I-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

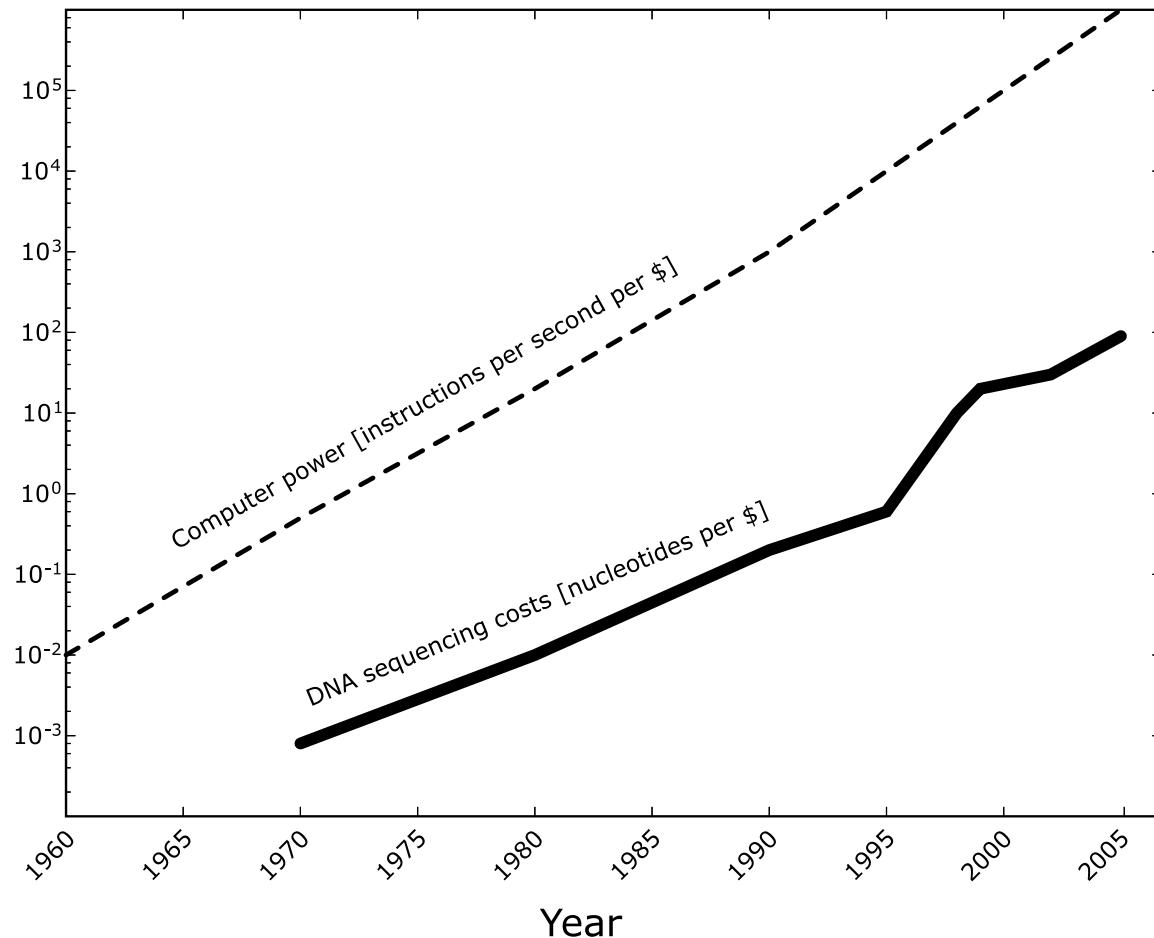


Figure 1-1: Exponential growth of sequencing throughput [227]. The figure tracks the number of nucleotides that can be sequenced for a dollar over time juxtaposed with the advancement of computing power. The hypothesis referred to as “Moore’s Law” — that computational power doubles every 18 months — 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].

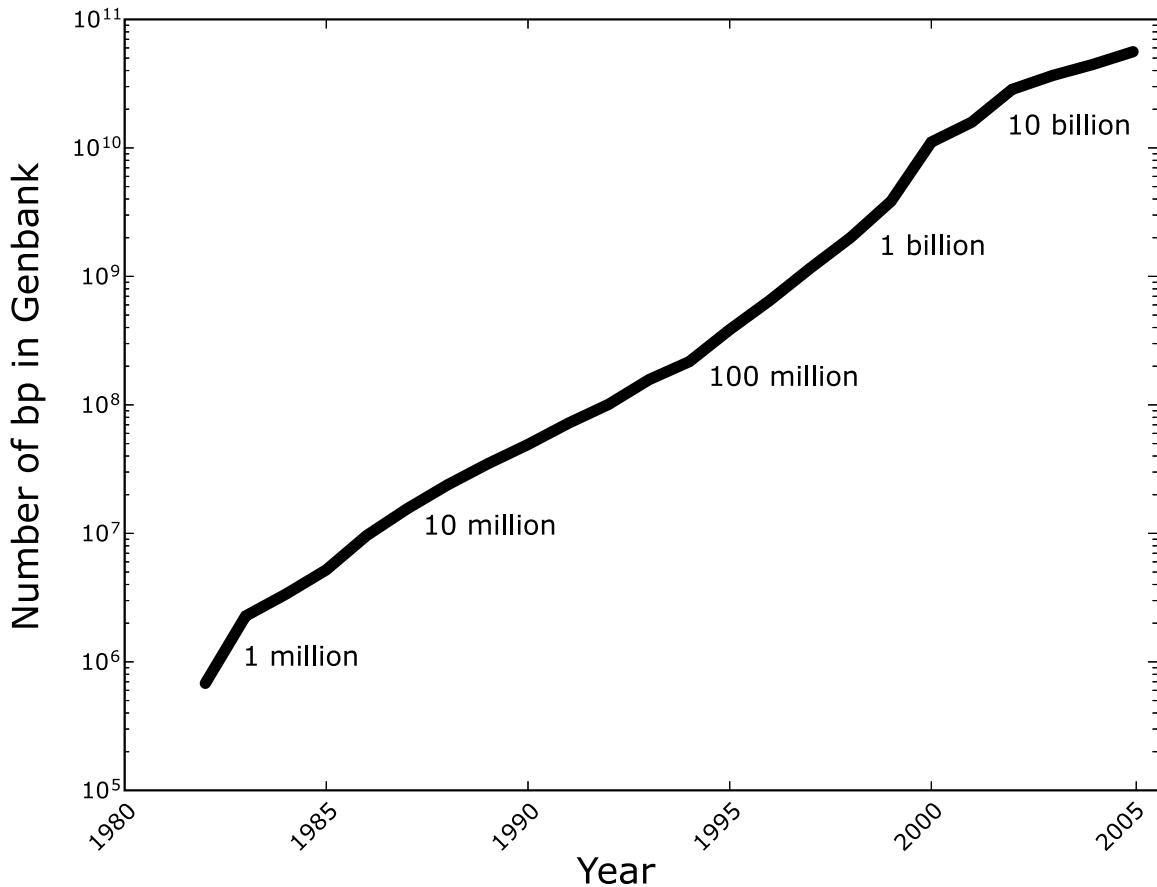


Figure 1-2: Exponential growth of Genbank [33]. Genbank is a comprehensive database of DNA sequences from over 200,000 organisms that were made public by direct submission from researchers. The database is maintained by the U.S. National Center for Biotechnology Information, and is updated daily. The 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].

Anatomics	Biomics	Chromosomics	Cytomics
Enviromics	Epigenomics	Fluxomics	Glycomics
Glycoproteomics	Immunogenomics	Immunomics	Immunoproteomics
Integromics	Interactomics	Ionomics	Lipidomics
Metabolomics	Metabonomics	Metagenomics	Metallomics
Metalloproteomics	Methylomics	Mitogenomics	Neuromics
Neuropeptidomics	Oncogenomics	Peptidomics	Phenomics
Phospho-proteomics	Phosphoproteomics	Physiomics	Physionomics
Post-genomics	Postgenomics	Pre-genomics	Rnomics
Secretomics	Subproteomics	Surfaceomics	Syndromics
Transcriptomics			

“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

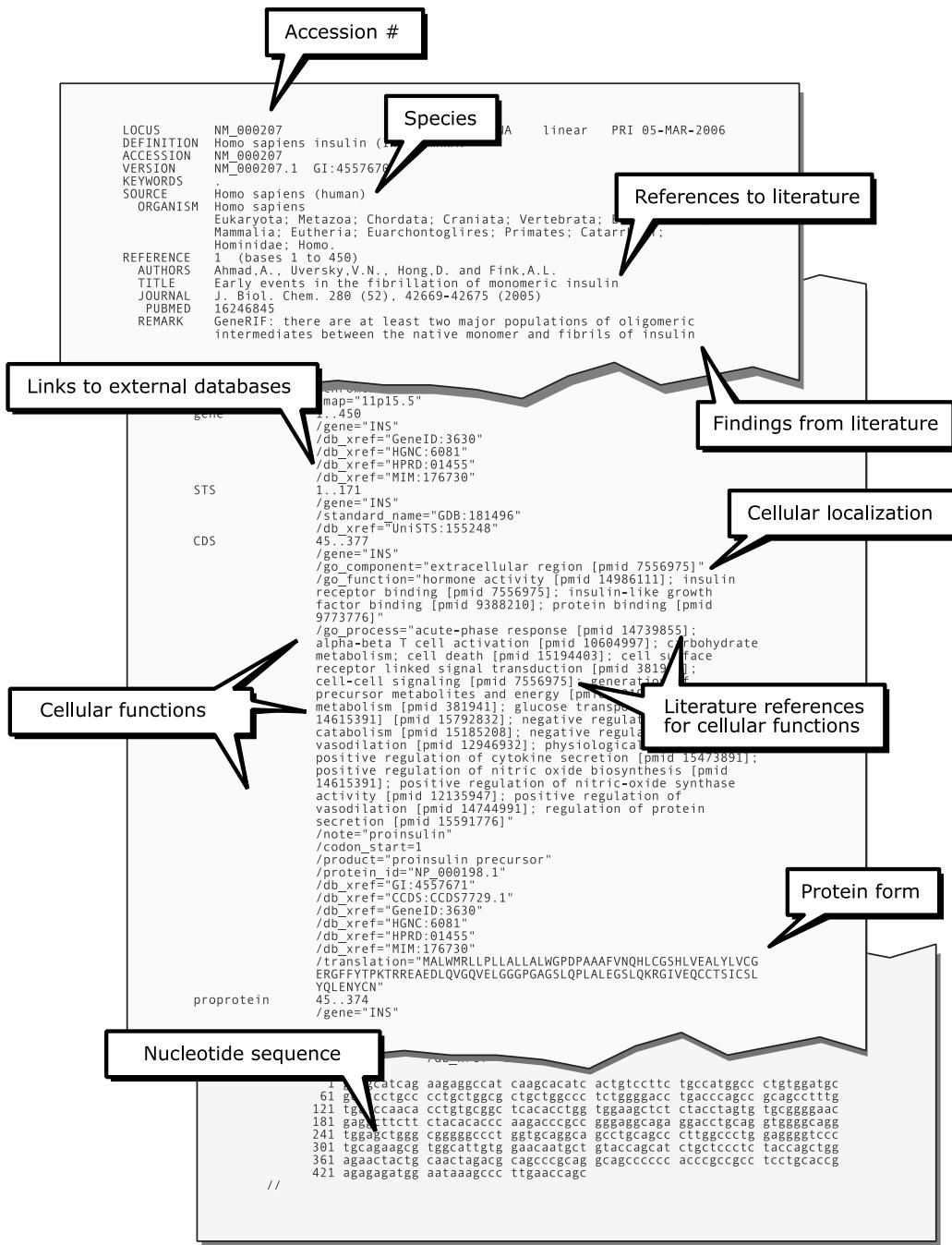


Figure 1-3: A sample Genbank record showing the litany of features annotating the human gene for insulin. As the figure suggests, this “extra” information typically dwarfs the gene sequence in size. The annotations are highly cross-referenced, linking genes to cellular functions, behaviors, localizations 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, an organism levels.

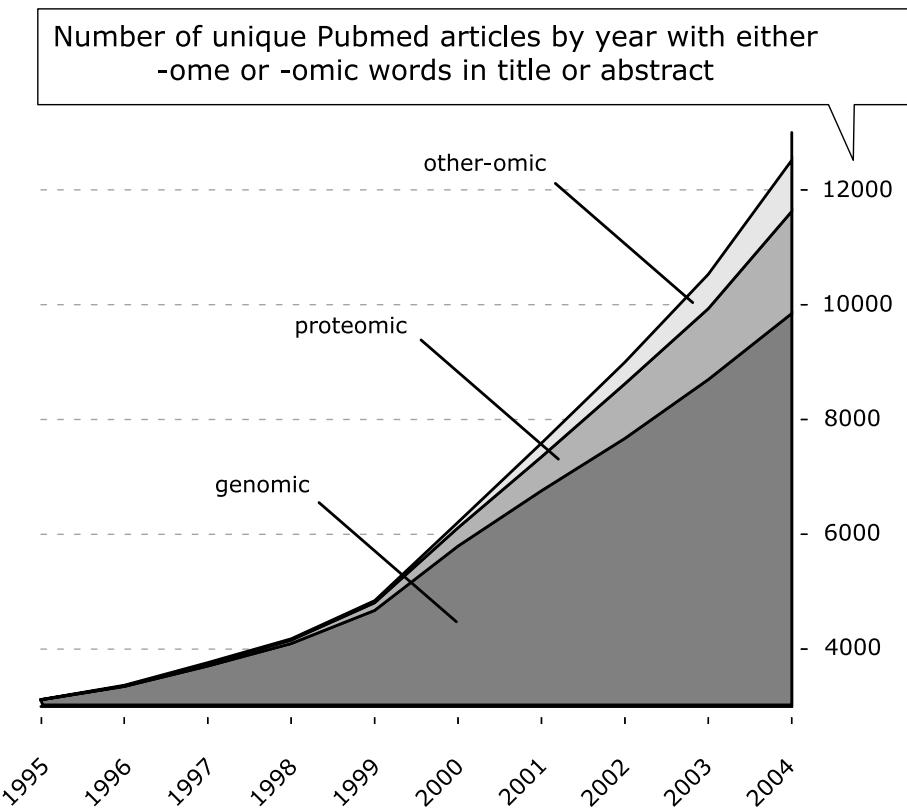


Figure 1-4: Usage of “ome” and “omic” words over time divided into three categories: genome and genomic, proteome and proteomic, and other ome and omic. The latter category comprises those words shown in Table 1.1 on page 22.

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 *Arabidopsis 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 CCACGCGTCCGAAAAA? 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.

```
>gi|8571926 Arabidopsis thaliana lipid transfer protein
CCACGCGTCCGAAAAAAACAGAAAGTAACATGAGATCTCTCTTATTAGCCGTGCGCTGGTTCTTGC
TTTACACTGCGGTGAAGCAGCCGTCTGCAACACGGTGATTGCGGATCTTACCCCTGCTTATCCTAC
GTGACTCAGGGCGGACCGGTCCCAACCCCTCTGCTGCAACCGTCTCACAAACACTCAAGAGTCAGGCTCAA
CTTCTGTGGACCGTCAGGGGTCTGCGTTGCATCAAATCTGCTATTGGAGGACTCACTCTCTCCTAG
AACCATCCAAAATGCTTTGGAATTGCTTCTAAATGTGGTGTGATCTCCCTACAAGTTCAGCCCTTCC
ACTGACTGCGACAGTATCCAGTGAGACAAGCAGAAAATCTAAAGGAAGCTACTACAAGAACTATAATAA
CCTAATAATTAATAATGAGGGCATTGGTTGCTAGTGCTAATTGATCAGTGATGTATTGTCATTTGA
ATGTTCTAATATCAGCAGGCACTTATCTCTGAAAAAAAAAAAAAA
```

```
>gi|8571922 Arabidopsis thaliana lipid transfer protein
CCACGCGTCCGAAAACACAAGCGTAGAAAACAAAACACTCAACTAATTGTGTTATCACCCAAAAGAGAGAG
CAAACACAATGGCTTCGCTTGAGGTTCTCACATGCTTGTGACAGTGTTCATCGTGCATCAGT
GGATGCAGCAATAACATGTGGCACAGTGGCAAGTAGCTTGAGTCCATGTCTAGGCTACCTATCGAAGGGT
GGGGTGGTGCCACCTCCGTGCTGTGCAGGAGTCAAAAGTTGAACGGTATGGCTAAACCACACCCGACC
GCCAACAAAGCATGCAGATGCTTACAGTCCGCTGAAAAGGGTTAATCCAAGTCTAGCCTCTGGCCTTCC
TGGAAAGTGCAGGTGTTAGCATCCCTATCCCATCTCACGAGCACCAACTGCGCCACCATCAAGTGAAGT
GGGAATAACGACATCATTGCTGAAGAGTATGGTTGCTATACGTAAGGACGGCTATCTAAGCT
GATATTACCTTGTCTTGTTGCTGATGGCTTGTAATCTTGCTTGTATGTTGATACTTGT
TCTTAACATGTTAAGATATGATAATATAGTATCGGTACCTTATTAAAAAA
```

```
>gi|8571922 Arabidopsis thaliana lipid transfer protein
CCACGCGTCCGAAAACACAAGCGTAGAAAACAAAACACTCAACTAATTGTGTTATCACCCAAAAGAGAGAG
CAAACACAATGGCTTCGCTTGAGGTTCTCACATGCTTGTGACAGTGTTCATCGTGCATCAGT
GGATGCAGCAATAACATGTGGCACAGTGGCAAGTAGCTTGAGTCCATGTCTAGGCTACCTATCGAAGGGT
GGGGTGGTGCCACCTCCGTGCTGTGCAGGAGTCAAAAGTTGAACGGTATGGCTAAACCACACCCGACC
GCCAACAAAGCATGCAGATGCTTACAGTCCGCTGAAAAGGGTTAATCCAAGTCTAGCCTCTGGCCTTCC
TGGAAAGTGCAGGTGTTAGCATCCCTATCCCATCTCACGAGCACCAACTGCGCCACCATCAAGTGAAGT
GGGAATAACGACATCATTGCTGAAGAGTATGGTTGCTATACGTAAGGACGGCTATCTAAGCT
GATATTACCTTGTCTTGTTGCTGATGGCTTGTAATCTTGCTTGTATGTTGATACTTGT
TCTTAACATGTTAAGATATGATAATATAGTATCGGTACCTTATTAAAAAA
```

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.



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?

$$\blacksquare \sim \blacktriangle \sim \square \quad (1.2)$$

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

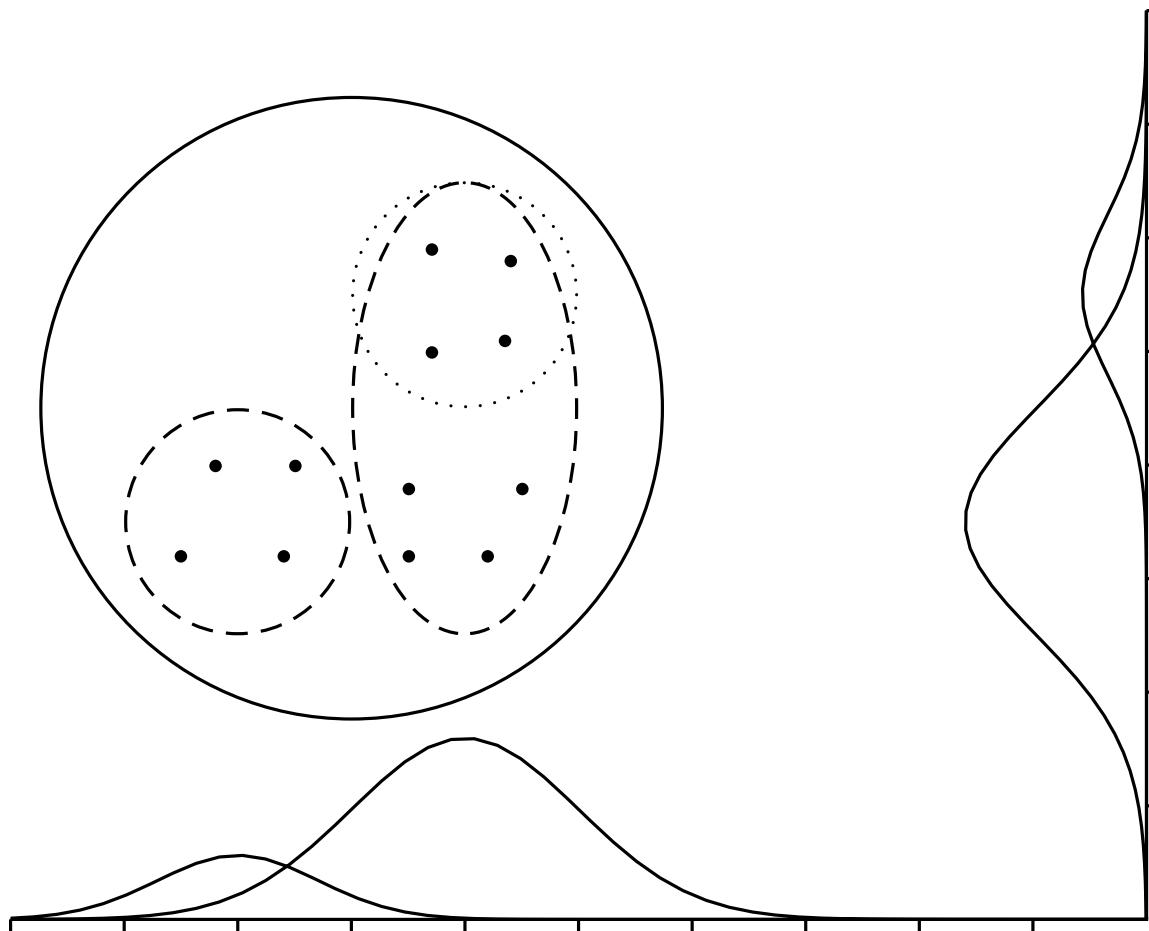


Figure 1-5: Clustering to find patterns in generic data. Given this data set, we would cluster the data together to find patterns (indicated here by the ellipses). But, there are many different patterns we could find. Which ones are correct?

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.

```
d jndf duckdeicf jnfmABRAHAMLINCOLN
idkei oddsnABRAHAMLINCOLNaknkwbad
ioxcABRAHAMLINCOLNabkjw1kdaxlakj
xkasnkjlfABRAHAMLINCOLNkdsjkjsdl (1.3)
```

We can refer to these sequences more formally by calling the set of sequences S , where S is defined such that $S = \{s_0, s_1, s_2, \dots, s_n\}$, and where sequence s_i has length W_i . So, in the above example, $n = 3$ and $s_0 = d\ jndf\ duckdeicf\ jnfmABRAHAMLINCOLN$. Furthermore, let 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 primitives are alphanumerical characters selected from some alphabet Σ . In the example above, this alphabet is the set of 52 lowercase and uppercase characters from the Roman alphabet. However, 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 characters {A, T, G, C}, representing the four bases found in DNA: adenine, 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 occurring 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.

$$s_0 = d\ j\ n\ d\ f\ d\ u\ c\ k\ d\ e\ i\ c\ f\ j\ n\ f\ m\ A\ B\ E\ L\ I\ N\ C\ O\ L\ N \quad (1.4)$$

$$s_1 = i\ d\ k\ e\ i\ o\ d\ d\ s\ n\ A\ B\ R\ A\ H\ A\ M\ L\ I\ N\ C\ O\ L\ N\ a\ k\ n\ k\ w\ b\ a\ d$$

$$s_2 = i\ o\ x\ c\ A\ L\ I\ N\ C\ O\ L\ N\ a\ b\ k\ j\ w\ l\ k\ d\ a\ x\ l\ a\ k\ j$$

$$s_3 = x\ k\ a\ s\ n\ k\ j\ l\ f\ A\ B\ R\ A\ H\ A\ M\ L\ I\ N\ C\ O\ L\ N\ k\ d\ s\ j\ k\ j\ s\ d\ l$$

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. Σ is a finite set of terminal symbols, disjoint from N ;
3. P is a finite subset of

$$(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*, \quad (1.5)$$

each element in P is called a “production” and is written in the form $\alpha \rightarrow \beta^1$; 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:

$$G = (\{\alpha, S\}, \{0, 1\}, P, S), \quad (1.6)$$

where the set of productions, P , is given by

¹The 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.

$$P = \begin{cases} S \rightarrow 0\alpha1 \\ 0\alpha \rightarrow 00\alpha1 \\ \alpha \rightarrow e. \end{cases} \quad (1.7)$$

Each line in the set of productions is essentially a replacement rule. For example, the operation $S \rightarrow 0\alpha1$ 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 0α with the sequence $00\alpha1$.” Finally, the last production should be read as “replace the character α 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.

$$S \Rightarrow 0\alpha1$$

Using the second production, produces a four character sequence.

$$S \Rightarrow 0\alpha1 \Rightarrow 00\alpha1$$

Finally, using the third production terminates the sequence.

$$S \Rightarrow 0\alpha1 \Rightarrow 00\alpha1 \Rightarrow 001 \quad (1.8)$$

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 α appears in two different productions, either production can be used and neither production is preferred *a priori*. For example the following sequence is an equally valid derivation of the same grammar.

$$S \Rightarrow 0\alpha 1 \Rightarrow 00\alpha 1 \Rightarrow 000\alpha 1 \Rightarrow 0000\alpha 1 \Rightarrow 00000\alpha 1 \Rightarrow 000001 \quad (1.9)$$

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

$$G = (\{\alpha, S\}, \{A, B, C, E, H, I, L, M, N, R\}, P, S), \quad (1.10)$$

where P is given by the set of productions

$$P = \left\{ \begin{array}{l} S \rightarrow A\alpha \\ \alpha \rightarrow \beta \\ \alpha \rightarrow BE\beta \\ \alpha \rightarrow BRAHAM\beta \\ \beta \rightarrow LINCOLN\gamma \\ \gamma \rightarrow e. \end{array} \right. \quad (1.11)$$

Again, in a manner similar to Equation 1.7 on page 33, the grammar shown in Equation 1.11 has a non-terminal character, α , 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

$$P = \left\{ \begin{array}{l} S \rightarrow A\alpha \\ \alpha \rightarrow \beta | BE\beta | BRAHAM\beta \\ \beta \rightarrow LINCOLN\gamma \\ \gamma \rightarrow e. \end{array} \right. \quad (1.12)$$

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.

$$S \Rightarrow A\alpha \Rightarrow A\beta \Rightarrow ALINCOLN\gamma \Rightarrow ALINCOLN \quad (1.13)$$

$$S \Rightarrow A\alpha \Rightarrow ABE\beta \Rightarrow ABELINCOLN\gamma \Rightarrow ABELINCOLN$$

$$S \Rightarrow A\alpha \Rightarrow ABRAHAM\beta \Rightarrow ABRAHAMLINCOLN\gamma \Rightarrow ABRAHAMLINCOLN$$

This is an ideal case. In general, in constructing any model describing any motif instances,

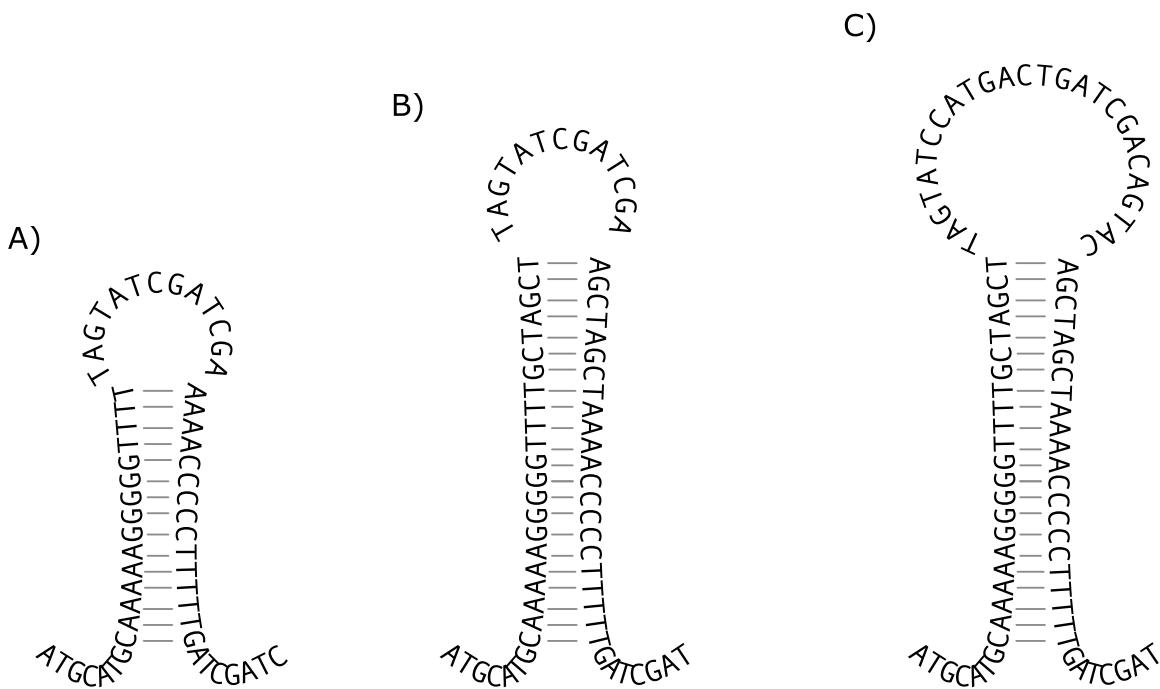


Figure 1-6: Hairpin loops in DNA secondary structures. A hairpin loop is a secondary structure in a sequence containing two regions that are reverse complements of each other. These regions form the “stem” of the hairpin loop. The figure shows three hairpin loops in which the stem size gets progressively larger. Also notice that, the bulbous region — the “loop” — which is not paired with any other region, can be of arbitrary size. The structures play an important part in the regulation of DNA transcription and, for RNA, in the process of translation.

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

grammar:

$$G = (\{\alpha, \beta, S\}, \{A, G, T, C\}, P, S), \quad (1.14)$$

where P is given by

$$P = \begin{cases} S \rightarrow \alpha \\ \alpha \rightarrow A\alpha T \mid T\alpha A \\ \alpha \rightarrow G\alpha C \mid C\alpha G \\ \alpha \rightarrow \beta \mid e \\ \beta \rightarrow A\beta \mid T\beta \mid G\beta \mid C\beta \mid e \end{cases} \quad (1.15)$$

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.

$$\begin{aligned} S &\Rightarrow \alpha \\ &\Rightarrow A\alpha T \\ &\Rightarrow AG\alpha CT \\ &\Rightarrow AGG\alpha CCT \\ &\Rightarrow AGGC\alpha GCCT \\ &\Rightarrow AGGCT\alpha AGCCT \\ &\Rightarrow AGGCT\beta AGCCT \\ &\Rightarrow AGGCTA\beta AGCCT \\ &\Rightarrow AGGCTAAGCCT \end{aligned} \quad (1.16)$$

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 $A \rightarrow xB$, where A and B are in N and x is any string in Σ^* ;
2. context-free, or type-2, if each production in P is of the form $A \rightarrow \alpha$, where A is in N and α is in $(N \cup \Sigma)^*$;
3. context-sensitive, or type-1, if each production in P is of the form $\alpha A \beta \rightarrow \delta y \Gamma$, where A is in N , y is non-null, and α , β , δ , and γ 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:

$$G = (\{\gamma, \beta, S\}, \{\clubsuit, \heartsuit, \spadesuit\}, P, S), \quad (1.17)$$

where P is given by

$$P = \begin{cases} S \rightarrow \gamma \\ \gamma \rightarrow \clubsuit \heartsuit \spadesuit \mid \clubsuit \gamma \beta \spadesuit \\ \spadesuit \beta \rightarrow \beta \spadesuit \\ \heartsuit \beta \rightarrow \heartsuit \heartsuit \end{cases} \quad (1.18)$$

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

A)

<u>Language</u>		<u>Grammar</u>
English:	Dick saw Jane help Mary draw pictures	regular
German:	Dick Jane Mary pictures draw help saw	context-free
Dutch:	Dick Jane Mary pictures saw help draw	context-sensitive

B)

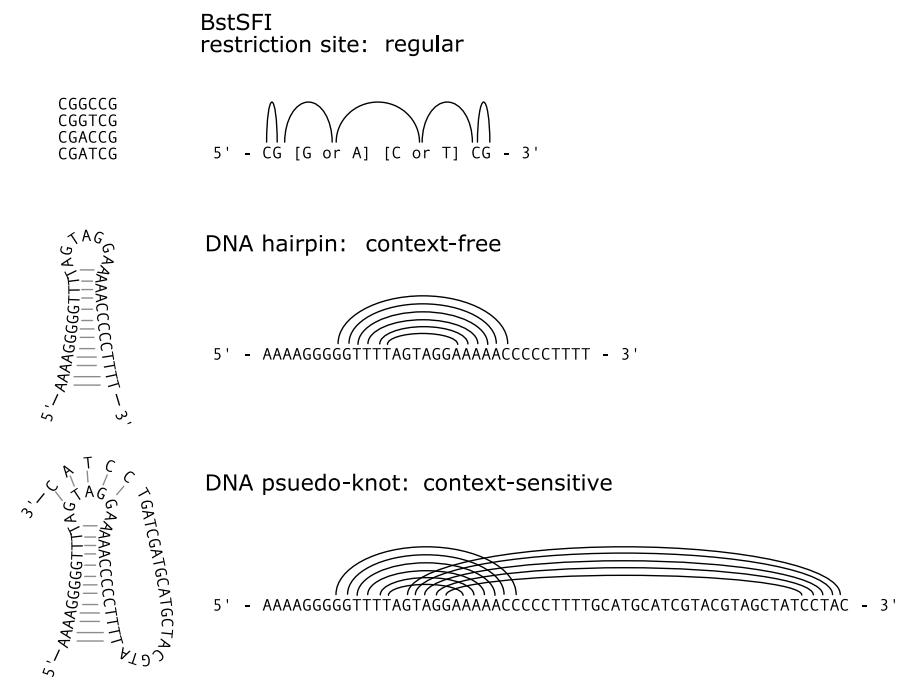


Figure 1-7: Noun–verb dependencies in various languages and their biological analogues. Part A) shows the sentence “Dick saw Jane help Mary draw pictures” translated grammatically into German and Dutch. That is, the words in the sentence are rearranged to reflect the rules of grammar in these two languages, 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:

$$\begin{aligned}
 S &\Rightarrow \gamma \\
 &\Rightarrow \clubsuit \gamma \beta \spadesuit \\
 &\Rightarrow \clubsuit \clubsuit \gamma \beta \spadesuit \beta \spadesuit \\
 &\Rightarrow \clubsuit \clubsuit \clubsuit \gamma \beta \spadesuit \beta \spadesuit \beta \spadesuit \\
 &\Rightarrow \clubsuit \clubsuit \clubsuit \clubsuit \heartsuit \beta \spadesuit \beta \spadesuit \beta \spadesuit \beta \spadesuit \\
 &\Rightarrow \clubsuit \clubsuit \clubsuit \clubsuit \heartsuit \beta \spadesuit \beta \spadesuit \beta \beta \spadesuit \beta \spadesuit \\
 &\Rightarrow \clubsuit \clubsuit \clubsuit \clubsuit \heartsuit \beta \spadesuit \beta \beta \spadesuit \beta \spadesuit \\
 &\Rightarrow \clubsuit \clubsuit \clubsuit \clubsuit \heartsuit \beta \beta \beta \spadesuit \beta \spadesuit \\
 &\Rightarrow \clubsuit \clubsuit \clubsuit \clubsuit \heartsuit \beta \beta \beta \beta \spadesuit \beta \spadesuit \\
 &\Rightarrow \clubsuit \clubsuit \clubsuit \clubsuit \heartsuit \heartsuit \beta \beta \beta \beta \spadesuit \beta \spadesuit \\
 &\Rightarrow \clubsuit \clubsuit \clubsuit \clubsuit \heartsuit \heartsuit \heartsuit \beta \beta \beta \beta \spadesuit \beta \spadesuit \\
 &\Rightarrow \clubsuit \clubsuit \clubsuit \clubsuit \heartsuit \heartsuit \heartsuit \heartsuit \beta \beta \beta \beta \spadesuit \beta \spadesuit \quad (1.19)
 \end{aligned}$$

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.

$$\underbrace{\clubsuit \clubsuit \clubsuit \clubsuit \dots \clubsuit}_{\text{exactly } n \text{ clubs}} \underbrace{\heartsuit \heartsuit \heartsuit \heartsuit \dots \heartsuit}_{\text{exactly } n \text{ hearts}} \underbrace{\spadesuit \spadesuit \spadesuit \spadesuit \dots \spadesuit}_{\text{exactly } n \text{ spades}} \quad (1.20)$$

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.

CGGCCG

CGGTCTG

CGACCG

CGATCG

(1.21)

The sequences are described by the following regular grammar:

$$G = (\{\alpha, \beta, \gamma, S\}, \{A, G, T, C\}, P, S), \quad (1.22)$$

where P is given by

$$P = \left\{ \begin{array}{l} S \rightarrow CG\alpha \\ \alpha \rightarrow A\beta \mid G\beta \\ \beta \rightarrow C\gamma \mid T\gamma \\ \gamma \rightarrow CG \end{array} \right. \quad (1.23)$$

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.

$$[LIVF] [LIV] [RK] . (9, 20) WS . WS [FYW]. \quad (1.24)$$

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. Or, in the general case, that any of the characters in Σ are acceptable. (Recall that Σ 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 wild-card 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, KR_{RR}, KR_{RRR}. Note that the strings KR and KR_{RRR} 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 [233] and Friedl [84].

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.

FLGARRQLCVVFKLAAKFQVCSKAKWQLCVFPAGFGKV (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

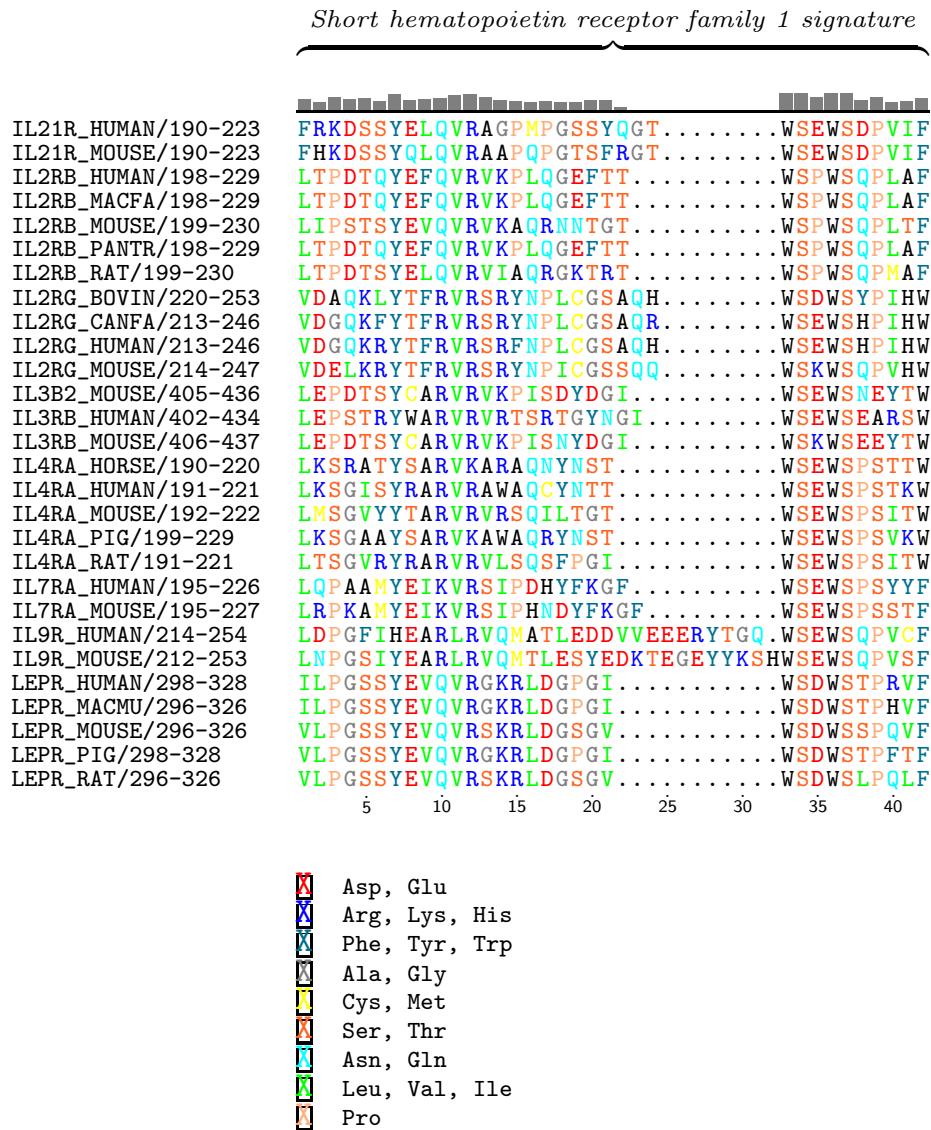


Figure 1-8: Regular grammar describing the short hematopoietin receptor family 1 signature [118]. These proteins are mostly receptors for interleukin cytokines. They are selectively expressed in lymphoid tissues and are typically membrane-bound [190]. The region shown in the figure is characterized by the regular expression `[LIVF].....[LIV][RK].(9,20)WS.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 stromal 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 physicochemical properties; and, the bracketed expressions, such as `[LIV]`, tend to group together amino acids with similar physicochemical properties.

of the regular expression in the sequence, which are underlined below.

FLGARRQLCVVFKLAAKFQVCSKAKWQLCVFPAVFGKV (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.

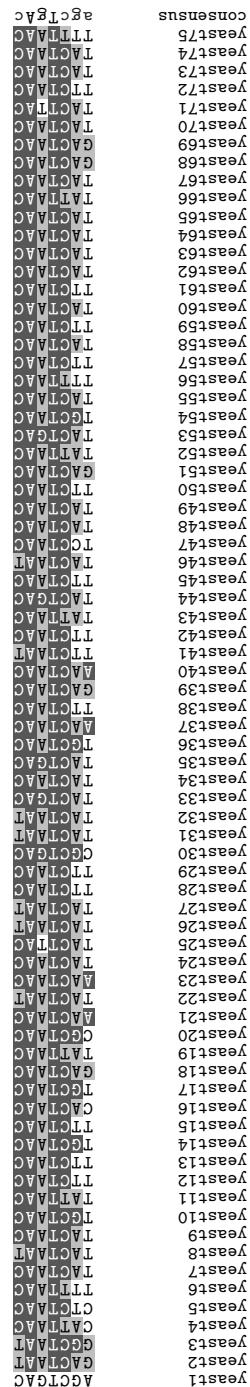


Figure 1-9: Yeast 3' splice sites. The figure shows the 3' splice junction of 75 introns from the fission yeast *Schizosaccharomyces pombe*. The coloring of the sequences is used to show the degree of conservation. Notice that certain columns, such as the final column, have strong preferences for certain 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 Σ^* . 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 Σ , 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,

$$G = (\{S, \alpha, \beta, \gamma, \delta, \epsilon, \zeta, \eta\}, \{A, T, G, C\}, P, S), \quad (1.27)$$

where P is the set of productions below.

$$P = \left\{ \begin{array}{l} S \rightarrow \alpha \\ \alpha \xrightarrow{0.067} A\beta \\ \alpha \xrightarrow{0.773} T\beta \\ \alpha \xrightarrow{0.093} G\beta \\ \alpha \xrightarrow{0.067} C\beta \\ \beta \xrightarrow{0.627} A\gamma \\ \beta \xrightarrow{0.240} T\gamma \\ \beta \xrightarrow{0.120} G\gamma \\ \beta \xrightarrow{0.013} C\gamma \\ \gamma \xrightarrow{0.120} T\delta \\ \gamma \xrightarrow{0.880} C\delta \\ \delta \xrightarrow{1.000} T\epsilon \\ \epsilon \xrightarrow{0.893} A\zeta \\ \epsilon \xrightarrow{0.027} T\zeta \\ \epsilon \xrightarrow{0.080} G\zeta \\ \zeta \xrightarrow{1.000} A\eta \\ \eta \xrightarrow{0.133} T \\ \eta \xrightarrow{0.867} C \end{array} \right. \quad (1.28)$$

Equation 1.15 can be represented much more compactly as a frequency matrix

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

in which each row corresponds to a single character in Σ and each column corresponds to a single non-terminal character in N (where Σ 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 η , 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] T [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, Θ , where

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

The calculation of the frequency and log-odds matrices for the 3' yeast splice sites is shown in Table 1.3. Here, Θ is given by

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

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 Θ . 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

$$\begin{aligned} \text{score} &= \Theta_{0,0} + \Theta_{2,1} + \Theta_{3,2} + \Theta_{1,3} + \Theta_{2,4} + \Theta_{0,5} + \Theta_{3,6} \\ &= -1.907 - 0.059 + 1.816 + 2.000 - 1.644 + 2.000 + 1.794 \quad (1.32) \\ &= 4.000. \end{aligned}$$

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

$$h_i = - \sum_j f_{ij} \log_2 f_{ij} \quad (1.33)$$

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:

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

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

$$h_i^o = - \sum_j q_j \log_2 q_j \quad (1.35)$$

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 $f_{ij} = (c_{ij} / \sum_j c_{ij})$. Part C) shows the log-odds matrix Θ , where each $\Theta_{ij} = \log_2(f_{ij}/q_j)$ 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 Θ and the columns are summed together.

A) Count Matrix (c_{ij}):

A	5	47	0	0	67	75	0
T	58	18	9	75	2	0	10
G	7	9	0	0	6	0	0
C	5	1	66	0	0	0	65

↓

B) Frequency Matrix (f_{ij}):

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

↓

C) Log-odds Matrix (Θ_{ij}):

A	-1.907	1.326	∅	∅	1.837	2.000	∅
T	1.629	-0.059	-1.059	2.000	-3.229	∅	-0.907
G	-1.421	-1.059	∅	∅	-1.644	∅	∅
C	-1.907	-4.229	1.816	∅	∅	∅	1.794

↓

D) Example sequence scoring:

query1	T	A	C	T	T	A	C
Σ	1.629	1.326	1.816	2.000	-3.229	2.000	1.794
					= 7.335		
query2	T	T	C	T	A	A	C
Σ	1.629	-0.059	1.816	2.000	1.837	2.000	1.794
					= 11.017		
query3	G	T	A	T	A	A	T
Σ	-1.421	-0.059	∅				= ∅

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

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

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

$$\begin{aligned} I &= H^o - H \\ &= - \sum_i \sum_j q_j \log_2 q_j - \left(- \sum_i \sum_j f_{ij} \log_2 f_{ij} \right) \\ &= \sum_i \sum_j f_{ij} \log_2 \left(\frac{f_{ij}}{q_j} \right). \end{aligned} \quad (1.37)$$

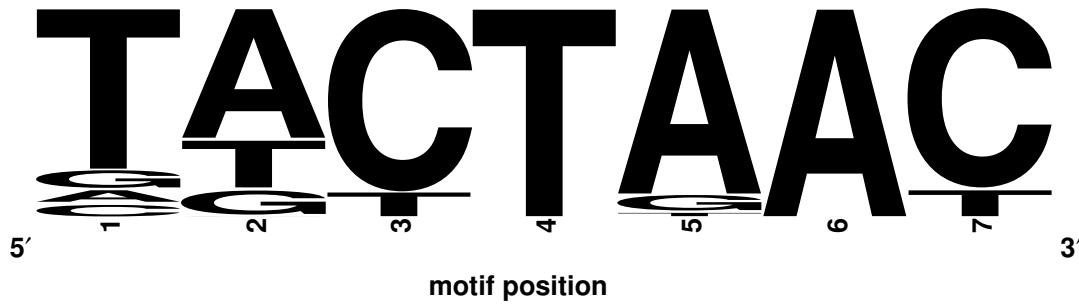
Notice that the information content of the motif is minimized when the nucleotide distribution for each column is exactly the background distribution of nucleotides. That is, when $f_{ij} = q_j$ for all i , the $\log_2(f_{ij}/q_j)$ terms are zero. This makes sense intuitively: if the PWM describes the 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



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

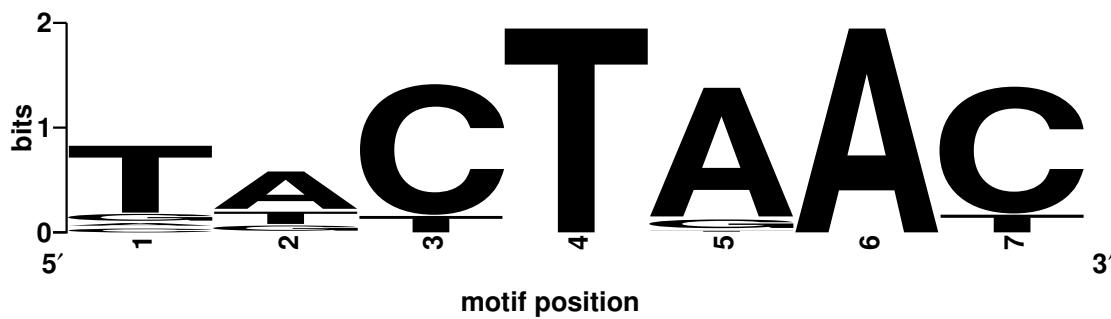


Figure 1-10: Yeast 3' splice site pictogram and logo. Part A) shows the PWM in Table 1.3 on page 52 represented as a pictogram. At each position in the motif, the height of the nucleotides is scaled in proportion to their frequency in f , with the more frequent nucleotides always placed on top. The pictogram 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.

AJDFIOASODVIKQTRPXYKIIWEJSJ

JKQTRPCRKXUCIQWEMFIOAKLGS (1.38)

ADUHFIKACRQTRPKMSKDAFIOAS

Without any prior knowledge, this task is nearly impossible. The colleague could have hidden a

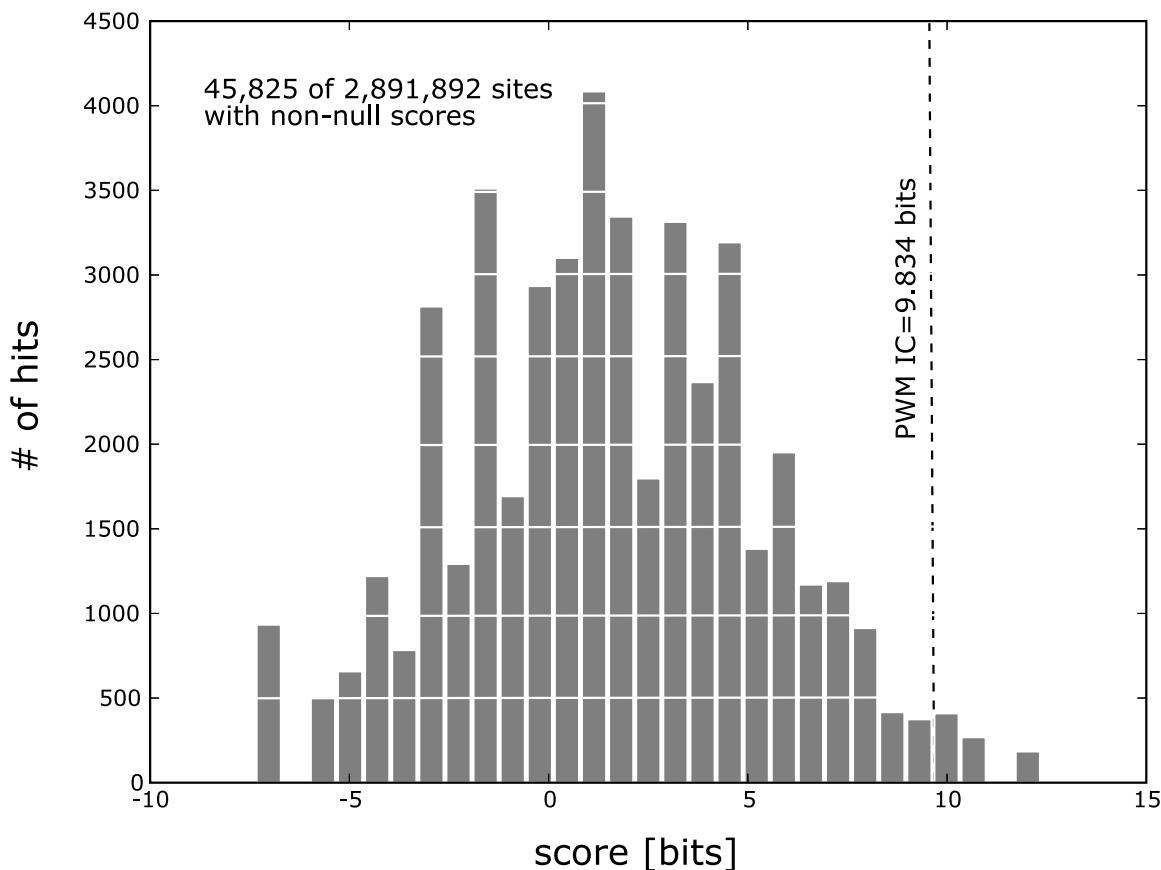


Figure 1-11: Distribution of log-odds scores obtained by searching the PWM in Table 1.3 on page 52 against chromosomes 1–4 of *Saccharomyces cerevisiae*. Of the nearly 3 million possible sites, only 46,000 had non-null scores. As the figure shows, the score distribution is roughly Gaussian. The dashed 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, *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.

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”.
- $p_i \in (\Sigma \cup \{\cdot\})^*$: 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].

Authors	Year	Citation
Queen et al.	1982	[202]
Galas et al.	1985	[87]
Mengeritsky and Smith	1987	[169]
Staden	1989	[237]
Neuwald and Green	1994	[179]
Jonassen et al.	1995	[132]
Wolferstetter et al.	1996	[270]
Sagot et al.	1997	[215]
Rigoutsos and Floratos	1998	[82, 207]
van Helden et al.	1998	[252, 253]
Jacobs Anderson and Parker	2000	[128]
Marsan and Sagot	2000	[167]
Pevzner and Sze	2000	[195]
Bussemaker et al.	2000	[47]
Kielbasa et al.	2001	[138]
Horton	2001	[123]
Keich and Pevzner	2002	[137]
Eskin and Pevzner	2002	[78]
Buhler and Tompa	2002	[46]
Sinha and Tompa	2002	[232]
Price et al.	2003	[199]
Sinha	2003	[231]
Danilova et al.	2003	[64]
Ganesh et al.	2003	[88]
Liang et al.	2004	[151]
Fogel et al.	2004	[83]
Pavesi et al.	2004	[191]
Hernandez et al.	2004	[114]
Markstein et al.	2004	[166]
Frith et al.	2004	[86]
Sumazin et al.	2005	[240]

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 $S = \{s_0, s_1, \dots, s_n\}$, 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 $\mathcal{C} = \{p_1, p_2, \dots, p_m\}$ and each $p_i \in \Sigma (\Sigma \cup \{.\}) \Sigma$. This is the set of all regular expressions that begin and end with a character, but may have an arbitrary number of wild-cards and characters in the middle subject to the L and W restriction, for example, AXG, 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”), $|\mathcal{L}(p)|$. 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 $|\mathcal{L}(p)|$ 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. 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

$$|\mathcal{L}(p_i)| \leq |\mathcal{L}(p_j)|. \quad (1.39)$$

Teiresias works in two phases: scanning and convolution. During the scanning phase, Teiresias enumerates all “elementary motifs” with exactly L characters and at most $W - L$ wild-cards (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

KDWVQKRK

CWCQKRK

WDQKRKNP

have five motifs with 1) exactly $L = 3$ characters, 2) no more than $W - L$ wild-cards for every window of $L = 3$ characters, and that 3) occur at least three times: $W.QK$, QKR , $QK.K$, KRK , and $Q.RK$. 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:

$$\begin{aligned} p_k &= p_i \oplus p_j \\ &= \begin{cases} p_k p'_i & \text{if } \text{suffix}_L(p_i) = \text{prefix}_L(p_j), \\ \emptyset & \text{otherwise.} \end{cases} \end{aligned} \quad (1.40)$$

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

$$\begin{aligned} \text{prefix}_3(W.QK) &= W.Q \\ \text{suffix}_3(W.QK) &= QK \\ \text{prefix}_3(QKR) &= QK \\ \text{suffix}_3(QKR) &= KR. \end{aligned} \quad (1.41)$$

```
>seq 0
KDWVQKRK
>seq 1
CWCQKRK
>seq 2
WDQKRKNP
```

\Downarrow
 Teiresias
 $L/W/K = 3/4/3$
 \Downarrow
 Elementary motifs:

motifs →	W.QK	QKR	QK.K	KRK	Q.RK
offset #0	KDWVQKRK (0,2)	KDWVQKRK (0,4)	KDWVQKRK (0,4)	KDWVQKRK (0,5)	KDWVQKRK (0,4)
offset #1	CWCQKRK (1,1)	CWCQKRK (1,3)	CWCQKRK (1,3)	CWCQKRK (1,4)	CWCQKRK (1,3)
offset #2	WDQKRKNP (2,0)	WDQKRKNP (2,2)	WDQKRKNP (2,2)	WDQKRKNP (2,3)	WDQKRKNP (2,2)

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:

$$\text{DF}.\text{A}.\text{T} \oplus \text{A}.\text{TSE} = \text{DF}.\text{A}.\text{TE}$$

$$\text{L}.\text{XF}.\text{A}.\text{MM} \oplus \text{A}.\text{MSE} = \text{L}.\text{XF}.\text{A}.\text{MME}$$

$$\text{WX}.\text{N}.\text{N} \oplus \text{N}.\text{PSE} = \emptyset.$$

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

$$\mathcal{L}(p_k) = \{(x, y) \in \mathcal{L}(p_i) \mid \exists (x, z) \in \mathcal{L}(p_j) \text{ such that } z - y = \mathcal{W}(p) - \mathcal{W}(\text{suffix}_L(p))\}. \quad (1.42)$$

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

$$|p_i \oplus p_j| < K \forall j, \quad (1.43)$$

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

>sequence 0		
MSKNIVLLPGDHVGPEVVAEAVKVLEAVSSAIGVKFNFSKHLIGGASIDAYGVPLSDEALEAAKK		
>sequence 1		
MSKQILVLPGDGIGPEIMAEAVKVLELANDRFQLGFELAEDVIGGAAIDKHGVP		
>sequence 2		
MKFLILLFNILCLFPVLAADNHGVGPQGASGVDPITFDINSNQTGPAFLT		
	↓	
	Teiresias:	
	$L/W/K = 5/8/2$	
	↓	
	Final motifs:	
	motif	location (seq,pos)
	GPE .. AEAVKVLE	(0,13) (1,13)
	IGGA . ID .. GVP	(0,42) (1,42)
	MSK . I .. LPGD .. GPE	(0,0) (1,0)
	A . D . HGV	(1,46) (2,17)

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 $p_3 = "A.D.HGV"$ the associated offset list is $\mathcal{L}(p_3) = \{(1, 46), (2, 17)\}$, 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 *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 $S = \{s_0, s_1, \dots, s_n\}$ and an integer $\mathcal{W}(p)$, 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].

Authors	Year	Citation
Stormo and Hartzell	1989	[238]
Lawrence et al.	1993	[147]
Liu	1994	[154]
Bailey and Elkan	1994	[19]
Leung et al.	1996	[149]
Goffeau	1998	[99]
Hertz and Stormo	1999	[115]
Workman and Stormo	2000	[273]
Hughes et al.	2000	[124]
GuhaThakurta and Stormo	2001	[101]
Bi and Rogan	2004	[35]
Raphael et al.	2004	[204]
Eskin	2004	[77]
Siddharthan et al.	2005	[229]
Liu et al.	2005	[156]
Leung and Chin	2005	[148]
Zhong et al.	2005	[282]
Tharakaraman et al.	2005	[243]
Down and Hubbard	2005	[70]
Macisaac et al.	2006	[159]

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].

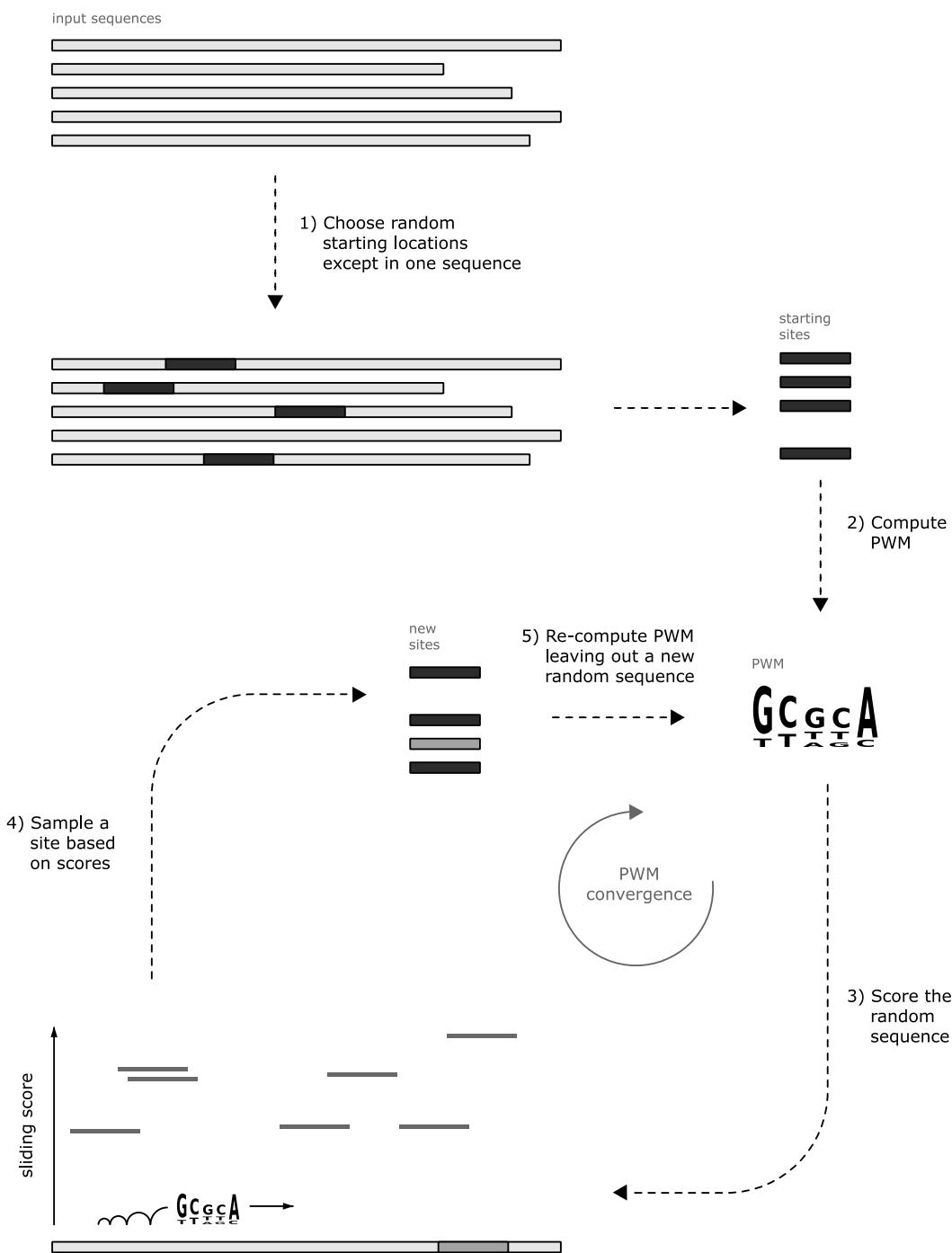


Figure 1-14: Schematic of the Gibbs sampling algorithm. As the figure shows, the Gibbs sampling method is an iterative algorithm that progressively refines a position weight matrix starting from a random PWM. If the input sequences contain a very strong motif, Gibbs sampling tends to converge very quickly 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, tetracyclin, or vanocomycin. 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-*

Aedes gambiae, the malaria mosquito, are upregulated after malaria (*Plasmodium berghei*) infection [58] and, in some cases, are capable of killing the ookinetes 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

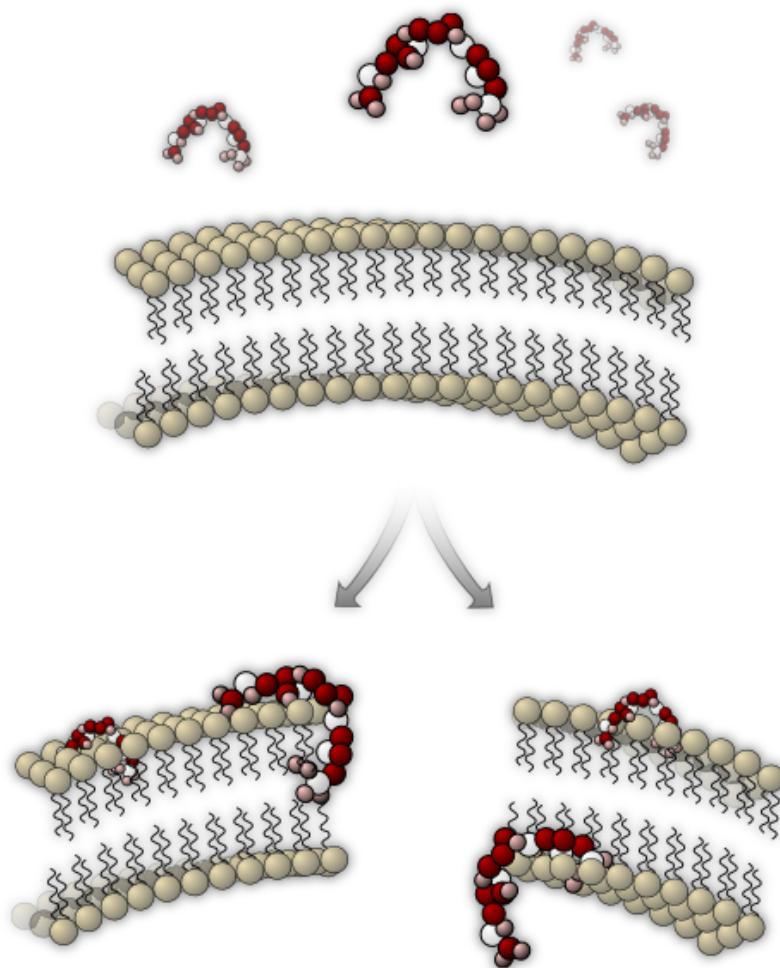
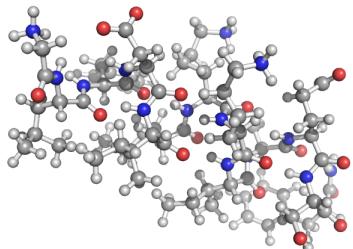
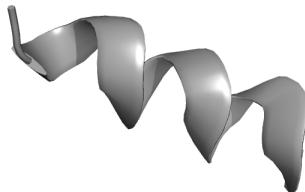


Figure 2-1: Antimicrobial peptide action. In the figure above, amphipathic antimicrobial peptides with net positive charges are attracted via electrostatic forces to the negatively charged outer-leaflet of the microbial membrane (step 1) [280]. This membrane is either the lipopolysaccharide layer or 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 phosphatydilserines of their outer leaflet are sequestered on the cytoplasmic side of the membrane, and 2) the membranes are stabilized by cholesterols [280].

A) Ball and stick



B) Ribbon



C) Peptide wheel view

PEPWHEEL of AUR12:LITRA from 1 to 13

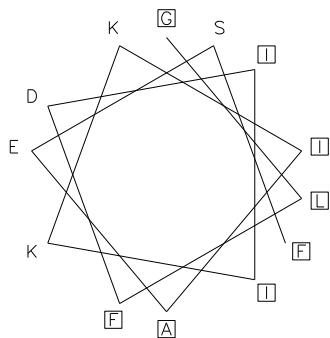


Figure 2-2: The structure of aurein-1.2 [261]. Aureins constitute a large family of secreted proteins originally isolated from the skin of frogs. This particular structure was isolated from the Australian Southern bell frog. The peptide conforms to the classic amphipathic alpha helical structure and 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).



Figure 2-3: A phylogenetic tree of amphibian AMPs. Because they tend to inhabit environments that are conducive to the growth of bacteria, amphibians are under evolutionary pressure to develop numerous and varied AMPs. This tree shows the degree of sequence similarity for over 400 AMPs isolated from amphibian sources. As the figure shows, amphibians have many families of AMPs, some of which are only distantly related, including the aureins, bombinins, bombesins, and brewinins.

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 *in vitro* bactericidal activity against pathogen strains that have developed a resistance to multiple conventional antibiotics [91, 239, 246].

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.

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

acaloleptin	achacin	adenoregulin	alpha-defensin
androctonin	andropin	apidaecin	attacin
aurein	azurocidin	bactenecin	bactericidin
bactinecin	beta-defensin	bombinin	bombolitin
buforin	buthinin	caerin	caltrin
cathelin	cecropin	ceratotoxin	citropin
clavanin	coleoptericin	corticostatin	crabrolin
defensin	demidefensin	dermaseptin	dermcidin
diptericin	drosocin	drosomycin	enbocin
formaecin	gaegurin	gallinacin	gloverin
granulysin	hadrurin	heliomicin	hemiptericin
hemolin	hepcidin	histatin	holotricin
hymenoptaecin	hyphancin	indolicidin	lebocin
macin	maculatin	maximin	metalnikowin
metchnikowin	misgurin	moricin	myticin
mytilin	mytimycin	nk-lysin	penaeidin
permatin	phormicin	phyllloxin	pleurocidin
polyphemusin	ponericin	protegrin	pseudin
pyrrhocoricin	ranalexin	ranatuerin	rhinocerosin
royalisin	rugosin	salmocidin	sapecin
sarcotoxin	sillucin	spingerin	styelin
tachycitin	tachyplesin	temporin	tenecin
termicin	thanatin	tricholongin	zeamatin

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 (S_o in Figure 2-4) from SwissProt/TrEMBL and AMSDb, we aligned each sequence in S_o against the entire SwissProt/TrEMBL database. If a sequence in SwissProt/TrEMBL aligned with a sequence from S_o 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 S_o^F in Figure 2-4, meaning sequences found using FastA in the first iteration.

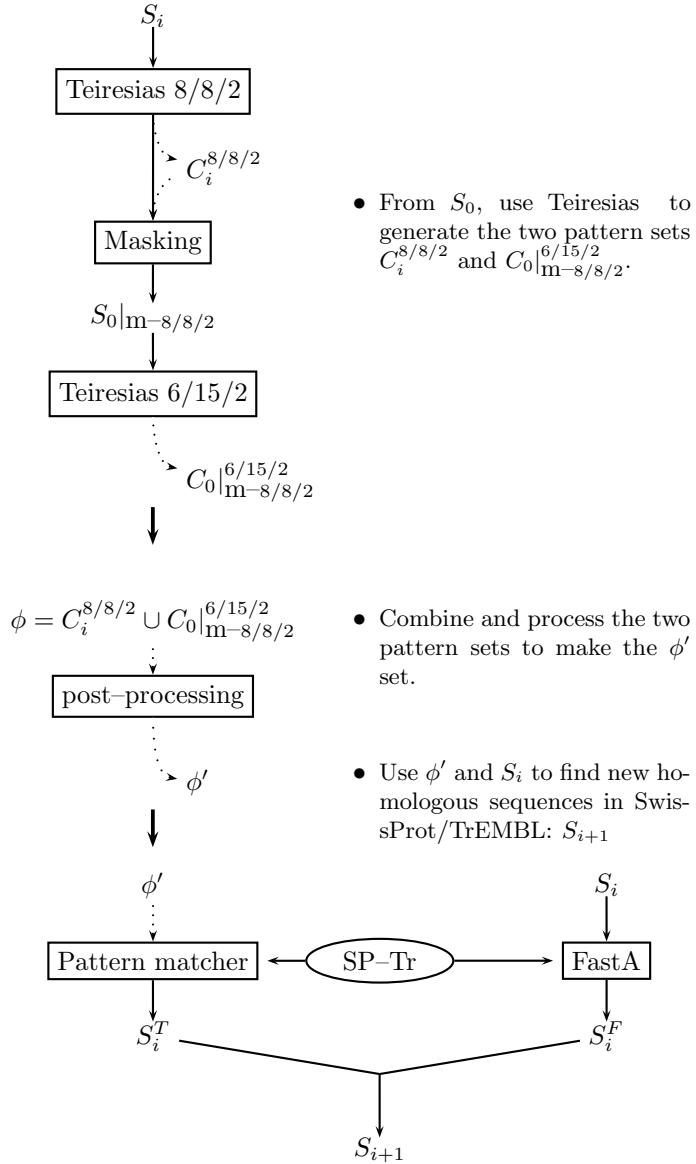


Figure 2-4: A schematic of the bootstrapping method used to collect antimicrobial sequences from SwissProt/TrEMBL. On the left, using TEIRESIAS, we computed an $8/8/2$ grammar set ($C_i^{8/8/2}$) from the initial set of sequences, S_0 . These grammars were masked from S_0 to make $S_0|m_{-8/8/2}$, from which the $6/15/2$ grammar set, $C_0|m_{-8/8/2}^{6/15/2}$, was found using TEIRESIAS again. The two grammar sets were combined and processed (see Appendix) to make ϕ' . 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_o sequence set, we derived two separate grammar sets ($C_i^{8/2/2}$ and $C_o |_{m-8/2/2}^{6/15/2}$), which we combined together. This combined set, ϕ , 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 ϕ' . If 80% of the amino acids in a peptide from SwissProt/TrEMBL were contained within instances of grammars from ϕ' 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_o^T .

Iterating the bootstrapping method

The new antimicrobial peptides found using FastA and TEIRESIAS, S_o^F and S_o^T respectively, were added to the initial database, S_o , 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, \dots) 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_o . These grammars are substrings in S_o that are repeated exactly, that is, grammars without any wild-cards or bracketed expressions. Let these grammars be called $C_o^{8/2/2}$, 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_o was masked¹ with the $C_o |^{8/2/2}$ grammars to make the $S_o |_{m-8/8/2}$ sequence set.

2. Finding Loosely Conserved grammars

¹“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 $S_o |_{m-8/8/2}$ sequences, we found all $8/15/2$ grammars, which we will call $C_o |_{m-8/2/2}^{6/15/2}$. These grammars are more loosely conserved than the $C_o |_{m-8/2/2}^{8/2/2}$ grammars and are typically greater in number.

3. Post-Processing the grammars

Let the union of the two grammar sets computed above be $\phi_o = C_o |_{m-8/2/2}^{8/2/2} \cup C_o |_{m-8/2/2}^{6/15/2}$. We would like to match grammars in ϕ_o against SwissProt/TrEMBL to find any remaining unknown antimicrobial sequences. But, to gain greater specificity and sensitivity, we first processed the grammars in ϕ_o to make a grammar set $\phi'o$.

- (a) For every grammar in ϕ_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 ϕ_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 “[RKM] [FWY] FK . [GQ] [KRQ] CPDAY”, each ten non-wild-card amino acids in length.
- (c) From this new, redundant ϕ_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 ϕ_o grammar set be called ϕ'_o .

The final sequence set, from the last iteration, became our database of known antimicrobial peptide sequences and the final ϕ' became our antimicrobial grammar database.

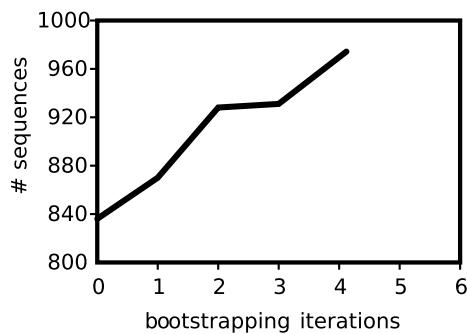


Figure 2-5: A plot of the progress of the bootstrapping method. The figure shows that our antimicrobial sequence database grew from 836 sequences to 931 sequences in 3 iterations.

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, S_3 , which contained a total of 931 sequences, was used as our antimicrobial sequence database and is available on-line at <http://cbcsrv.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.”

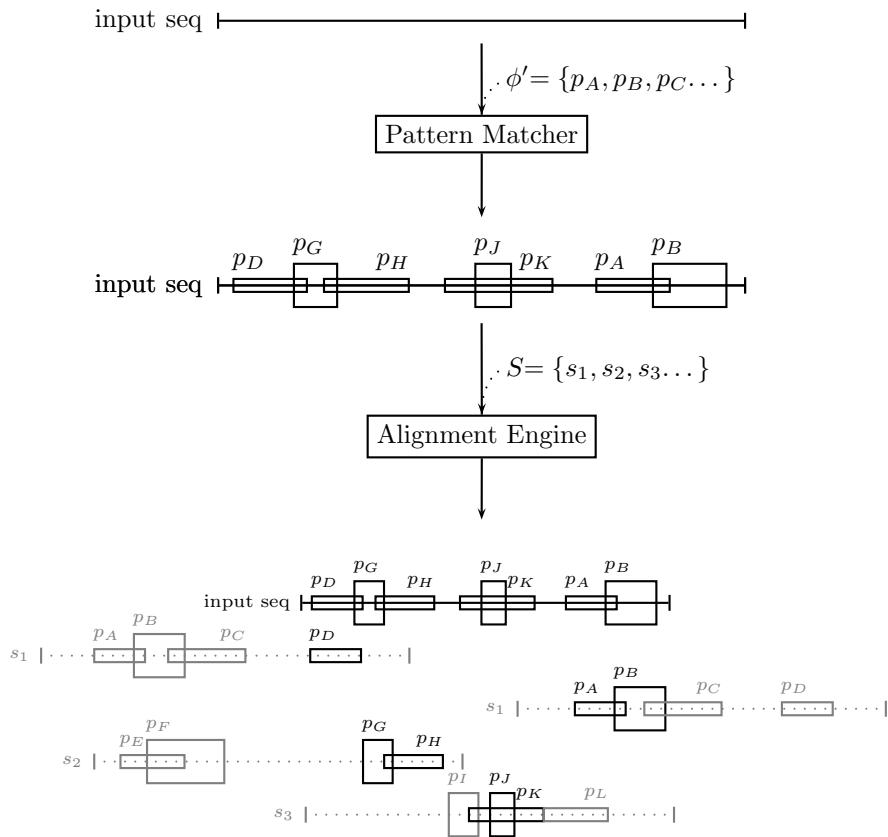


Figure 2-6: A schematic of our grammar-based alignment method. In the figure, a user-supplied input sequence is searched for instances of grammars ($p_A, p_B, p_C \dots$) that occur in the set of known antimicrobial sequences (S). Grammars that occur in both the input sequence and sequences in S are 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 <http://cbcdrv.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 ϕ'). 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.

Query sequence grammar-based alignments

Figure 2-7: Example results of the grammar-based alignment method. The figure shows the annotation of a query sequence after it has been searched exhaustively for grammars in our database of ~200K grammars describing the “language” of AmPs. This alignment was generated using the methodology detailed in Figure 2-6 on the page before. Each of the sequences below the query is an AmP sequence from our database, S. The sequences are aligned against the query because they share grammars in common at those particular loci. However, different sequences may share very in degrees of conservation, even if they have the same grammar. For example, see Figure 2.2 on the facing page.

Table 2.2: Motif conservation for the query shown in Figure and the motif L[VQH][ALV][KLPQ][AS][EAF][APQS][ALRV]QA.

QUERY	LQAQAEPLQA					
P17534	LVLLAAFQVQA	40.00%	Cryptdin-related protein 4C-1	Mus musculus (Mouse).		
P19660	LVLPSSASAQA	30.00%	Bactenecin 5 precursor (BAC ₅)	Bos taurus (Bovine).		
P19661	LVLPSSASAQA	30.00%	Bactenecin 7 precursor (BAC ₇)	Bos taurus (Bovine).		
P28309	LVLLSFFQVQA	30.00%	Cryptdin-2 precursor	Mus musculus (Mouse).		
P28310	LVLLAAFQVQA	40.00%	Cryptdin-3 precursor	Mus musculus (Mouse).		
P28311	LVLLAAFQVQA	40.00%	Cryptdin-4 precursor	Mus musculus (Mouse).		
P28312	LVLLAAFQVQA	40.00%	Cryptdin-5 precursor	Mus musculus (Mouse).		
P32195	LVVPSSSASAQA	30.00%	Protegrin 2 precursor (PG-2)	Sus scrofa (Pig).		
P33046	LVVPSSSASAQA	30.00%	Indolicidin precursor	Bos taurus (Bovine).		
P49930	LVVPSSSASAQA	30.00%	Antibacterial peptide PMAP-23	Sus scrofa (Pig).		
P49931	LVVPSSSASAQA	30.00%	Antibacterial peptide PMAP-36	Sus scrofa (Pig).		
P49932	LVVPSSSASAQA	30.00%	Antibacterial peptide PMAP-37	Sus scrofa (Pig).		
P50704	LVLLAAFQVQA	40.00%	Cryptdin-6/12 precursor	Mus musculus (Mouse).		
P50705	LVLLAAFQVQA	40.00%	Cryptdin-7 precursor	Mus musculus (Mouse).		
P50707	LVLLAAFQVQA	40.00%	Cryptdin-9 precursor	Mus musculus (Mouse).		
P50708	LVLLAAFQVQA	40.00%	Cryptdin-10 precursor (Fragmen	Mus musculus (Mouse).		
P50711	LVLLAAFQVQA	40.00%	Cryptdin-13 precursor	Mus musculus (Mouse).		
P50712	LVLLAAFQVQA	40.00%	Cryptdin-14 precursor (Fragmen	Mus musculus (Mouse).		
P50713	LVLLAAFQVQA	40.00%	Cryptdin-15 precursor	Mus musculus (Mouse).		
P50714	LVLLAAFQVQA	40.00%	Cryptdin-16 precursor	Mus musculus (Mouse).		
P51525	LVVPSSSASAQA	30.00%	Prophenin-2 precursor (PF-2) (Sus scrofa (Pig).		
P82270	LHAQAAEARQA	70.00%	Theta defensin-1, subunit A pr	Macaca mulatta (Rhesus macaque).		
P82271	LHAQAAEARQA	70.00%	Theta defensin-1, subunit B pr	Macaca mulatta (Rhesus macaque).		
P82318	LQAQAEPLQA	100.00%	Neutrophil defensins 1, 3 and	Macaca mulatta (Rhesus macaque).		
Q01524	LQAKAEPPLQA	90.00%	Defensin 6 precursor (Defensin	Homo sapiens (Human).		

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 [109].) 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 ($E\text{-value} \leq 10$), including the naturally-occurring AmPs. (More extensive similarity searching using PSI-Blast and E-value thresholds up to ≤ 50 also failed to detect similarity to any natural AmPs.) This is possible because each

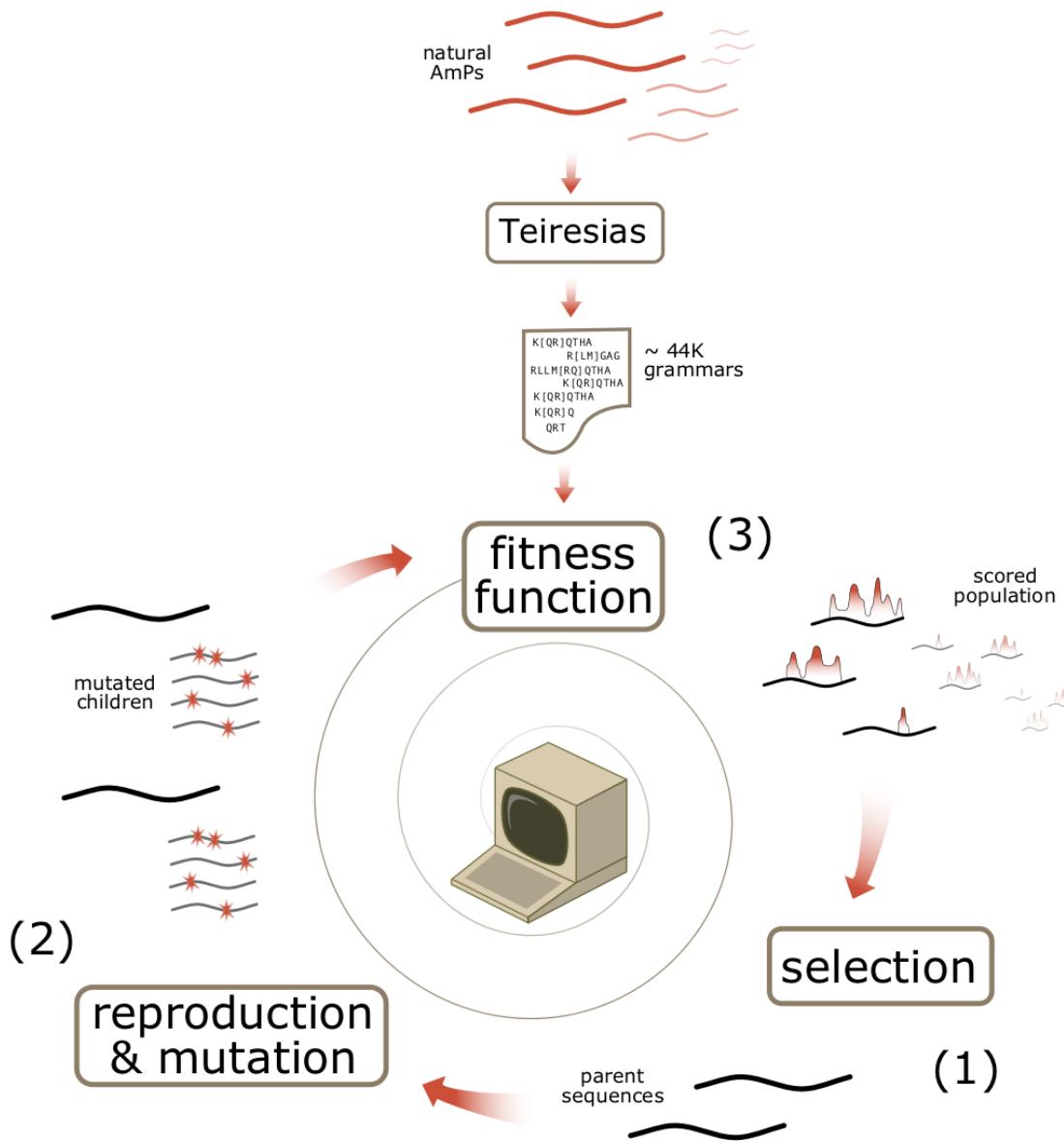


Figure 2-8: A schematic of the *in silico* directed evolution strategy. Position (1) shows the starting point: the database of 100K parental sequences. Each of these sequences has 4 mutated children (2) and the entire population is scored using the Z-score and our database of grammars from natural AmPs. 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.

Peptide	S	Size	Charge	pI	Sequence
synth-1:					
synth-1	1	20	4.5	11.92	NKVKKPLTGAHRLLLFTTFLFV
negative-1a	0				VVLLKLLFFKFNLPHKTRRTAG
negative-1b	0				LVLTLFLFATPKLNGRVKKFH
synth-2:					
synth-2	1	31	10.0	11.28	MKKIKKEAGKNIKLAPKEVAAKKSKKSPTK
negative-2a	0				PAAGESKVVKANIKKKAKILPTMKLKKEIKKS
negative-2b	0				SEASLIKAKIKKIAMKKVTKGKAKNKPPLPEK
synth-3:					
synth-3	0.92	63	3.0	10.41	MKDKNSTGPILLSSALLIAVTAGGSPVAAAPWNPFAAILKAALQIAGAAEPKEVITAKKGPTKADA
negative-3a	0				GWAGLIVVAETAIADKMSLKAAGEPPNQNDGAVLKTPPKAAASAKPLGAAKTLAFISPVTLALAK

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 microdilution 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, “ungrammatical” 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-

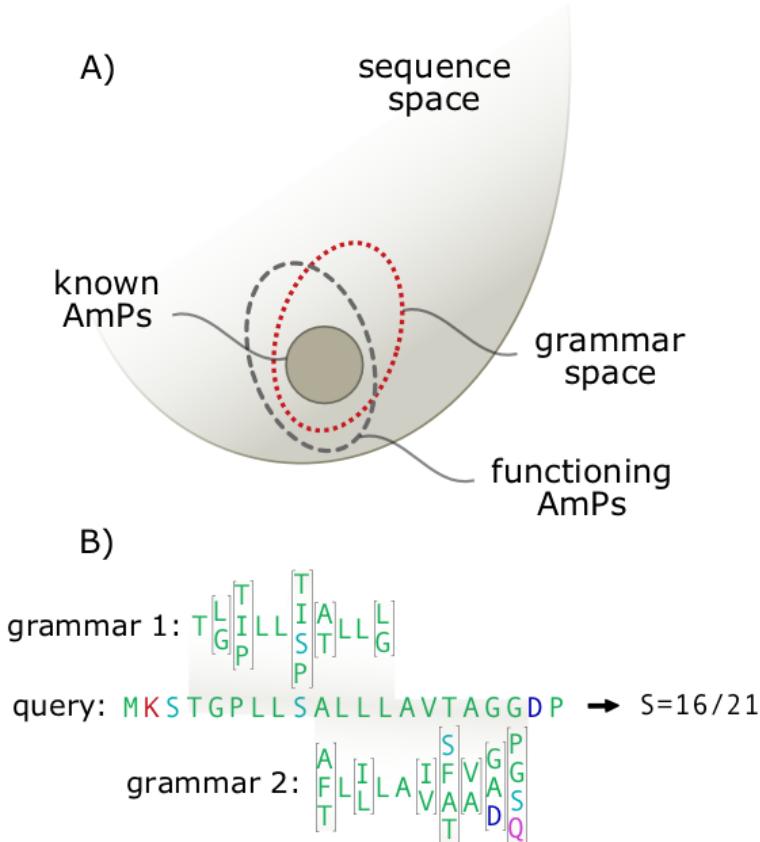


Figure 2-9: The AmP design space. Part A shows the sequence space surrounding the set of natural AmPs. The “sequence space” is the combinatorially large set of all possible sequences. Even for a 20-residue peptide like synth-1 (see Table 2.3) this space is huge: $20^{20} \approx 10^{26}$ sequences. (For 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 ($\text{MIC}_{80} \leq 8 \mu\text{g/mL}$). 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 $\mu\text{g/mL}$ (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 $\mu\text{g/mL}$, preventing 99% of the bacterial growth with an MIC_{80} of roughly 40 $\mu\text{g/mL}$. 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 $\mu\text{g/mL}$.

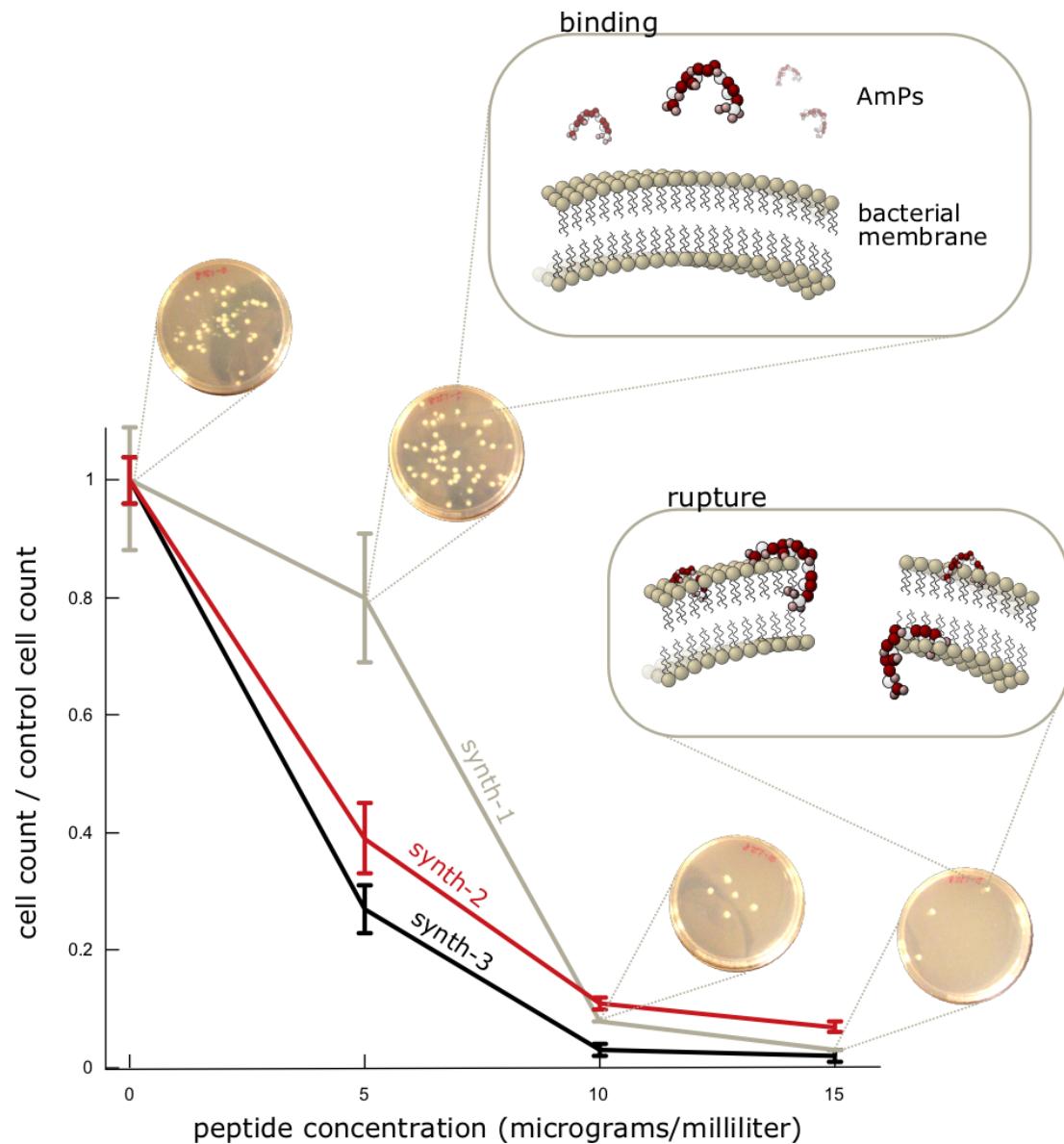


Figure 2-10: Bacteriostatic activity of the three synthetic AmPs against *B. cereus*. The break-outs show photographs of the colonies, which decreased in number with increasing peptide concentration. The inlaid schematic shows the generally accepted mechanism of AmP action: the electrostatic affinity for the outer-leaflet of the bacterial membrane leads to binding and rupture of the cell [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.)

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: the ratio of the cell count at a particular concentration of AmP to the cell count at 0 µg/mL. The entries in dark gray show high viability (low antimicrobial action) and the white entries show low viability (high antimicrobial action). The names prepended with a “*” are peptides that were chemically synthesized rather than produced via *in vitro* translation.

species →	<i>B. subtilis</i>					<i>C. glutamicum</i>					<i>E. coli</i>					<i>C. rodentium</i>				
peptide conc. (µg/mL) →	0	5	10	15	0	5	10	15	0	5	10	15	0	5	10	15	0	5	10	15
synth-1:																				
synth-1	1.00	0.80	0.08	0.03	1.00	0.73	0.21	0.15	1.00	1.00	1.00	0.93	1.00	0.99	0.97	0.98				
negative-1a	1.00	0.96	1.00	1.00	1.00	1.00	1.00	0.93	1.00	1.00	1.00	0.99	1.00	1.00	0.99	1.00	0.99	1.00	0.98	
negative-1b	1.00	0.98	1.00	1.00	1.00	1.00	1.00	0.96	0.99	1.00	1.00	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	
synth-2:																				
synth-2	1.00	0.27	0.03	0.02	1.00	0.28	0.21	0.12	1.00	1.00	1.00	0.99	1.00	0.99	0.99	0.97				
negative-2a	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.96	0.91	1.00	0.98	0.99	1.00	1.00	0.95	0.99	0.95			
negative-2b	1.00	1.00	1.00	1.00	1.00	1.00	0.92	0.95	0.98	1.00	0.98	0.97	0.97	1.00	0.98	0.94	0.99			
synth-3:																				
synth-3	1.00	0.39	0.11	0.07	1.00	0.40	0.26	0.18	1.00	1.00	1.00	0.99	1.00	1.00	0.98	0.10				
negative-3a	1.00	0.96	0.98	0.98	1.00	1.00	0.96	0.96	1.00	1.00	1.00	0.99	1.00	1.00	0.96	0.96				
Chemically synthesized:																				
*synth-1	1.00	0.72	0.14	0.14	1.00	0.56	0.16	0.11	1.00	0.89	0.92	0.86	1.00	0.90	0.90	0.90				
*negative-1a	1.00	1.00	0.84	0.87	1.00	1.00	1.00	1.00	1.00	0.92	0.90	0.94	1.00	1.00	0.98	1.00				
*negative-1b	1.00	0.92	0.98	0.96	1.00	1.00	0.96	1.00	1.00	1.00	1.00	0.97	1.00	1.00	1.00	1.00				

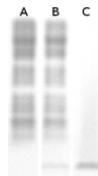


Figure 2-11: SDS-PAGE gel showing the synth-1 *in vitro* translation product (lane B). Lane A shows the translation mixture with no peptide and lane C shows the *synth-1 peptide, which was produced via solid phase synthesis and validated by mass spectroscopy.

Table 2.5: Minimum inhibitory concentration of the preliminary design synthetic AMPs against a variety of bacteria. In the table MIC₅₀ 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.

	synth-1		synth-2		synth-3	
	MIC ₅₀	MIC ₈₀	MIC ₅₀	MIC ₈₀	MIC ₅₀	MIC ₈₀
Gram-positive:						
<i>B. subtilis</i>	7.5	10	3.5	6	4.5	8.5
<i>C. glutamicum</i>	4	13.5	3.5	11	4	10
Gram-negative:						
<i>E. coli</i>	-	-	-	-	-	-
<i>C. rodentium</i>	-	-	-	-	12	15

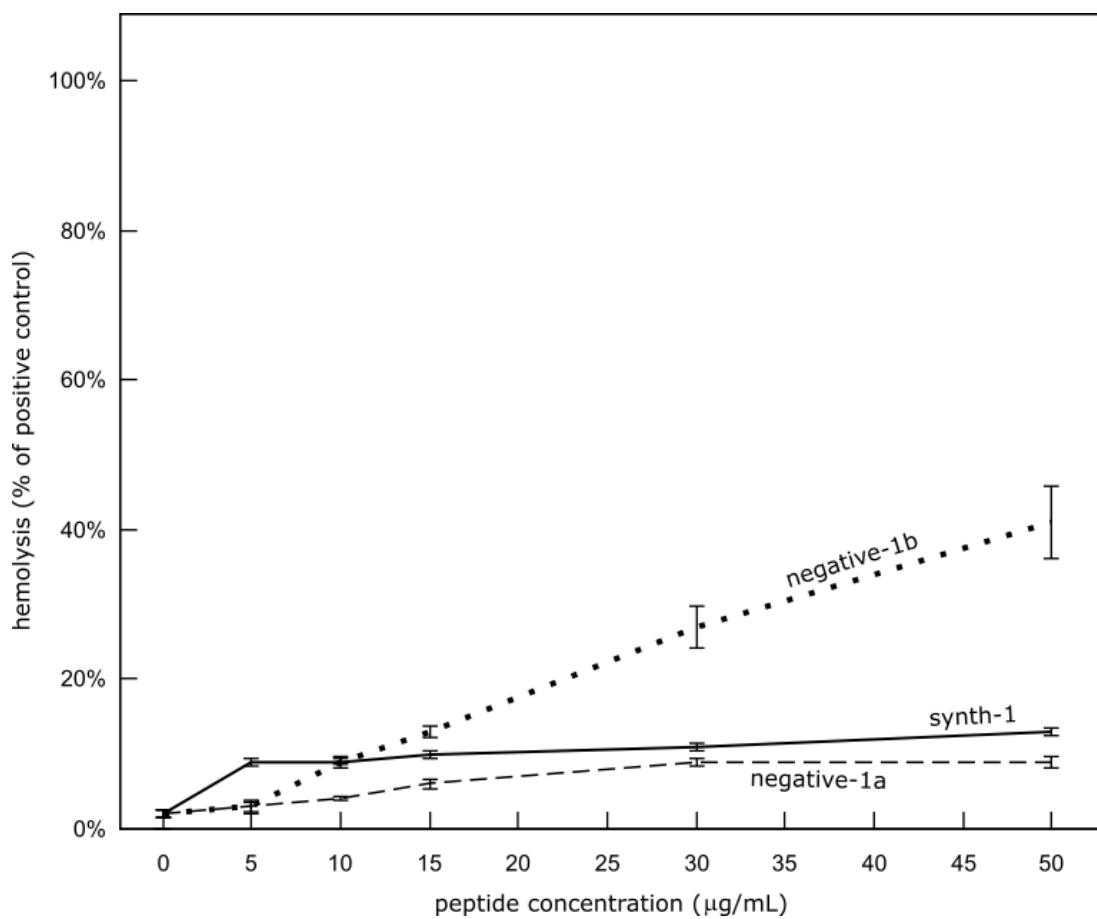


Figure 2-12: Activity of the synth-1 family of peptides against human erythrocytes determined using a procedure described elsewhere [153]. The ordinate shows the degree of hemolysis relative to 50 $\mu\text{g}/\text{mL}$ of melittin, which causes complete hemolysis.

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 HM_{50} of approximately 60 $\mu\text{g}/\text{mL}$. *Synth-1 and *negative-1b were less active against erythrocytes, with HM_{50} concentrations (by extrapolation) of roughly 260 and 290 $\mu\text{g}/\text{mL}$, respectively.

2.4.3 Later experimentation

As mentioned briefly in the preceding sections, the experimental data suggesting that synth-1 had to 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 $L = 7$, $W = 15$, $K = 5$ with the following amino acid equivalency groups $[[AG], [DE], [FYW], [KR], [ILMV], [QN], [ST]]$. 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 de-referenced 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-

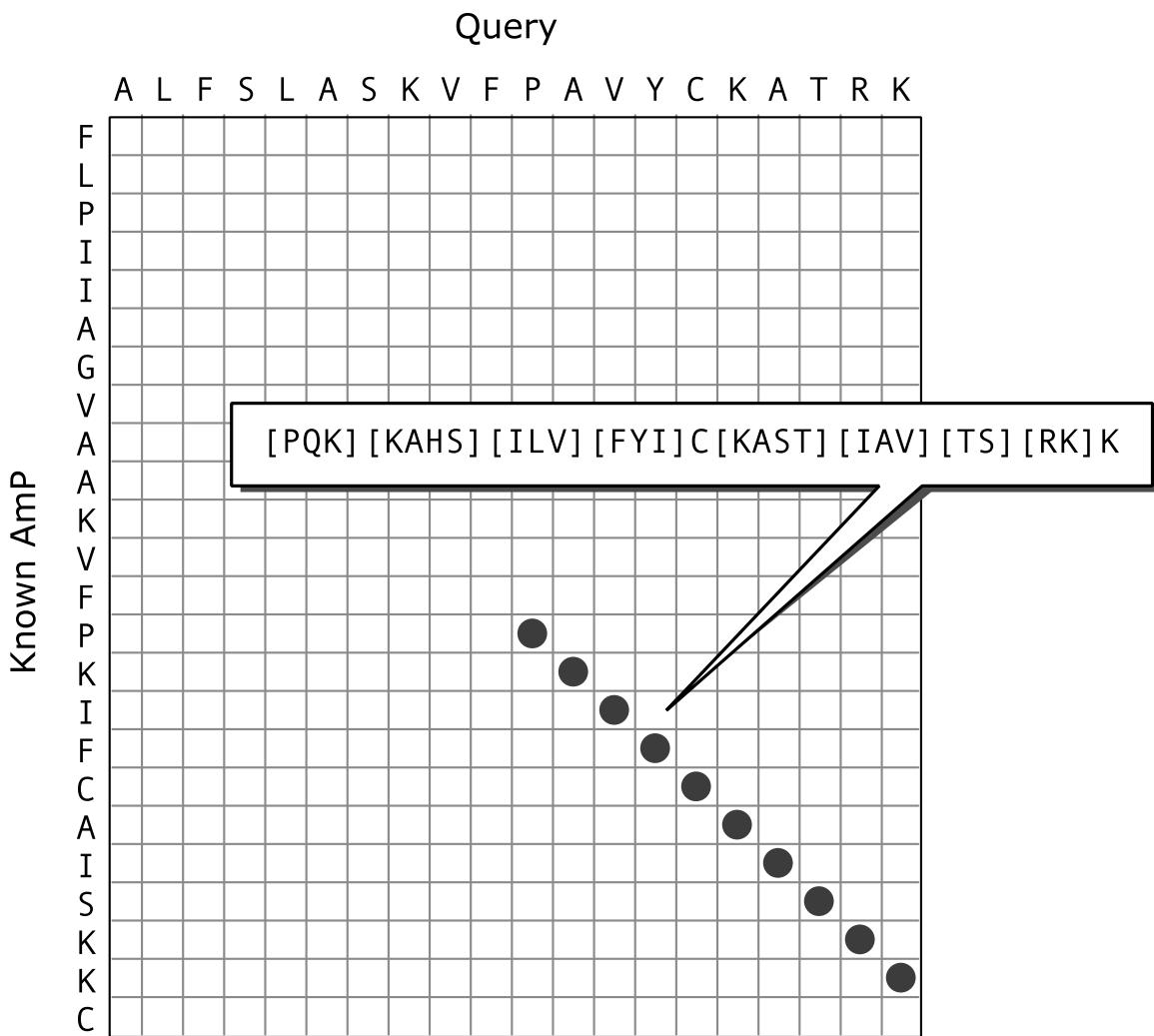


Figure 2-13: An example grammar-based dot plot. The figure shows a query sequence all the top and a single known Amp sequence on the vertical axis along the side. The matrix has an entry for each pair of amino acids between the two sequences. The breakout shows a grammar from our database that matches 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.

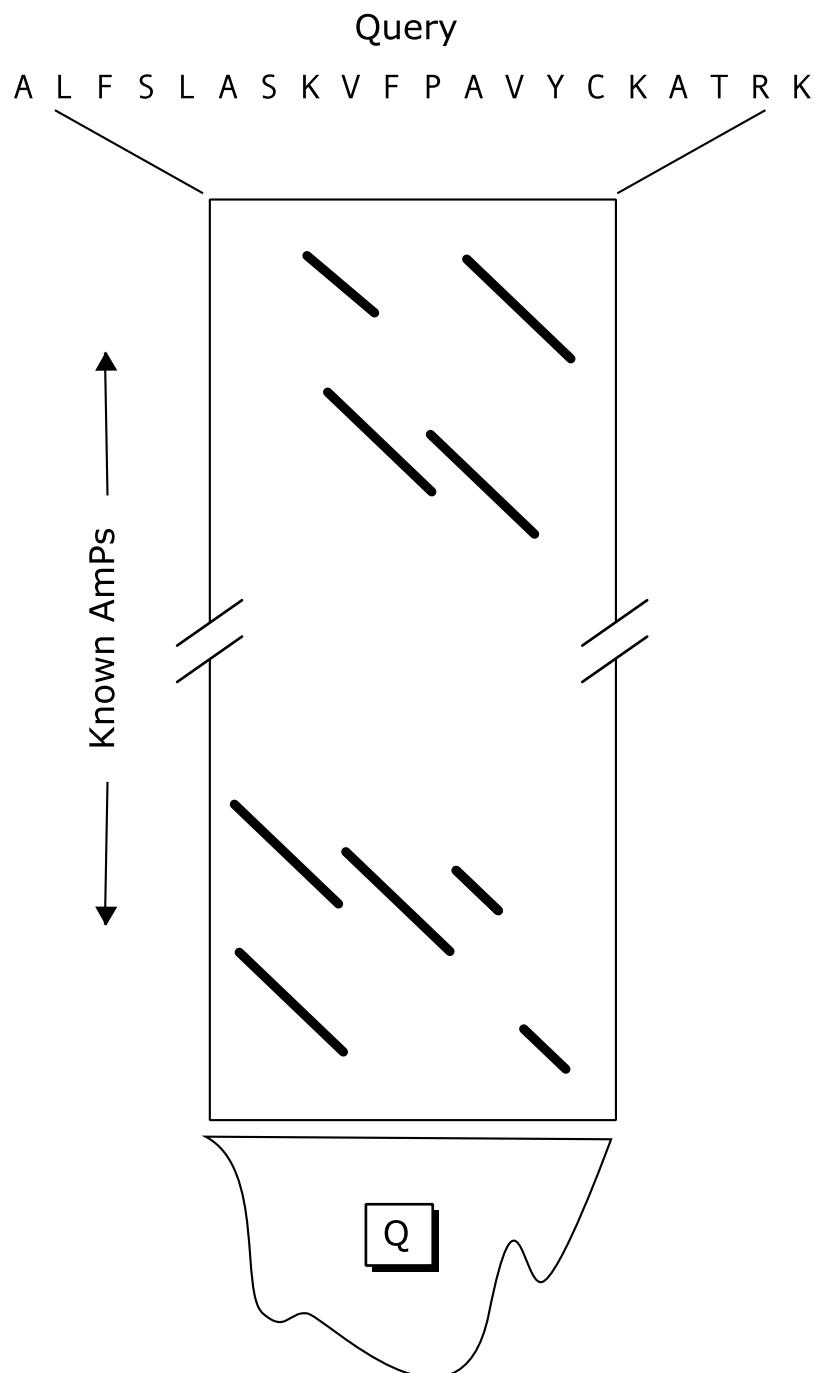


Figure 2-14: An example grammar-based dot plot for computing Q . The figure shows a “zoomed out” view of many dot matrices concatenated together. (See the dot matrix shown in Figure 2-13 on the page before.) At the top of the matrix is the query sequence, which is the synthetic, hypothetical AmP that 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 Multipep 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 × 10⁵ 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 (FLGVVFKLASKVFPAGFGKV) 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 $\mu\text{g}/\text{mL}$. “+” = MIC greater than $256 \mu\text{g}/\text{mL}$. ++ = MIC greater than $128 \mu\text{g}/\text{mL}$, not sufficiently soluble to test at $256 \mu\text{g}/\text{mL}$.

Peptide	Sequence	<i>E. coli</i>	<i>B. subtilis</i>	Shuffled Sequence	<i>E. coli</i>	<i>B. subtilis</i>
1	ALFSLASKVVPSPVFSMVTKK	+	+	MVVFSPKFKSTVAKLLSSA	+	+
2	VVFRVASKVFPAPVYCTVSKK	128	+	TAKVVVFVFSYVVPKKRAC	+	+
5	FLFGLASKVFPAPVYCKVTRK	64	256	FLPVLVKVFYRSKKTAAGCF	++	64
6	LSAVGKIASKVVPSPVIGAFK	+	+	GVSSPIAVAKFKGAVASLIK	+	+
7	PVIGKLASKVVPSPVFSMIKR	+	+	SRVPLKSPVKIVGSKVMIFA	+	+
9	GLMSILVKDIAKLAAKQGAKQ	256	+	GLKKDALQSIVKKAQIAAMG	+	+
15	SALGRVASKVFPAPVYCSITK	+	+	LYSPTCVKAAVSRFIGKVSA	+	+
22	LGALFRVASKVFPAPISMVK	256	64	SVPSVGAVLFFKRAAVMKLI	+	+
23	ALGKLASKVFPAPVYCTISRK	128	+	KYGPALVIAVKKCSCLTFRA	+	+
24	GFIGKLASKVVPSPVSYCKVTG	128	+	GGSTLGVFVKKSKACVIVPY	Not soluble	
25	PVVFSVASKVVPSPVLSALKR	+	+	KSPFVLVVSSRVAVIKSLP	+	+
28	FLGVVFKLASKVFPAPFGKV	64	16	GVSVAGAKKVVLVFPFLF	+	+
29	PAVFKIASKVVPSPVSYCKVSR	128	+	KVYVVKIAVPCFPKSARSVS	+	+
30	GALFGLASKVFPAPVGFACK	256	+	KVVLFGAAGAKLFKASFFGP	Not enough material	
31	SAVGKLASKVFPAPVFSMVTK	+	+	FMKVLAVFGSVTSAPKASK	+	+
33	VKDLAKFIAKTVAKQGGCYL	++	++	ALVYAGIKKTAFLKVQKCDG	+	+
34	GVVGKLASKVVPSPVFGSFTK	+	+	SVKPGVSSVKGTLVKKFFG	+	+
35	LPVVFRVASKVFPALISKLT	+	256	KVFIATLVSFSLLAKPPRV	+	+
36	SAVGSVASKVVPSPSLISKVTK	+	+	STVKVASKLAVVVSPISKGS	+	+
39	MKSIAKFIAKTVAKQGAKQG	+	+	AKKAQKSGAQTIIVKIFAKGM	+	+
42	LPAVFKLASKVVPSPVFGLVK	+	+	VVAKKFFFVLVKGAPVLSPS	+	+
43	SFVFKLASKVVPSPVSALTR	256	256	ASPTVFRSSVFLSFLVVAKK	+	+
44	SVIGKIASKVVPSPVYCAISK	+	+	IASAVPVCVKGKISKSYSIV	+	+
45	PVVGVRVASKVFPAPIVGLVK	+	+	VKRAGKGVAVVPSPPLFKIVV	+	+
51	FLFRVASKVFPALIGKFKK	64	16	RKVAPALIKSFVFLFKFKKG	+	+
55	LSFVGRVASKVVPSPSLISMK	256	+	SSSIPIKMVLVRALVFVKSG	+	+
56	SALGRLASKVVPAPIVGKTT	+	+	TLVGVVAKLVATKIGSSPRA	+	+
57	LGVVGSLASKVVPAPISKVK	+	+	PKVVGSLIVVVAKVSSALG	+	+
62	LPAVFKLASKVFPAPVYCKAS	128	+	PSLLYKAKAVFCKPSAVAVF	++	++
63	LPVLFKLASKVFPAPVFSLLK	256	64	VSVKKVLPFAPLKSLLSFAF	256	256
65	VVGRVASKVVPSPSLIGLFTTK	+	+	FKVVISKPGLSVRGVTALVT	++	++
69	SVVFGVASKVVPSPVIGKVKT	+	+	VFSVKGGKPSVVIKVVVAST	+	+
75	FLPFVGRVRIASKVVPSPVIGKV	+	+	SKFPLAGIFSVPVGKRVVVI	+	+
77	GKKLAKTIAKEVAKQGAKFA	64	+	VIAFAKTKEAKAKLKGQAKG	+	+
81	PFVGRVASKVVPSPVYCAITR	Not soluble		PAVYKSIVGFSPVARVTVCR	Not soluble	
82	FVGSLASKVVPSPVFGAIKTK	+	+	KTPVVLKASIKVSSAGFGF	+	+
83	LPVVFKIASKVVPSPVSKIT	+	+	KIVKVITVKSISPASLVPVF	++	++
84	GAVFGVASKVVPSPVFSAIKK	+	+	SVKVAKSIPSAVFGAGKVF	+	+
85	FVGGVASKVVPSPVYCKVSKK	+	+	KVGKGSYPCSFVKVVAKVS	+	+
88	VVFKLASKVVPSPVYCTITKK	256	+	VTKCSVPAVVYILVKTFKS	+	+
96	GALFSLASKVVPAPIVGLIKK	256	+	LPVLFSSAIAKVGIGLGAKV	+	+

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 $\mu\text{g}/\text{mL}$. “+” = MIC greater than $256 \mu\text{g}/\text{mL}$. ++ = MIC greater than $128 \mu\text{g}/\text{mL}$, not sufficiently soluble to test at $256 \mu\text{g}/\text{mL}$.

Peptide	Sequence	<i>S. aureus</i>	<i>B. anthracis</i>	Shuffled Sequence	<i>S. aureus</i>	<i>B. anthracis</i>
28	FLGVVFKLASKVFPAPVFGKV	8	16	GVSVAGAKKVKVLVFPFLF	+	+
51	FLFRVASKVFPALIGKFKKK	16	16	RKVAPALIKSFVFLFKFKKG	128	256
22	LGALFRVASKVFPAVISMVK	64	64	SVPSVGAVLFKRAAVMKLI	+	+
63	LPVLFKLASKVFPAVFSSLK	128	128	VSVKKVLPFAPLKSLLSFAF	+	+
5	FLFGLASKVFPAPVYCKVTRK	256	128	FLPVLVKVFRYSKKTAAGCF	+	+
43	SFVFKLASKVVPSVFSALTR	256	128	ASPTVFRSSVFLSLFVVAKK	+	+
35	LPVVFRVASKVFPALISKLT	256	128	KVFIATLVVSSFLAKPPRV	+	+

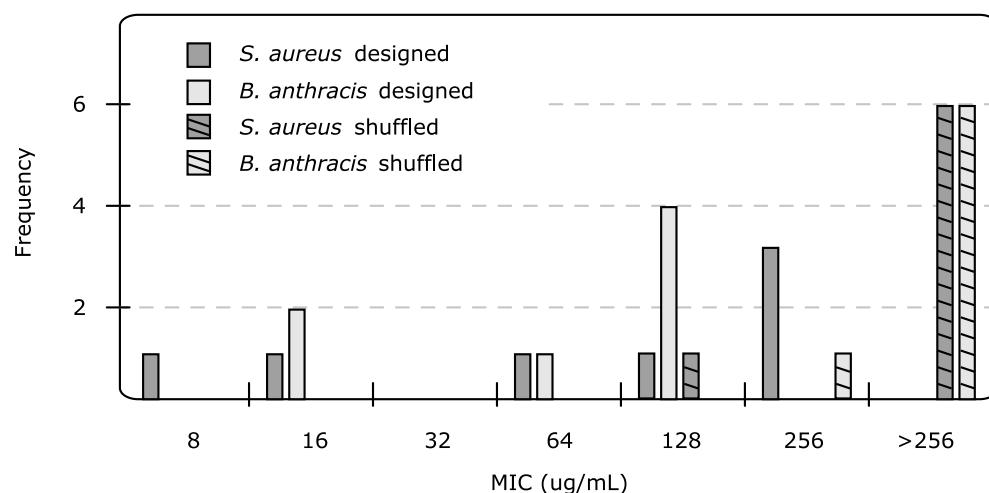


Figure 2-15: Activity of rationally designed AmPs against *S. aureus* and *B. anthracis*. The figure shows that shuffled peptides (the hashed bars) tend to be grouped on the right side of the plot, indicating that they have little or no antimicrobial activity. Only one of the shuffle peptides shows activity; however, it appears twice on the plot, once at $128 \mu\text{g}/\text{mL}$ against *S. aureus* and once at $256 \mu\text{g}/\text{mL}$ 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:

- M. Styczynski, K. Jensen, I. Rigoutsos, & G. Stephanopoulos. “An extension and novel solution to the (l,d) -motif challenge problem.” *Genome Inform Ser Workshop Genome Inform.* 2004;15(2):63–71; and
- K. Jensen, M. Styczynski, I. Rigoutsos, & G. Stephanopoulos. “A generic motif discovery algorithm for sequential data,” *Bioinformatics* 22:21-28 (2006).

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.

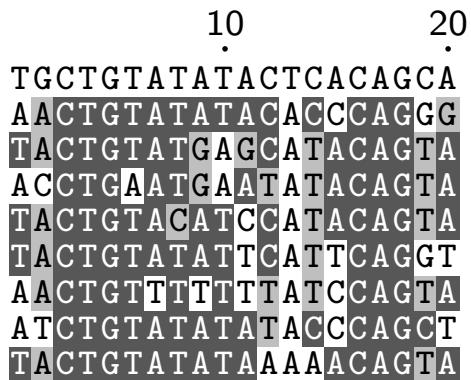
And this chapter, I described Gemoda: a motif discovery tool that has many of the strengths of Teiresias, but can find motifs that are best represented as PWMs. The details of Gemoda are discussed in the later sections of this chapter. But here, for motivation, consider the following output from the 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	TGCTGTATATACTCACAGCA
0	374	AACTGTATATACACCCAGGG
1	320	TACTGTATGAGCATAACAGTA
2	230	ACCTGAATGAATATAACAGTA
3	357	TACTGTACATCCATACAGTA
4	296	TACTGTATATTCAATTCAAGGT
5	306	AACTGTTTTTATCCAGTA
6	324	ATCTGTATATAACCCAGCT
7	361	TACTGTATATAAAAACAGTA

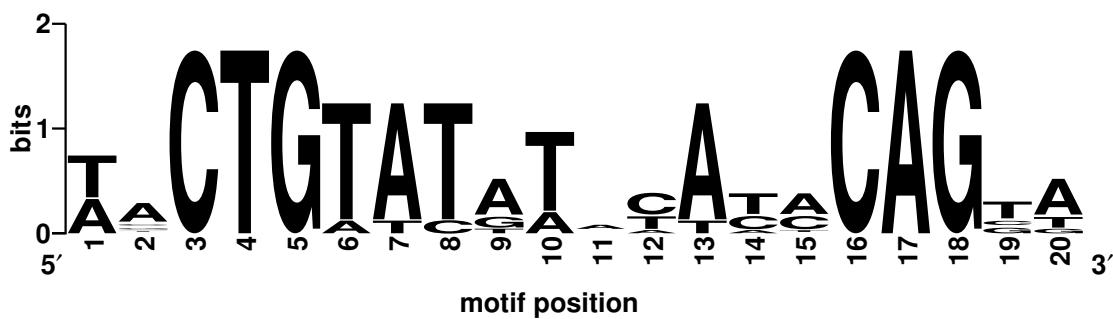
where, instead of one pattern per line, each line represents one of the offsets and the numbers on the left-hand side are, collectively, the offset list. Notice that here, only a handful of the positions within the pattern are fully conserved. However, most of the positions have “preferences.” For example, 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.

A)



B)



C)

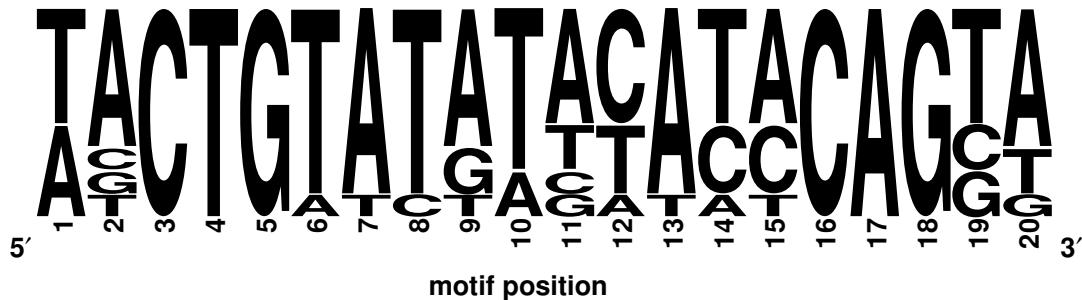


Figure 3-1: Alignment representing the LexA cis-regulatory binding site. Part A) of the figure shows the aligned sequences colored to indicate the degree of conservation. Part B) of the figure shows a sequence logo representing the information content of a PWM computed from the alignment of the motif 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 many 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 $S = \{s_1, s_2, \dots, s_n\}$, 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

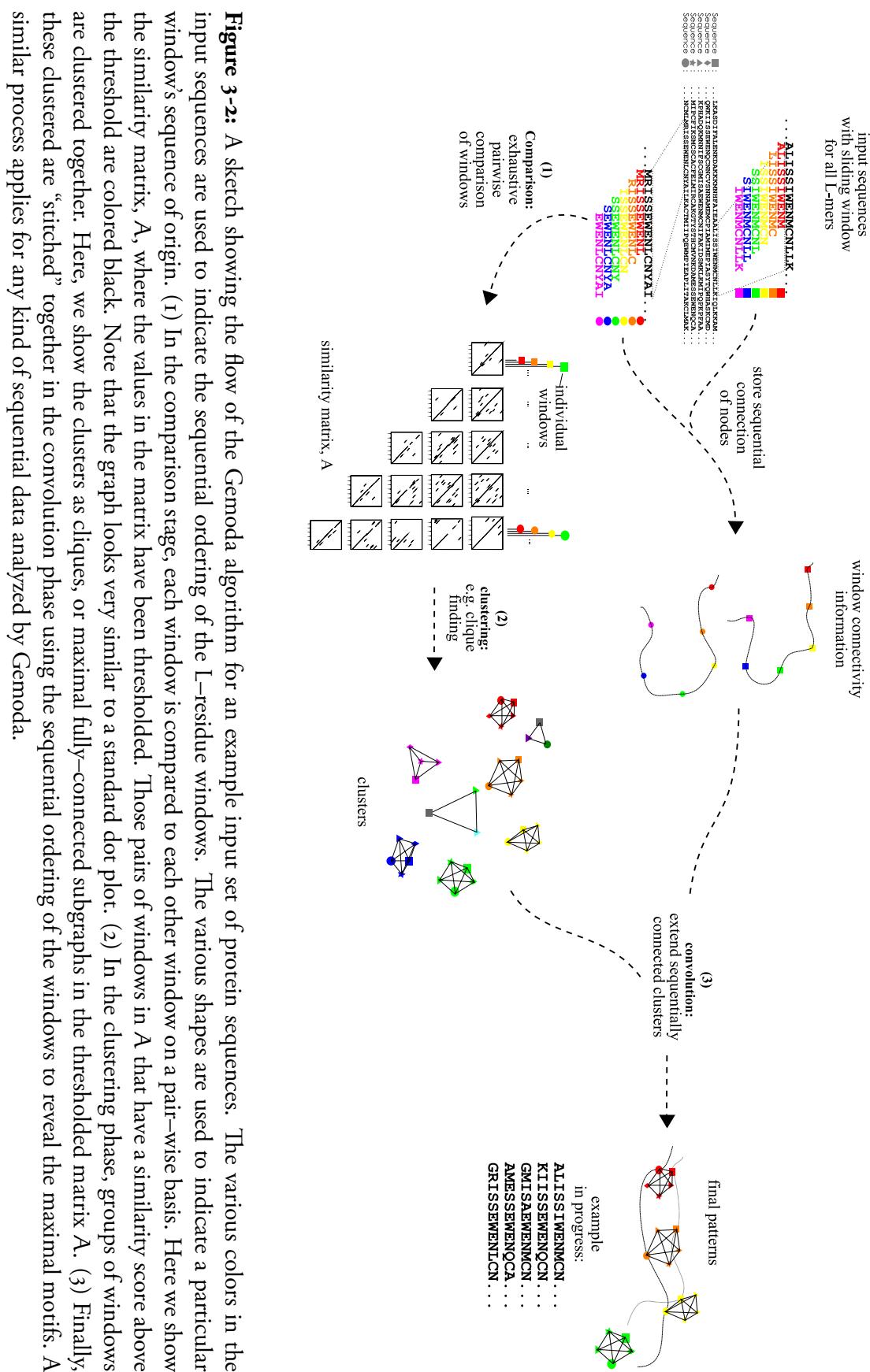


Figure 3-2: A sketch showing the flow of the Gemoda algorithm for an example input set of protein sequences. The various colors in the input sequences are used to indicate the sequential ordering of the L-residue windows. The various shapes are used to indicate a particular window's sequence of origin.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

amino acid: 1 if it is over the 50th percentile and 0 otherwise. The two peptides are AIKDWR and DIHV. Our two sequences are then

$$\begin{aligned} \text{seq-0} &= \begin{pmatrix} 0.61 & 2.22 & 1.15 & 0.46 & 2.65 & 0.60 \\ 0 & 1 & 1 & 0 & 1 & 1 \end{pmatrix} \\ \text{seq-1} &= \begin{pmatrix} 0.46 & 2.22 & 0.61 & 1.32 \\ 0 & 1 & 1 & 0 \end{pmatrix}, \end{aligned}$$

such that

$$\begin{aligned} s_{0,0,0} &= 0.61 \\ s_{1,0,0} &= 0.46 \\ s_{1,1,1} &= 1 \\ s_{0,3,1} &= 0 \\ s_{1,2,0} &= 0.61, \end{aligned}$$

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 10th window of size L in the 5th 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 $\mathcal{M}(s_{i,j:j+L-1}) \mapsto q \in [1, N]$. (For simplicity, we define $(s_{i,j} + 1)$ to be $s_{i,j+1}$, unless $s_{i,j+1}$ does not exist, in which case $(s_{i,j} + 1)$ is undefined.) Similarly, $\mathcal{M}^{-1}(q) \mapsto (s_{i,j:j+L-1})$ such that $i \in [1, n]$ and $j \in [1, W_i - L + 1]$.

We also define a similarity function $\mathcal{S}(s_{i,j:j+L-1}, s_{q,z:z+L-1})$, that takes as arguments two arbitrary windows and returns a real-valued number indicating the level of similarity between the two windows. In the most simple case, \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 $\mathcal{W}(p)$ 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 $\mathcal{S}(\mathcal{M}^{-1}(i), \mathcal{M}^{-1}(j)) = \mathcal{S}(s_{i,j:j+L-1}, s_{q,z:z+L-1})$.

A is then, quite simply, a similarity matrix for all N windows based on the similarity function \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 $\mathcal{F}(A, B)$ 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:

$$\begin{aligned}\mathcal{M}^{-1}(0) &= s_{0,0:4} \\ &= \text{AATTG} \\ \mathcal{M}^{-1}(1) &= s_{0,1:5} \\ &= \text{AATTG} \\ &\vdots \\ \mathcal{M}^{-1}(6) &= s_{1,2:6} \\ &= \text{TAGGA}.\end{aligned}$$

The members of the matrix A are computed as follows:

$$\begin{aligned}A(0,0) &= \mathcal{F}(\text{AATTG}, \text{AATTG}) = 0 \\ A(0,1) &= \mathcal{F}(\text{AATTG}, \text{ATTGG}) = 2 \\ &\vdots \\ A(2,5) &= \mathcal{F}(\text{TTGGC}, \text{ATAGG}) = 3 \\ &\vdots \\ A(6,6) &= \mathcal{F}(\text{TAGGA}, \text{TAGGA}) = 0.\end{aligned}$$

The matrix A is then

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

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 \mathcal{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 $\mathcal{C}(A) = c^L = \{c_1^L, c_2^L, \dots, c_Z^L\}$ where each c_i^L is a set of indices in A and $c_i^L[q]$ is the q^{th} member of c_i^L . Note that \mathcal{C} can be any function; common clustering functions include hierarchical clustering, k -nearest-neighbors clustering, and many others. We call each c_i^L an “elementary motif” of length L . We note that a clustering function may assign each node (window) to one or more groups. In the latter case, each c_i^L may have a non-null intersection with any c_j^L . That is, a single window may appear in an arbitrarily large number of clusters.

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

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

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

Our objective is to find all the maximal motifs in the sequence set using the elementary patterns. We do this by performing $c_i^k \curvearrowright c_j^k$ for all i and j at each length $k \geq L$ until c^k is empty ($|c^k| = 0$). We then define the set of maximal motifs comprising c^k for all k as P , the final set of motifs that are returned to the user. This simple induction scheme guarantees that all (and only) the maximal motifs are in P given appropriate clustering functions (see 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 cliques¹ 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_i^L is a clique in the g -thresholded A matrix, the Gemoda algorithm has a guarantee of compositional and length maximality, relative to the threshold g . 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 [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_i^L is a maximal disjoint subgraph in the g -thresholded A matrix (i.e., c^L represents the connected components of A), the computational complexity for the clustering phase is significantly less than for clique-based clustering. As well, in the case where Gemoda is applied to nucleotide and amino acid sequences, the motifs from this connected components method may be more intuitive than motifs found using clique-based clustering.

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

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>. 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, Φ , such that

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

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

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

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

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

In this equation, the combinatorial factor represents the number of different ways that windows can be sampled in groups of r , the cumulative product represents the necessary conditions for the formation of a clique of length L , and the last factor represents the likelihood of extending a clique of support r to be length q . In this way, the relative likelihood measure attempts to represent the expected number of motifs of length q and support r that would occur at random given the calculated similarity matrix. Notably, this significance is based solely on the similarity matrix A , and so it can be used for either categorical or real-valued sequence data clustered with the clique-finding method.

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 “ \square ” 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 ABCEFVG, that would be represented with the following Fasta-formatted file:

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

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 |cn| ≠ 0 do
        for i := 0 to |cn| step 1 do
            ismaximal := true
            for j := 0 to |cn| step 1 do
                f := cni ∩ cnj
                if |f| ≠ 0
                    if f ⊂ cn+1 = false
                        cn+1 := cn+1 ∪ f
                    else
                        choose maximal(f, cn+1)
                    fi
                    if |f| = |cni|
                        ismaximal := false
                    fi
                fi
            od
            if ismaximal = true
                P := P ∪ cni
            fi
        od
        n := n + 1
    od
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 extend 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
```

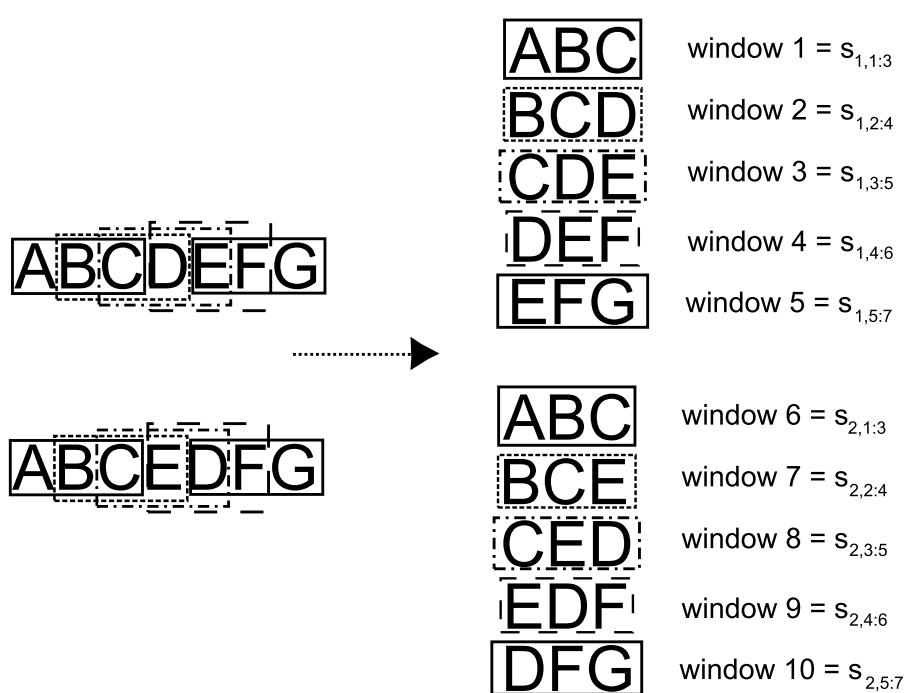
```
pattern 1:      len=5    sup=2    signif=5.000000e+00
  0    1      BCDEF
  1    2      CEDFG
```

```
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 earlier, a motif of width $\mathcal{W} \geq L$ 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.

a)



b)

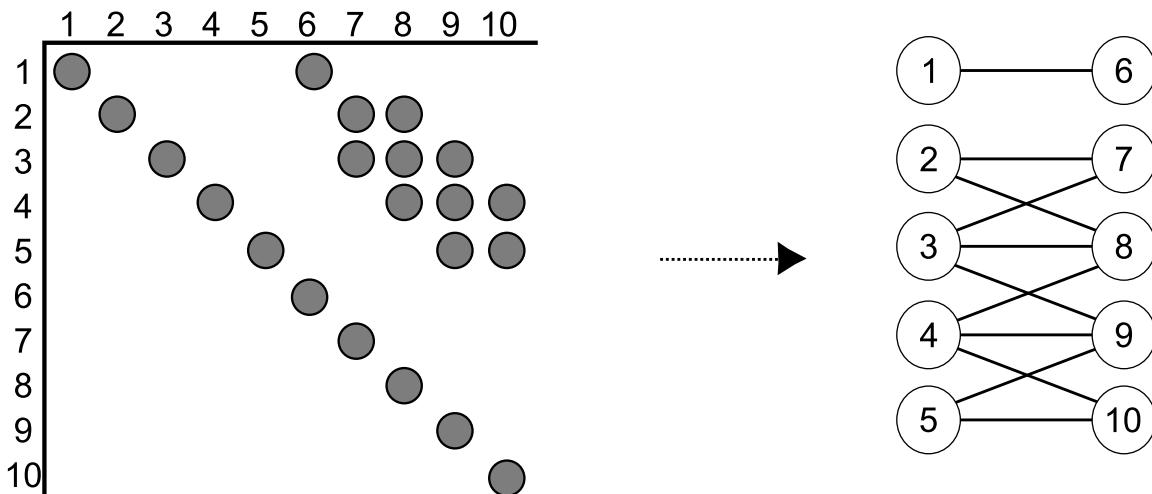
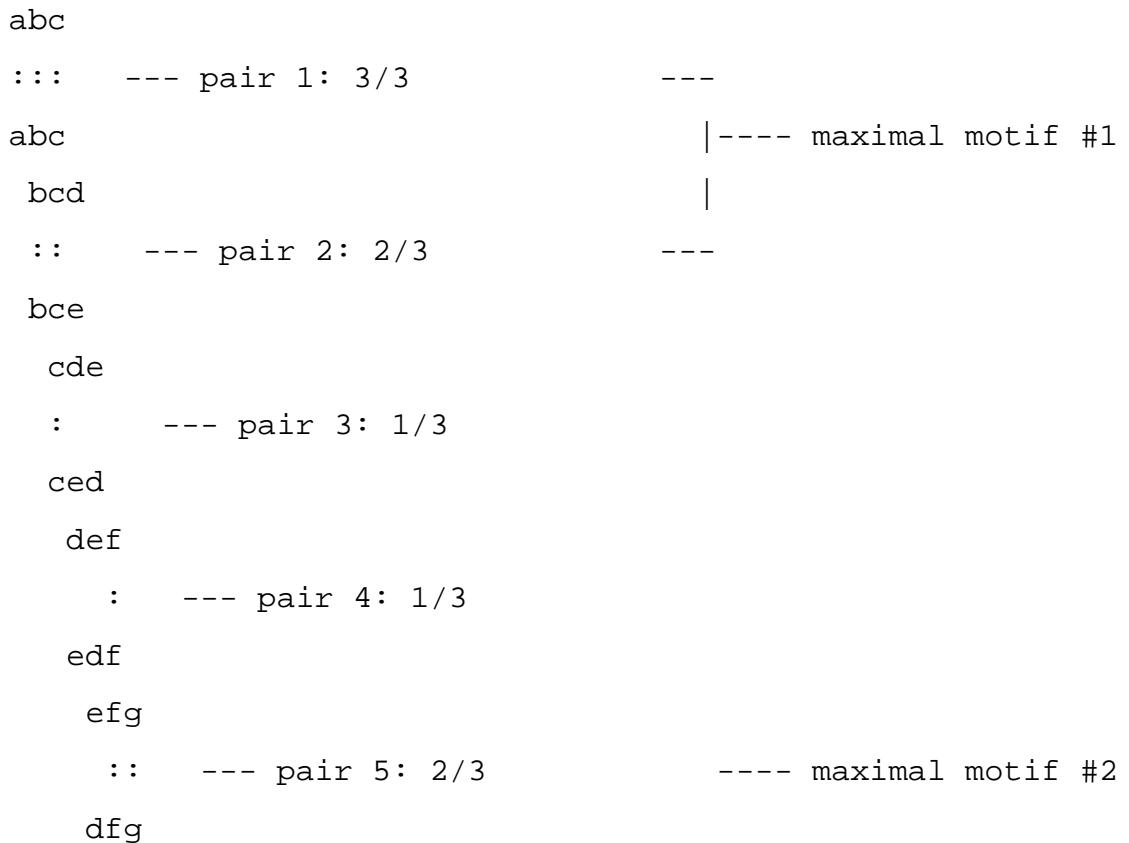


Figure 3-4: A natural language example illustrating the steps that Gemoda takes. In a), we see the three words, or sequences, being broken into overlapping windows of three letters each. Gemoda would then compare each of these windows to each other using either of the similarity metrics described in 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.

This is illustrated in the following diagram.



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` . and .`fg`, and `abc..fg` is not a maximal motif motif (with $L = 3, g = 2$).

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

MOTIF

MOTOR

POTION

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

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

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

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

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_o^L and

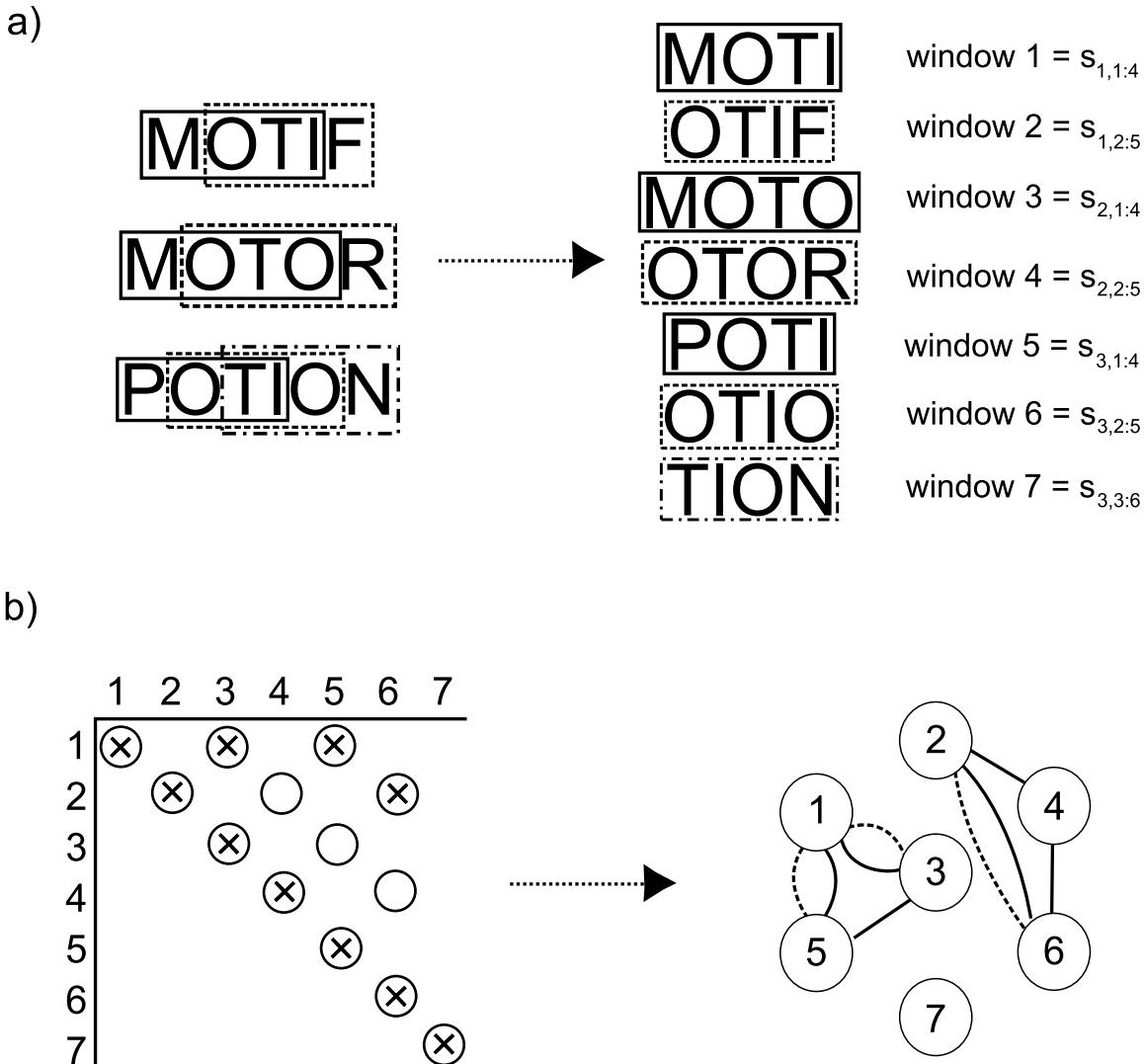


Figure 3-5: A second natural language example illustrating the steps that Gemoda takes. In a), we see the three words, or sequences, being broken into overlapping windows of four letters each. Gemoda would then compare each of these windows to each other using either of the similarity metrics described 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.

c_i^L , respectively.

Finally, we perform the convolution step. Using the last set of motifs described (with connected components clustering and the identity similarity function), we perform the convolution operation on each ordered pair of motifs; in this case, it means performing $c_o^L \curvearrowright c_i^L$, $c_i^L \curvearrowright c_o^L$, $c_i^L \curvearrowright c_i^L$, and $c_o^L \curvearrowright c_o^L$. For the first operation, we find the windows immediately after each of the windows in c_o^L , which is the set $\{2, 4, 6\}$. The intersection of this set with motif c_i^L is the convolved motif of length $L + 1$, which is $\{2, 6\}$; we can call this c_o^{L+1} . In performing $c_i^L \curvearrowright c_o^L$ and $c_i^L \curvearrowright c_i^L$, 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_o^{L+1} . Performing $c_o^{L+1} \curvearrowright c_o^{L+1}$ results in a null set, meaning that there are no more motifs. At this point, we terminate convolution. It is worth noting that c_o^L is returned as a maximal motif because window 4 cannot be extended, but c_i^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

$$\{\{\text{MOTIF, POTIO}\}, \{\text{MOTI, MOTO}\}\}.$$

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

$$\{\{\text{MOTIF, POTIO}\}, \{\text{MOTI, MOTO, POTI}\}\}.$$

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

$$\{\{\text{MOTIF, MOTOR, POTIO}\}\}.$$

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).

>sp|067012|SPOT_AQUAE - Aquifex aeolicus.
MSKLGEVSELEEDLEKLLSHYPQHAAEIIQRAYEFKEKHQKRTGPEYIIPHLNVALKLAELGMHDHETIAAALLHDTEDETDPTAEEIKERFGERVAKLVEGVTKIGKIKYKSEQAENYRKLIATATE
DPRVLLKLSDRLDNVKTWLWFRPEKKRKAIAKETMEYIAPLAHLRGWSKINBLEDWAFKYLYPEEYEVKRNFVKESRSKNEEYLRYKVPVKRKELEYGIBAEIKYRSKHYISIWEKTRKGIRLED
VHDILGVRILIVNTVPECYTVLGIHLSLFRPVGPKFDYISLPKPNLYQLSHTTIVADKGKLVEFQIRTWEMHERAEKGIAASHWAYKEGNPNSDAGVYWSLRELVESIQGSTNPSEVLNKSNLFFEEV
FVFTPKGDVLWPLKGSKPTVPLDYLTHVGNHCAAGNSKRNIPVNLNQYELSVFVTSRARNKIQFLKKQSERERYLSEGKRILERIWERKGLSHEDLINKIWERRVRDFDE
EELLALGKRKISSANLILPIKPKKEERGRSGTFLDLSNIKHEVACKCKIPDGELIGVITRKGVLHECSCSLNKLNVRLNPEKVEVOLQASGYQTDIRVVASDRIGLSDITVKVISES
GSNIVSSMTNTREGKAVMDFTVEVNKEHLEKIMKKIKSVEGVKICKRLYH

>sp|051216|SPOT_BORBU - Borrelia burgdorferi (Lyme disease spirochete).
MIQAYEIAHLIKINDLEKARNIFKKTVENTYKDEFERKSIFKALEIAEQLHYGQYRESGEPYIIPHMIVSFLAKFQLDKFATIAGLHDVLEDTNVEKEEIVKEFDEEILSLIDGVTKIHDHNKTRS
IKEANTISKMFAMTHDIRIILKADLHNMTTSLYPLKRNQRDIAKDCDLSTYPIAERLGISSLTELEDSLHKLYDKYEIKNIFSETKIREKKLYGKLSKIEBKQSGIAEITVRSKHFY
SIFRKHMOTRNLKTOIFDFTLGRICCKQKCEYILEVHVRWPKP1BGPDKYIASPKENKYLSLHTTWRIPEDNQOLIEQIORTEDMRANIYANGAWAHYKEQIEBLADDLSFINRKKWQOESANKS
QYSMNDIHKEELLNTFIYVYTPEGEVVELPGSNSIDFAYIHTDIDGQALYAKINGKISSITCPKLNQEIVEIFTSKDSKPVDIWLNSVRTKKARSKIRSWLNKNNTFIVDNNIAYLVGANKEQRKL
FSLFKSYTTCIKRIAIPECSPTTGEDIIGIHHKDEIIVHNENCQKLKSYKKPQLIEVEWEATPTRKVHIIILLKELKGIFSYLENIFTLNDVRLISEKIEDCGNGHIGITNIIVSSNAKNITKIISA
LKENPNILQIMQICEDDNYD

>sp|P17580|SPOT_ECOLI - Escherichia coli, Escherichia coli O157:H7, and Shigella flexneri.
MYLFESLNQLOIYLTPEDOIKLRQAYLVARDAHEGQTRSSGEPYIIPHVAACILAEMKLDYETLMAALLHDIEDTPATYQDMOLFCKGSVAELVEGVSKLDKLKFRDKKEAOAENFRKMINAMVQD
IRVILIKLADRTHNMRTLGSLRPDKRRRAIETLEIYSPLAHRLGIHHKTELEELGFPEALYPNRYRVIVEKVKARGNRKEMIQKILSEIEGRQLQEAGIPCRVSGREKHLYSIYCKMLKEQRFHSM
DIYAPRIVVNDSTCDYTRVLQGMSHLYKPRPGRVYDIAIPKANGYQSLSHTSMIGPHGVPEVQVQIRTEMDQDMAEOMQAMHAWYKEHGETSTAQIRAQWMSLQSLLELQSGASSFEIESVKSDLFDPDE
IVYVTPGRIVELPAGATPFDVFAVHTDIGHACVGRVDPQYPLSPLQDGTQVTEITAPGARPNAWNLVFFVUSSKARAKIRQLLNLKRDSDSLSGRLLNHALGSRSKRLNIPQENQRELDRM
LATLDDLLAEIGLGNAMSVVVAKNLQHGDSAIPPATQSHGLPLIKGADGVLITFAKCCRIPGDPDPIAHVSPGKGLVIHESCRNIRGYQKEPEKFMAVEWDKETAQEFIGEITKVMFHNQGALANLT
AINTTSNIQSLNTEEKDGRVYSAFIRLTARDRVRHLANIMKIRVMPDVKVTNRN

>sp|P43811|SPOT_HAEIN - Haemophilus influenzae.
MIARADEHGGFRSSGEPYIIPHTVAVASTIAQHLNLDEHAVEMAAHLHDVIEDTPYTEEQLEKEEFGASVAEIVDGVSKLDDKLKFRTRQEAQFNRKMLIAMTRDVRVLIKADLRTHNMRTLGSRLPDKRR
RIAKETELEYCPLAHLRLHEIHKNELEDSLSPQAMPHYRVEYLKLLVDPVARSNRQDLEIERSOEIKVRENSGIFARVWGRENHLKYIYQKMR1KDQEFHSIMDIYAFRIVKVNVDYCXRVLGQMHNLKY
PRPGRVKDYIAVPKANGYQSLQTSMIGPKGPVPEVHVIHTEDMEQVAEMGITAHWVYKENGNDSTTAQIRVQRWLQSLVEIQQSGVNSFIFIENVKSEFFPKIEVYVTPKGRIVELPMGATAVDPFAYAV
HSDVNGTCVGTVEHVKYPLSKLASQEGTQVNTIIDTPNPAHEFWAVLNFTVARTAKTRHLYKQRCEDDVALGVELEFWNLVQPHNLGDFSIQQTIRVTLDALALSLDELLERBIEGLGNQASMSIHQFVG
VPLESANTKLNFELESKILITAPIQMGKTOFACQCCPILGDPDVGCTEKNTVVVHQQCASLKNACROSLAKWDNVNQSAVNFEAELQIELNQNALLSMLTMAISASESSLNQIWELEENNLLVILQ
VCVKDIDKHLANIVHRLKIGITGVVNVKRNINEL

>sp|P47520|SPOT_MYCGE Probable - Mycoplasma genitalium.
MATIQEIECDFPLAKIAQKFTNAEIELINKAFYHAKTWHENQKRLSGEPFFIPLRITALSLSVWNMDPITICAGLLHDIEDTQTEANIAMIFSKEIAELVTKVTKITNESKKQRHLKNNKEENLNLSKF
VNIAINSQNEINVMVLKADLRDNIASTEFLP1ECKVIAKETLELYAKIAGRIGMVPVTKLADLSFKLWDLKNYQDNTLSKINQKQVFYDNEWNDFKQQLAQNQIYEQLESRKGTYIYQK
VHEONQSKIDHLFAIRLPLTKESELDSYCHILHLNFLDSKYKDFDIAQSKQNLQYOSIHTTVRLKGLNVEIOIRTOQMDNVSQPKGLASHWYKEQEGGLLAPALOLNVLTVKQHSHDFLKRIFGTDII
KINVSASHEPVNIQINVDNNKLLDIAFENYPKQFAKLTKEIDGVEINSFDTSENEMLIEFYFGKNNNLKSKWIRYMNPNIPYREVVKSLAKLAKSGRSYSELAFYKEGEKQLKASETEIQKRL
NTLRLKXMSDYLALIECTNTDEHLLFLAKNNNDKWLTKPLFKASFVKNFHSYFEQIEGIFITKIVIEPCCSKIPDPMPEQVTGILTBNILSVHRYGCKNLQNKQKLIIPILYWNQIQLKLKPRKFR
SYININGVWSEKTINKICQTIINGDGYCIKIPKINKQKDEFDNLNTLFVNYYQQLTLMQDITTKNISFSWKYL

>sp|P75386|SPOT_MCPN Probable - Mycoplasma pneumoniae.
MFYNWKLKLYKFSKMATEVIERDFLQKTAQKFPAFPEVALIDKALDYSKWHGQKRLSGEPFFIPLRITALRLEVWNMDNSNTVCAGLLHDIEDTQTEADLTAIFGKEITDLVVKVTKITSESKKOR
LNRKKEDLNLSLNVNIAMSSQSQEVNALVLLKADRLDNISSEIEFLAVEKQKIIAKETLELYAKIAGRIGMVPVTKLQDLSFKVLDPKNFNNTLSKINQKQVFYDNEWGNFKQKLEEMLEQNQIYEYRLES
RIKGJYSTYQKHFQHNIQAKLDFALRIVLKSESCDYHLLGLHNLNTFVPLMKHFKDYIAPKNSQFVYNSIHTTVRLKGLNVEQIRTRQMDRVSFKYGFASHIWYKEKEGGLLASALQVNYLNLSKQMH
DDFFKFRGFTGDDIILKVNNSDNEPVNPKVNLNEVNSNSKLDIAYLEIYLPQKFNKLEKILDVGVEVMSDFDVTAEENMIEFVGCKTNKLKWRRLYRMNNHVFRRERVKLDNLKLLKAVKAYSPELYKEALEH
LKLADETQIKQRNLNAGIQLKLTFLIELEYPHFPKNEHYFLASNNQKWRRELIPKFKFALSQAVFQNSYFEQIEGIFYITKIVIETCCTKIPDPMPEQVTGILTBNILSVHRYGCKNLQNKQKLIIPILY
NAHQLKMRPKFRCQINIRGVWSETTVNKIVQTIIEGDSYLERIIPKIDKQKDEFDNLNTMFIDNYHQHLITIMEQITTKNISYVWKYL

>sp|034098|SPOT_SPICI - Spiroplasma citri.
MDRDKIYEEVLAQIQLYKIDEATLKEIQQAYEAEKHHQGRVNRSGARYIIPHLWTTFLAQWQRMGPKTLIAQGLLHDVLEDTPTAEEQLEFQGIEIANLVEGVTKVSYFAKENRTQIKAQYLRKLYLS
MAKDIRVIVKLADRLHNLKTLGKPERQIJAESLEIYSAIAHRLGMKAVKQBEIEDISFKIINPVQYKNIVSLLNESSNKERENTINQKIEBLKKLITEKCKMSVKYGRSKSIYISYRKMNQFGK
FDDIHDILAVRIITNSVDDCYKVLFVGVHQYHTPLNRRFKDYIATPKHNLQYQLSHTTIVADDGLIFEVQIRTEEMDELAEQGVAAHWRYKEGENYDIAKQKQDIDERLDFKRLDLENISVQERDEIQQ
EVYKPDHMLBQIYQNDIFSSLYVLTPTGNGVKTFLPGSTLDFAYKHKISEGKETIGKPLFSPITVLSGDKWDVIIATQPKPNHSLWVSKTSALEKIKYKLLKELVTSDAKSVNLEKIKQ
TKSQTIEYIAKDKLWLKWLNSSETDRLERLHAINFNNDIEFLDWDVANDEYLTLEEINLVLYDHTETSQNEKILKLLQDKYQKQAKLQDIIIVQGINSKVISQCLPPIYEDITGYVSKAEGIKVHLKTCR
NIQSGDQDRQVEVSWEAVCKNQYDCAIRIEAIDRALLVDVTKVLSHLNASVQMSANVSGDLMNLTIKTIIVKVSNADRLQQIRSSLTIPDIKUVVERVM

>sp|P74007|SPOT_SYN3 Synechocystis sp. (strain PCC 6803).
MNAVAALPPTIHTTCQADHIIDELPQWLQWREIEQGQDETTAPHCLICRAFCFAYDLHQQRKRSGEPYIAHPVAVAGLLRDLGQDEAMIAAGFLHDVVEDTDISIEQIEALFGEETASLVE
GVTKLSKSFNSTTQEAFNRRPLAMKADIRVIVLKLADRLHNMTLDBLSPKQRAIETKDIAPLANRGLIWRFKWELEDLSFKYLEDPSYRKIQLSVLWVKGDRDLSRLETVKDMRLFRRLDE
FDDIHDILAVRIITNSVDDCYKVLFVGVHQYHTPLNRRFKDYIATPKHNLQYQLSHTTIVADDGLIFEVQIRTEEMDELAEQGVAAHWRYKEGENYDIAKQKQDIDERLDFKRLDLENISVQERDEIQQ
EVYKPDHMLBQIYQNDIFSSLYVLTPTGNGVKTFLPGSTLDFAYKHKISEGKETIGKPLFSPITVLSGDKWDVIIATQPKPNHSLWVSKTSALEKIKYKLLKELVTSDAKSVNLEKIKQ
TKSQTIEYIAKDKLWLKWLNSSETDRLERLHAINFNNDIEFLDWDVANDEYLTLEEINLVLYDHTETSQNEKILKLLQDKYQKQAKLQDIIIVQGINSKVISQCLPPIYEDITGYVSKAEGIKVHLKTCR
NIQSGDQDRQVEVSWEAVCKNQYDCAIRIEAIDRALLVDVTKVLSHLNASVQMSANVSGDLMNLTIKTIIVKVSNADRLQQIRSSLTIPDIKUVVERVM

>sp|P74007|SPOT_SYN3 Synechocystis sp. (strain PCC 6803).
MNAVAALPPTIHTTCQADHIIDELPQWLQWREIEQGQDETTAPHCLICRAFCFAYDLHQQRKRSGEPYIAHPVAVAGLLRDLGQDEAMIAAGFLHDVVEDTDISIEQIEALFGEETASLVE
GVTKLSKSFNSTTQEAFNRRPLAMKADIRVIVLKLADRLHNMTLDBLSPKQRAIETKDIAPLANRGLIWRFKWELEDLSFKYLEDPSYRKIQLSVLWVKGDRDLSRLETVKDMRLFRRLDE
FDDIHDILAVRIITNSVDDCYKVLFVGVHQYHTPLNRRFKDYIATPKHNLQYQLSHTTIVADDGLIFEVQIRTEEMDELAEQGVAAHWRYKEGENYDIAKQKQDIDERLDFKRLDLENISVQERDEIQQ
EVYKPDHMLBQIYQNDIFSSLYVLTPTGNGVKTFLPGSTLDFAYKHKISEGKETIGKPLFSPITVLSGDKWDVIIATQPKPNHSLWVSKTSALEKIKYKLLKELVTSDAKSVNLEKIKQ
TKSQTIEYIAKDKLWLKWLNSSETDRLERLHAINFNNDIEFLDWDVANDEYLTLEEINLVLYDHTETSQNEKILKLLQDKYQKQAKLQDIIIVQGINSKVISQCLPPIYEDITGYVSKAEGIKVHLKTCR
NIQSGDQDRQVEVSWEAVCKNQYDCAIRIEAIDRALLVDVTKVLSHLNASVQMSANVSGDLMNLTIKTIIVKVSNADRLQQIRSSLTIPDIKUVVERVM

Figure 3-6: Guanosine-3',5'-bis(diphosphate) 3'-pyrophosphohydrolase ((ppGpp)ase) (Penta-phosphate guanosine-3'-pyrophosphohydrolase) sequences. 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].

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-

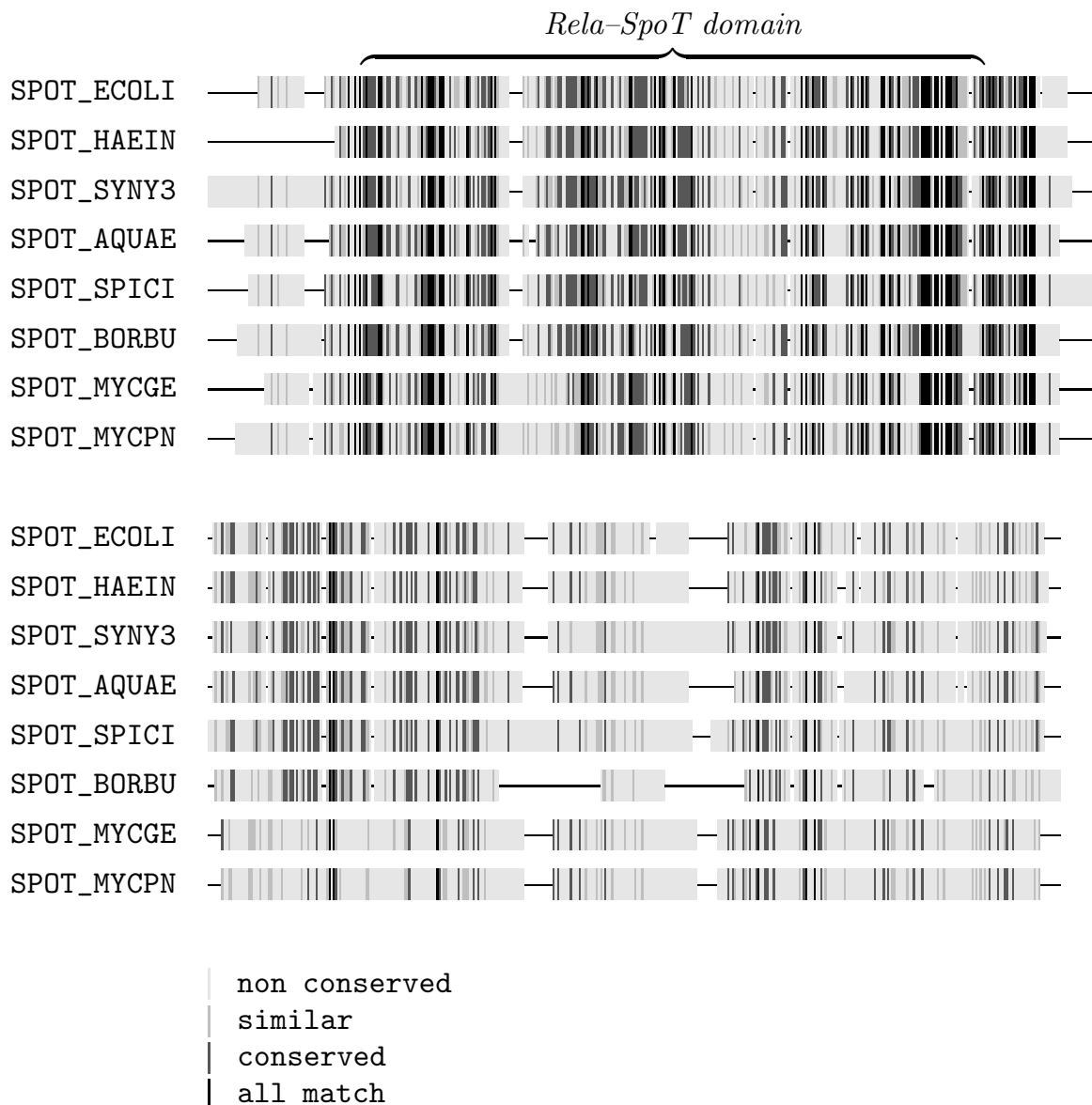


Figure 3-7: The RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences.

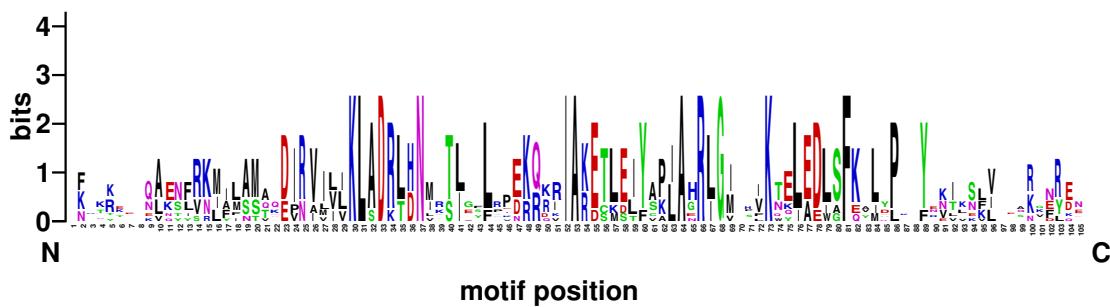


Figure 3-8: Logo representation of the RelA_SpoT motif detected in the 3.1.7.2 enzyme sequences. In this figure, the horizontal axis represents the position in the motif shown in Figure 3-7 on the preceding page, in the vertical axis represents the information content at each position.

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 $g \text{ \AA}$ 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 \AA or less (based on super-

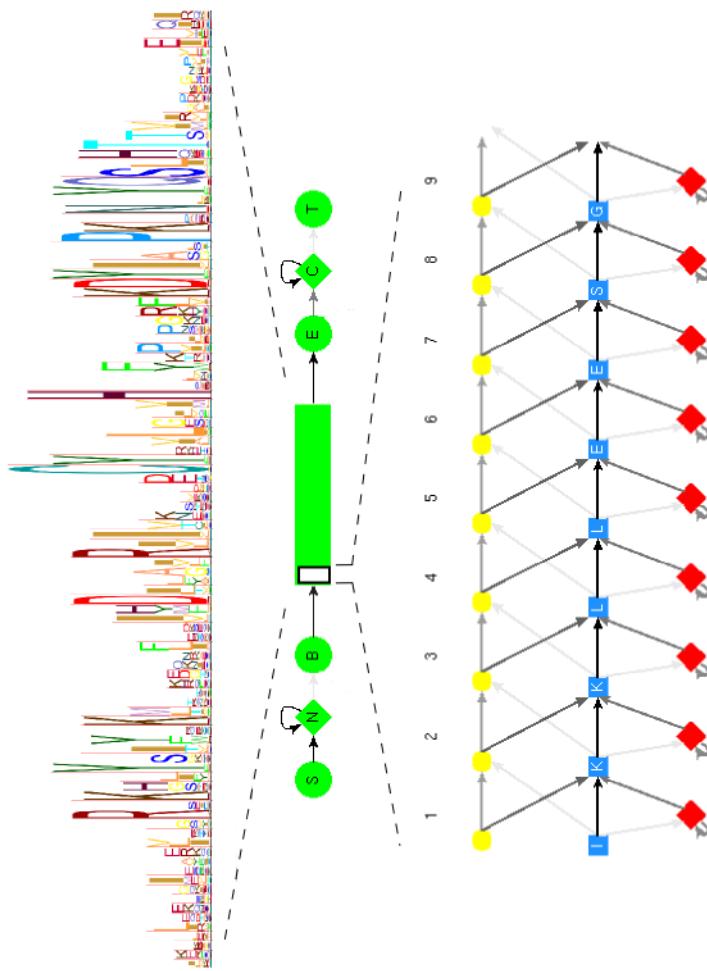


Figure 3-9: Hidden Markov model representation of the *RelA_SpoT* motif detected in the 3.1.7.2 enzyme sequences. In this figure, the boxes represent the different possible Markovian states at the first few positions in in the motif shown in Figure 3-7 on page 139 [220].

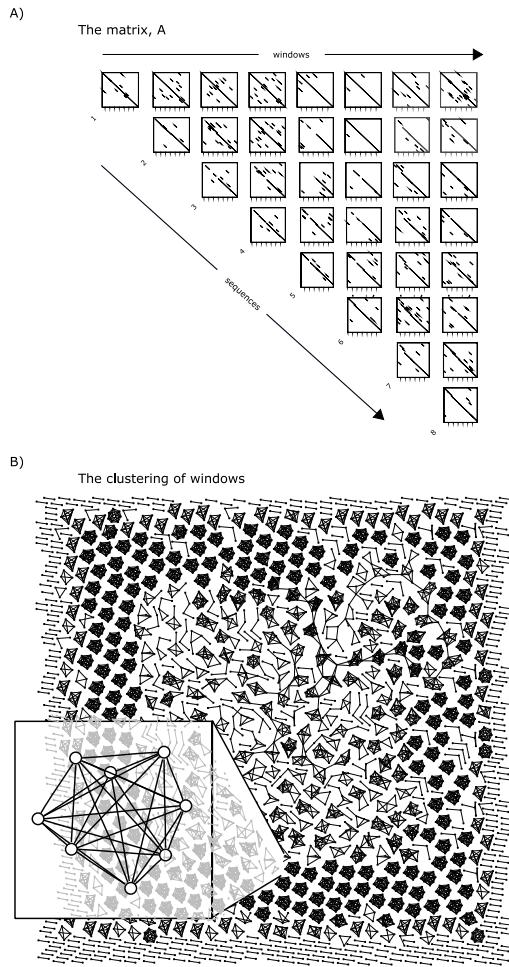


Figure 3-10: The similarity graph for the 3.1.7.2 enzyme example. (A) is the similarity matrix A , which contains one row and column for each window of 50 residues in the set of input sequences. Entries in the matrix have been thresholded such that pairs of windows that can be aligned with a bit-score 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.

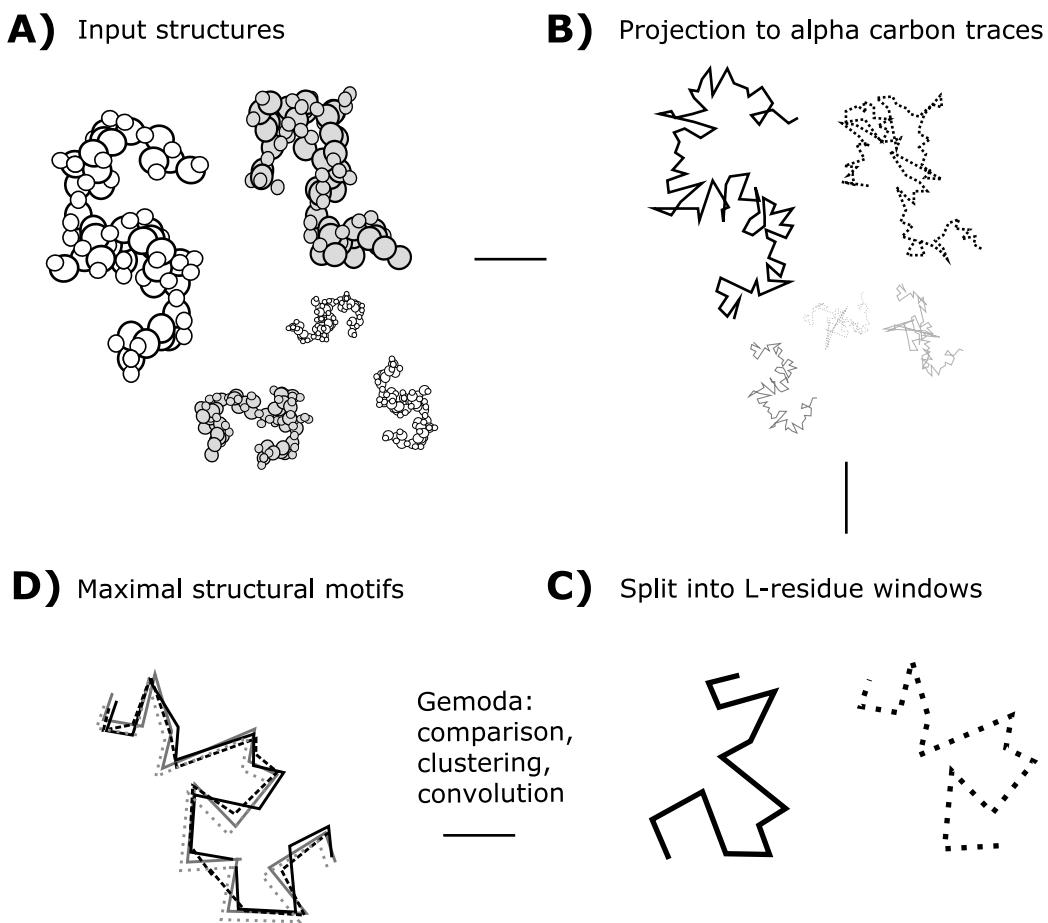


Figure 3-11: Alpha carbon trace projection used by Gemoda

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 β FIT protein and two, in different chains, in the α HXQ 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

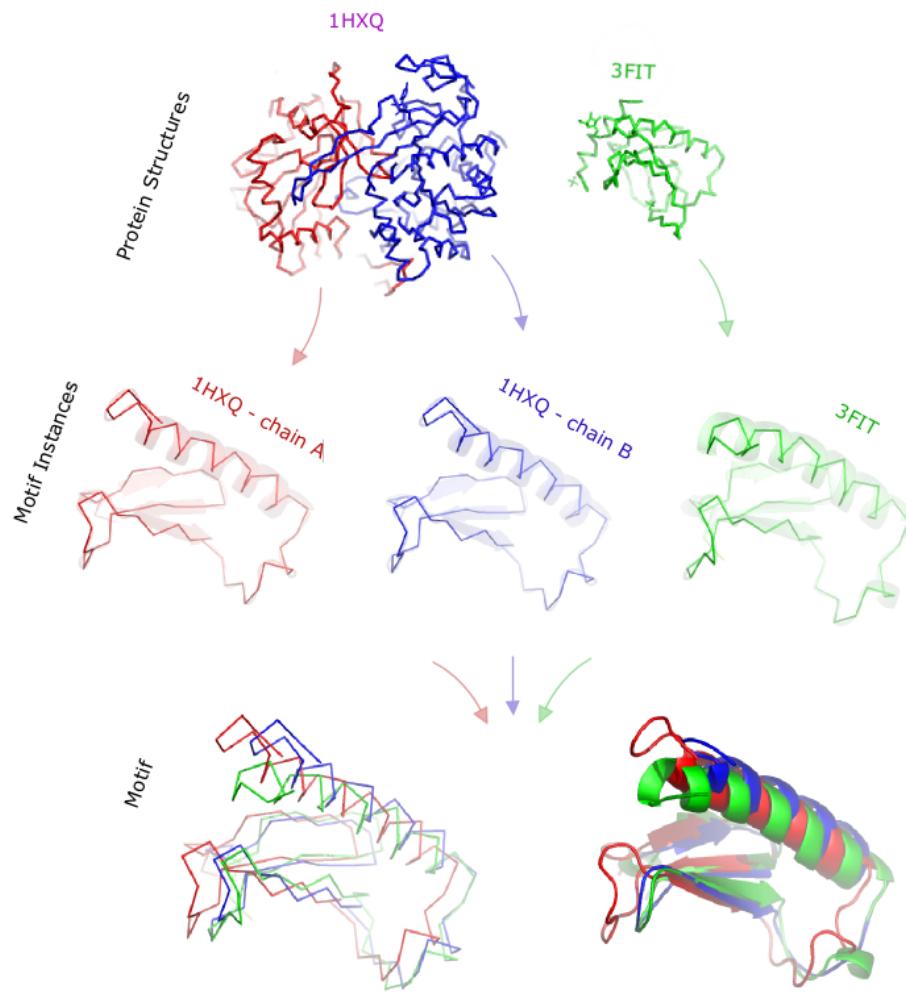


Figure 3-12: A motif showing structural conservation between the human galactose-1-phosphate uridylyltransferase and fragile histidine triad proteins originally reported by Holm and Sander [121]. The motif, as shown here, was “discovered” using the Gomoda algorithm along with three other, smaller, structural 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.

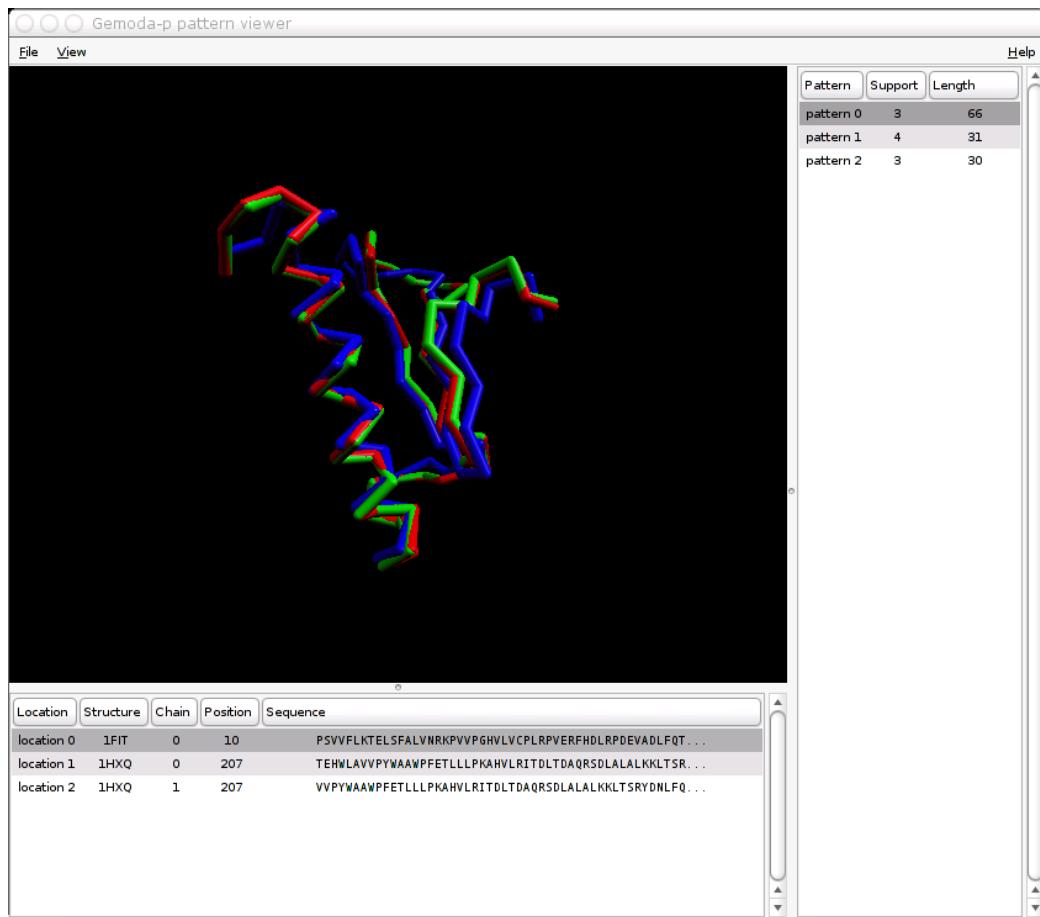


Figure 3-13: The human galactose-1-phosphate uridylyltransferase and fragile histidine triad structural motif (see Figure 3-12 on the preceding page) in Gemoda's 3-D structure viewer, which was written by the authors for viewing output from gemoda-p.

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 [195] presented two algorithms that looked towards solving this problem; Buhler and Tompa [45] 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 [195] 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 *a priori*. Instead, it is more likely that she would know the evolutionary distance between motif instances, i.e. the rate of mutation d/l . It is also unrealistic to mutate the embedded motif *exactly* d times; rather, the researcher is more likely to be interested in motifs that are d or fewer mutations away from each other. That is, in a real-world scenario, we would more likely have a reasonable estimate of the upper limit d/l of the mutation distance between embedded motifs. There may also be multiple, different motifs in the dataset. Finally, as experimental data are commonly rife with noise, it is likely that some of the sequences may be false-positive candidates for the motif; that is, some sequences may contain no motifs at all.

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

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

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

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 \mathcal{H} , where \mathcal{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 0.9 or 0.95. 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

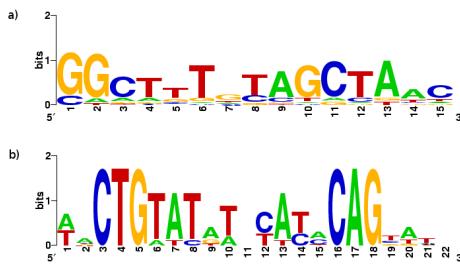


Figure 3-14: The sequence logo for a) the motif implanted in each sequence for the (l,d) -motif problem and b) the LexA binding site motif generated from the highest-scoring motif returned by Gemoda.

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].

Futhermore, 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.

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

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] . . . 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-formatted 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.

program	# processors	execution time (s)
biogrep	1	8683
biogrep	2	4477
biogrep	4	2266
biogrep	6	1620
perl	1	11780
perl	6	1916
patscan	1	28466

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 [109] and PAM series [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 [110, 112] and Vogt et al. [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 [10] and FastA [194] and multiple sequence alignment tools such as Clustal-W [245] and t-coffee [183].

The BLOSUM series of matrices was constructed in 1992 from Blocks 5 [109]: a database of protein blocks, or highly conserved protein regions, derived from families in the PROSITE database [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 [111, 197] and other functions [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 [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 BLOSUM62 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.

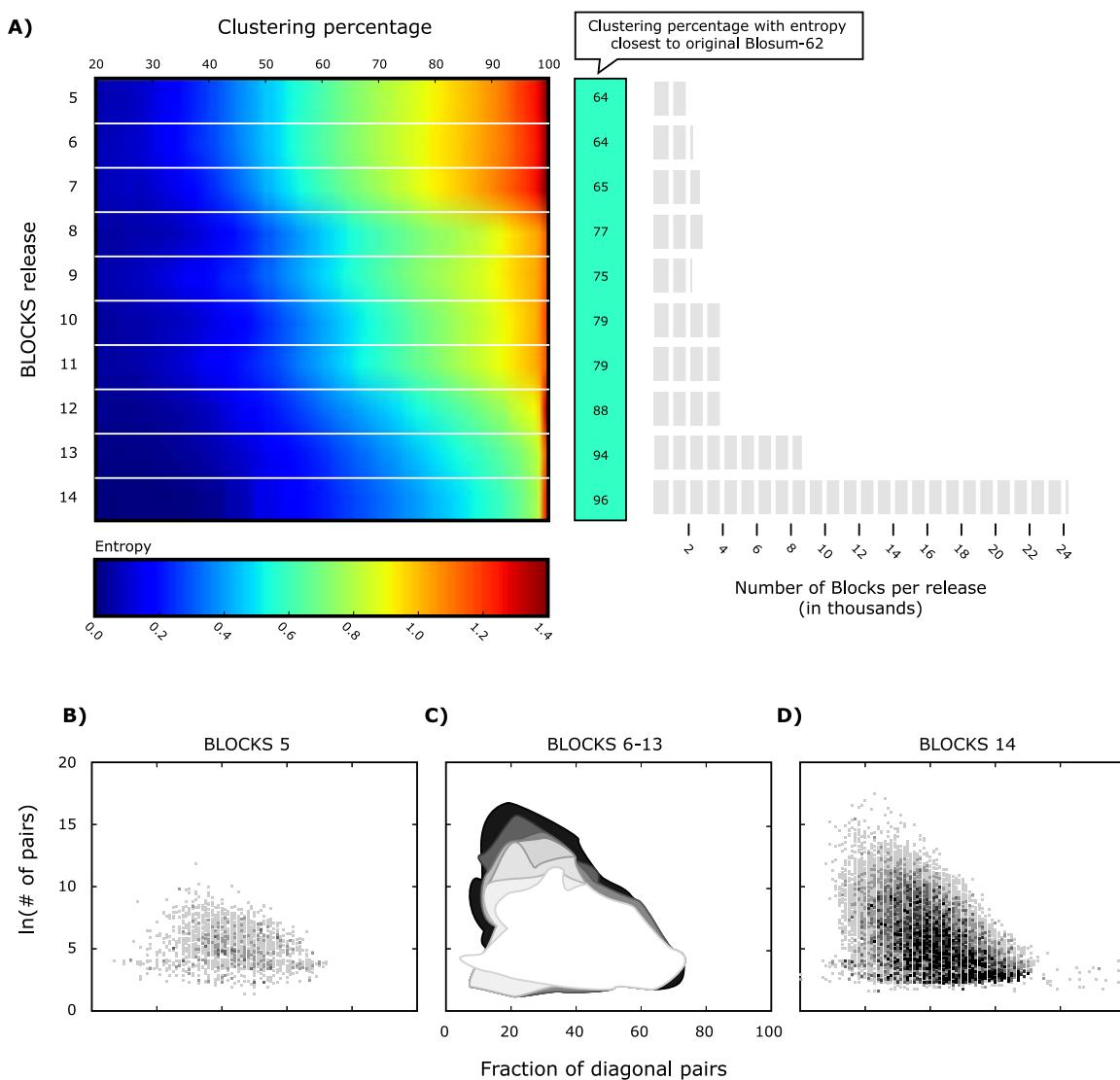


Figure 4-1: Characteristics of the BLOSUM matrices calculated from successive releases of the Blocks database. Panel A) shows the entropy of the scoring matrices computed from various Blocks releases as a function of the clustering percentage used by the BLOSUM algorithm (see methods). Blue colors indicate 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 BLOSUM62 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 ssearch 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 [10] 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 [110], as initially described by Pearson [193]: 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 BLOSUM62 matrix. We used this set of isentropic matrices for our sequence alignment tests. A given matrix's relative entropy reflects the required minimum length of homology in order for it to be distinguished from noise [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.

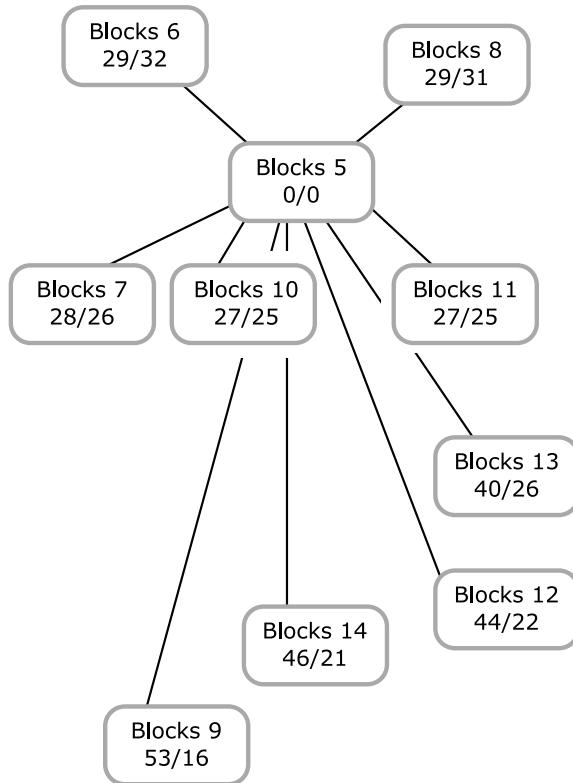


Figure 4-2: The relative performance of updated BLOSUM matrices. This figure is designed to emulate Figure 4 from Henikoff and Henikoff [109]. All matrix performances are compared to the revised BLOSUM62 isentropic analogue derived from Blocks 5, RBLOSUM64. Vertical distance from Blocks 5 indicates relative 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.

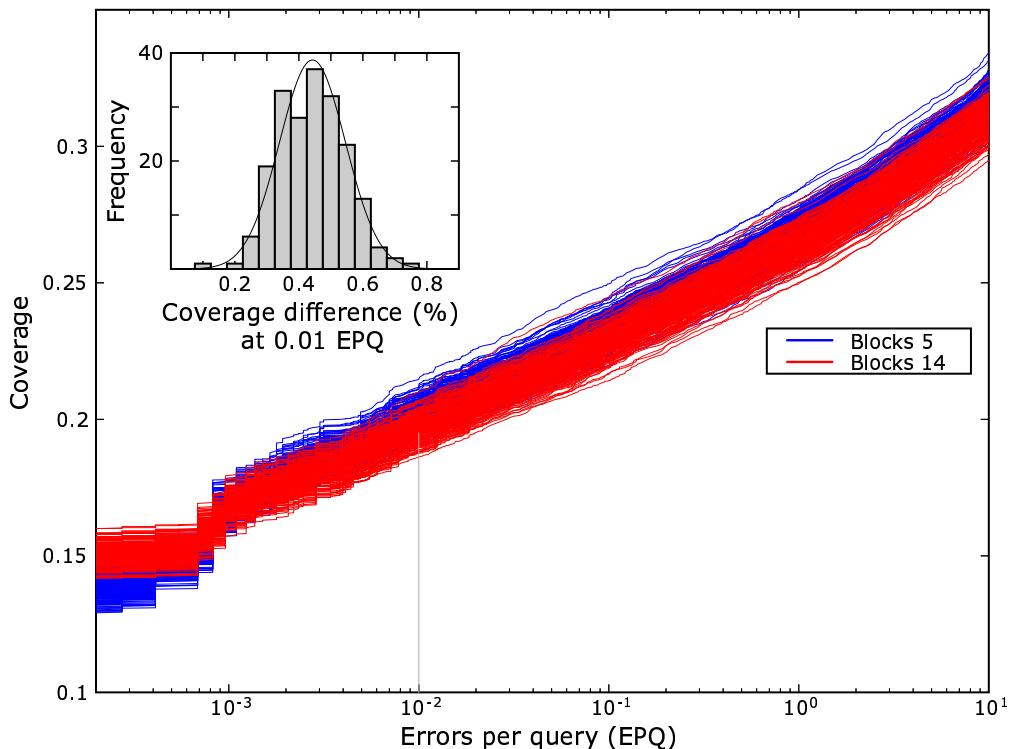


Figure 4-3: A complete set of Bayesian bootstrap replicates, with inset histogram of coverage difference. These data were created using the PSCE software [200]. (See Price et al. [200] for a thorough explanation of Bayesian bootstrapping). Each thin, faintly colored line represents one Bayesian bootstrap 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.

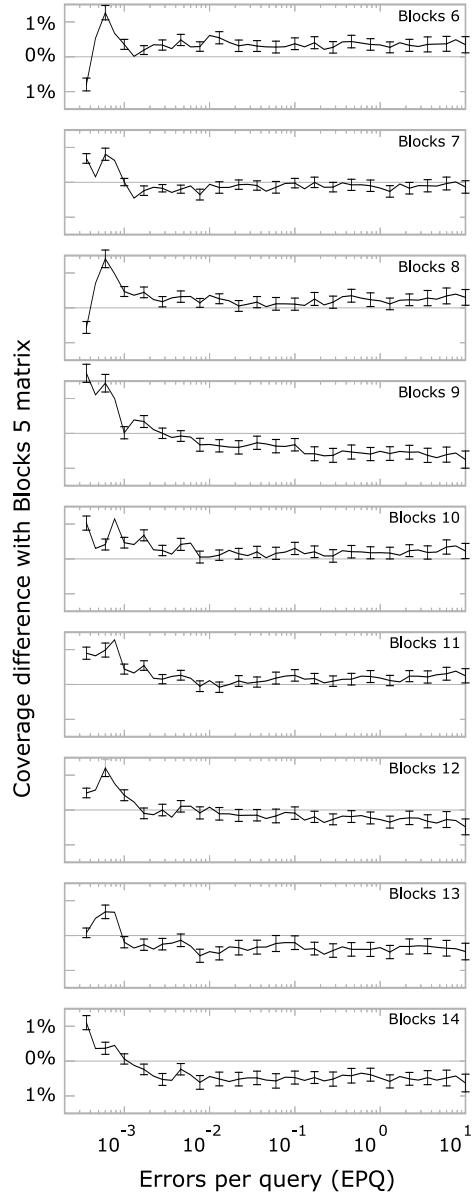


Figure 4-4: Plots of the differences in performance of updated RBLOSUM matrices. Each matrix is compared to the RBLOSUM64 matrix in 200 Bayesian bootstrap replicates to find the mean difference in coverage, and the confidence interval for that coverage, at a specific EPQ rate. These differences are plotted 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

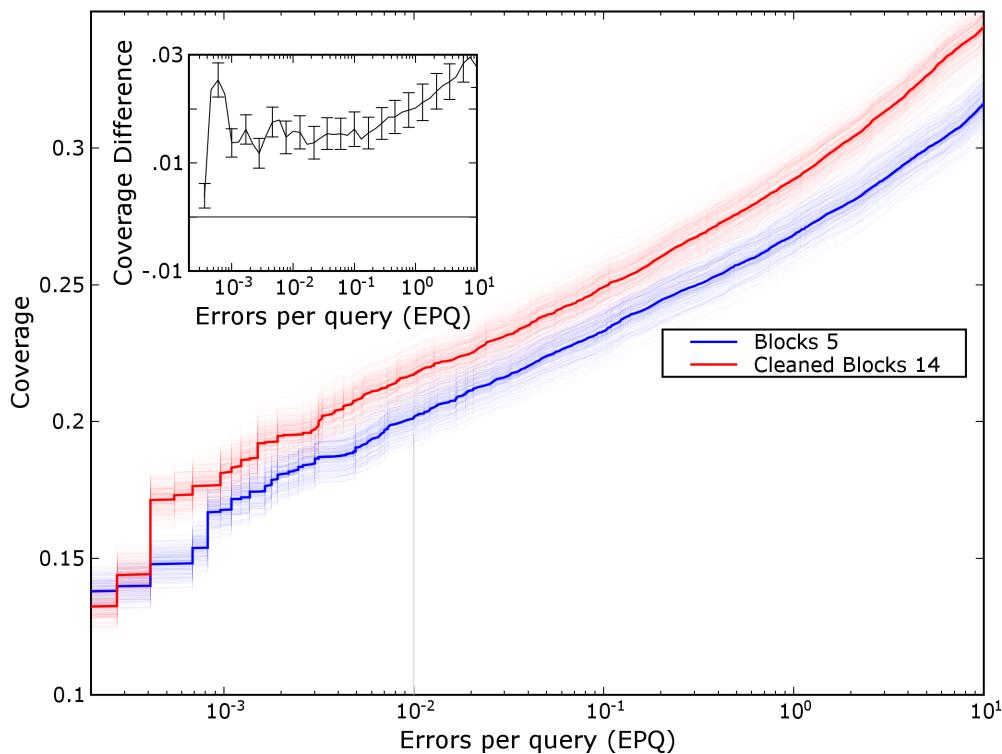


Figure 4-5: Coverage of a cleaned RBLOSUM matrix compared to the RBLOSUM64 matrix. Again, thin, faint lines represent individual bootstrap runs, while the dark line represents the parent dataset. These two distributions are quite distinct, with the cleaned RBLOSUM matrix being significantly more effective 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 Plamondon

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 *o* and *g*, written by 44 writers with each writer submitting 250 digits (8 samples were discarded by the original authors).

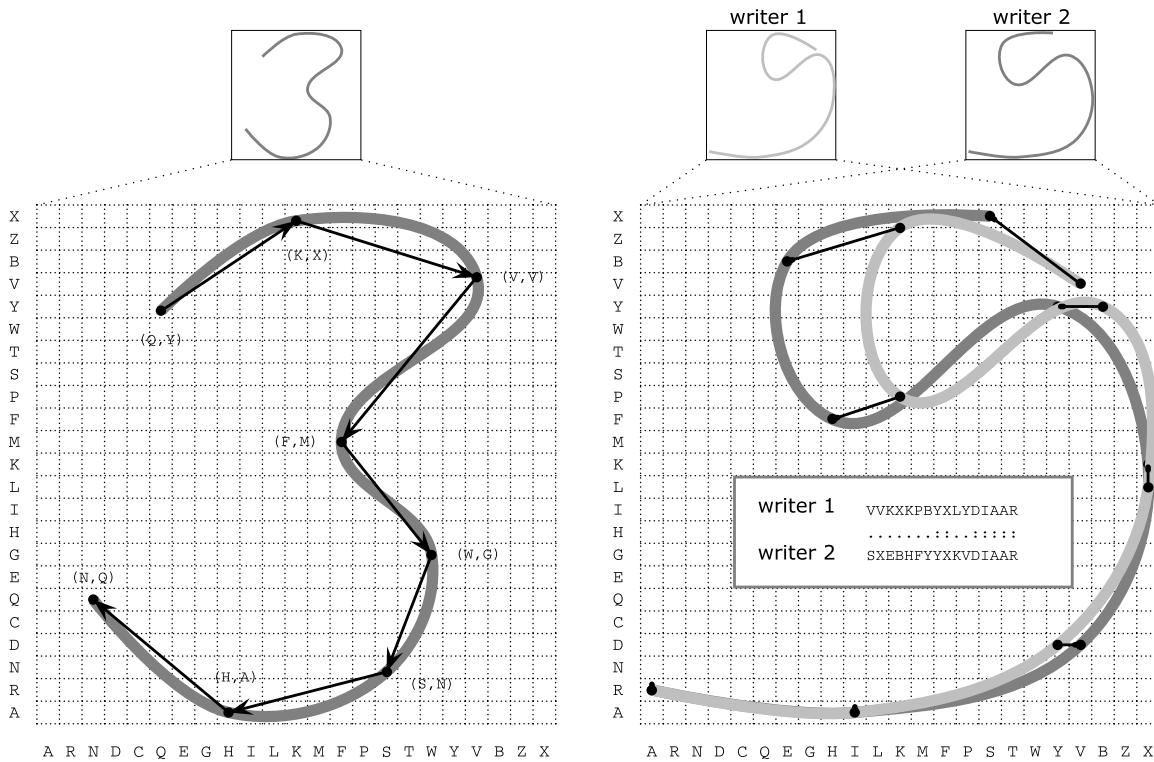


Figure 4-6: Projection of a digit written with a PDA stylus into protein space. Concatenating the set of points gives a protein sequence representative of the digit. In this case, the sequence is QYKXVVFMWGSNHANQ. An alignment of nines from two different writers. The boxes at the top show the input from 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.

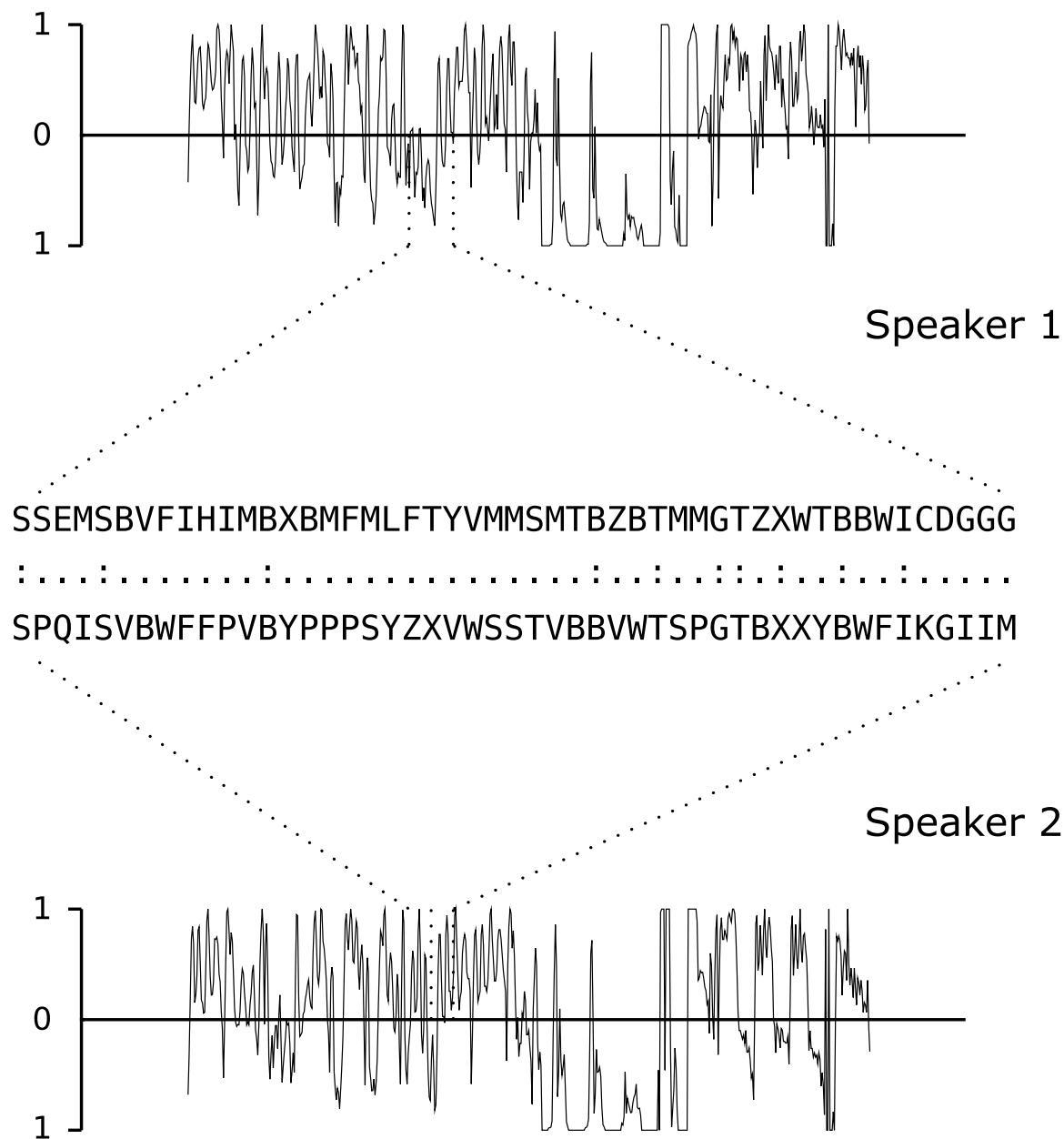


Figure 4-7: An alignment of the spoken-letter “X” recorded from two different speakers. The plots at the top and bottom are recordings for first and second speakers, respectively. The breakout in the center shows a section of the protein projection of each recording and the alignment generated using 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.

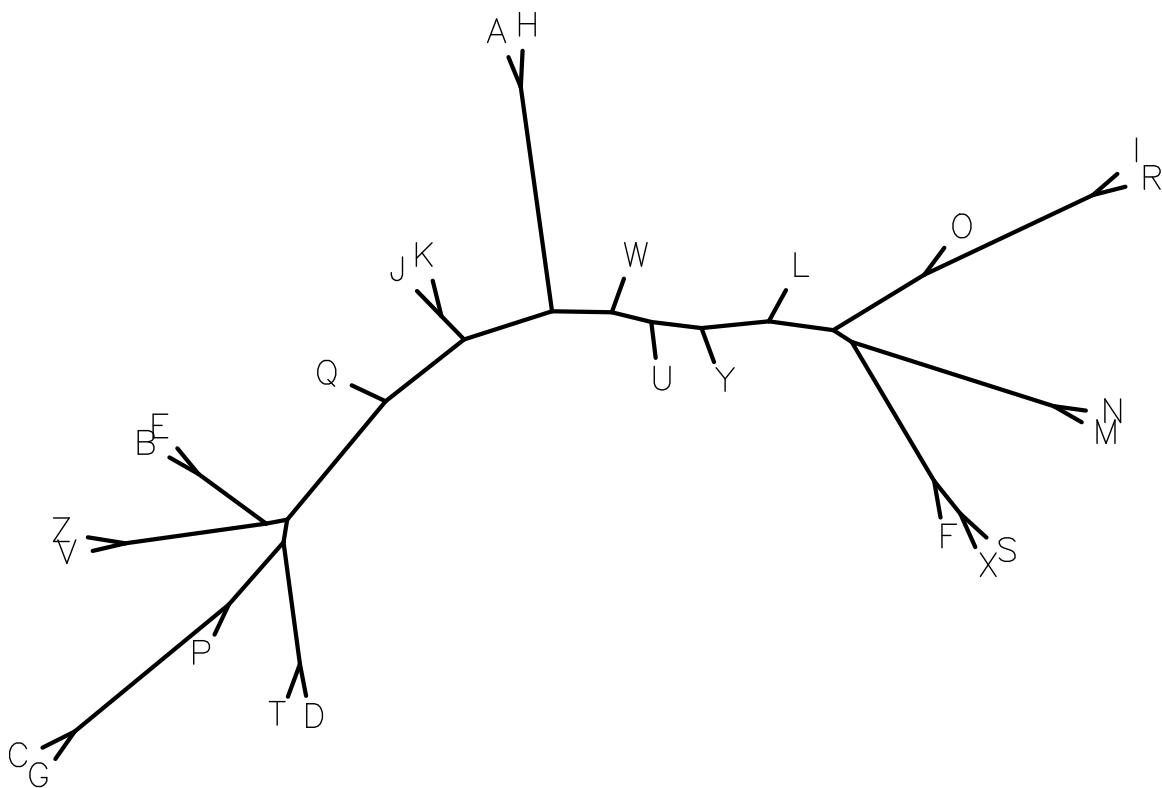


Figure 4-8: A phylogenetic tree of voice–proteins. This tree was created using the Phylib [81] tree drawing program from a multiple sequence alignment of all 26 voice–proteins from a single speaker. The multiple sequence alignment was made using the ClustalW [116] alignment tool, with the scoring matrix 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.

Experiment	Classification	Classification in Alimoglu & Alpaydin, 1996
1	97.34%	97.80%
2	99.64%	n/a

(a) Handwriting recognition results.

Experiment	Classification	Classification with clustering	Classification in Dietterich & Bakiri, 1995
1	93.84%	98.91%	96.73%
2	92.61%	98.61%	n/a

(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 Dietterich 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

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, Blosum50 [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.

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

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,-0.55, 0.11, 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 3.4t11 using the default gap open and extension penalties, and the following options: `-p -Q -d0 -f-8 -g-1 -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 physicochemical 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 BIGC670101 [36] — to the abstract — for example, the normalized frequency of participation in an N-terminal beta-sheet, which is AAindex parameter CHOP780208 [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 $3 * 453 = 1359$ 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].



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].

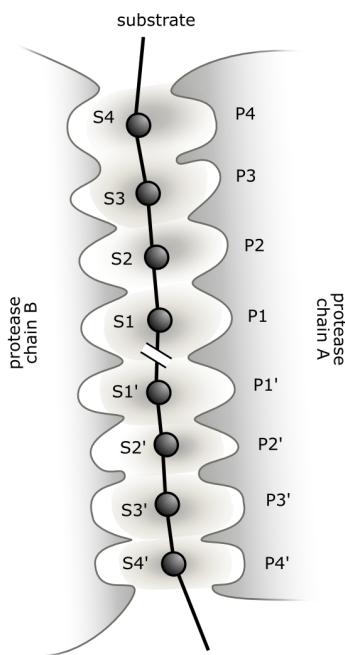


Figure 4-10: Schematic of the HIV-I protease active site. The active site comprises eight binding pockets (P_1 – P_4 and P'_1 – P'_4) into which eight residues from the target protein fall. The target protein is cleaved between the S_1 and S'_1 residues. One half of the catalytic unit is made up by chain 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 (S_4, S_4') are relatively unimportant, whereas the central residues (S_1, S_1') are quite important in determining cleavability. For the S_1 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
- MITSo₂₀₁₀₁: 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 $S_3 > 0.52$ ”. 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: S_{1'}, 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 *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.

	DT	LR	NB	BN	SVM	SVM–rbf
DT	-	2	1	3	2	2
LR	0	-	0	0	0	0
NB	0	3	-	1	2	1
BN	0	1	0	-	1	1
SVM	0	0	0	0	-	1
SVM–rbf	0	2	0	1	2	-

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.

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.

total wins	total losses	model
8	0	LR
7	1	SVM
5	3	BN
5	5	SVM-rbf
1	7	NB
0	10	DT

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.

	FULL	CFS	PCA
FULL	-	0	1
CFS	3	-	4
PCA	2	1	-

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.

5	1	FULL
5	3	PCA
1	7	CFS

```

CHOP780207_S2' <= 0.41765
| FAUJ880105_S1 <= 0.57778
|   FASG760102_S1 <= 0.27711: uncleaved (32.0/1.0)
|   FASG760102_S1 > 0.27711
|     QIAN880122_S4' <= 0.81022
|       PRAM900101_S1 <= 0.27463
|         MEEJ810102_S4 <= 0.33702
|           RACS820112_S2 <= 0.58621
|             ZIMJ680101_S1' <= 0.52117
|               PRAM820101_S2' <= 0.43367
|                 | ROSM880103_S3' <= 0.23077: cleaved (2.0)
|                 | ROSM880103_S3' > 0.23077
|                   CHOP780207_S4 <= 0.21176: cleaved (2.0)
|                   CHOP780207_S4 > 0.21176: uncleaved (11.0/1.0)
|               PRAM820101_S2' > 0.43367
|                 RADA880105_S2 <= 0.75274
|                   PRAM900101_S1 <= 0.06866: cleaved (10.0/2.0)
|                   PRAM900101_S1 > 0.06866: uncleaved (4.0)
|                 RADA880105_S2 > 0.75274
|                   QIAN880137_S3' <= 0.5124: cleaved (69.0/3.0)
|                   QIAN880137_S3' > 0.5124
|                     RACS820112_S2 <= 0.43103: cleaved (2.0)
|                     RACS820112_S2 > 0.43103: uncleaved (4.0/1.0)
|               ZIMJ680101_S1' > 0.52117: cleaved (248.0/7.0)
|             RACS820112_S2 > 0.58621
|               RACS820103_S4 <= 0.43007
|                 CHAM830104_S3' <= 0
|                   RADA880105_S2 <= 0.75274: uncleaved (5.0/1.0)
|                   RADA880105_S2 > 0.75274: cleaved (2.0)
|                 CHAM830104_S3' > 0: cleaved (11.0)
|               RACS820103_S4 > 0.43007: uncleaved (6.0)
|             MEEJ810102_S4 > 0.33702
|               GARJ730101_S4' <= 0.01426: uncleaved (9.0)
|               GARJ730101_S4' > 0.01426
|                 CHAM830104_S3' <= 0
|                   QIAN880102_S4 <= 0.57143: uncleaved (7.0/1.0)
|                   QIAN880102_S4 > 0.57143: cleaved (3.0)
|                 CHAM830104_S3' > 0: cleaved (9.0)
|             PRAM900101_S1 > 0.27463
|               GEIM800106_S1' <= 0.94
|                 RACS820102_S3 <= 0.81522
|                   FAUJ880108_S2' <= 0.4375: uncleaved (31.0/1.0)
|                   FAUJ880108_S2' > 0.4375: cleaved (4.0/1.0)
|                 RACS820102_S3 > 0.81522: cleaved (6.0)
|               GEIM800106_S1' > 0.94: cleaved (9.0)
|             QIAN880122_S4' > 0.81022
|               MITS020101_S1 <= 0.35354
|                 ZIMJ680101_S1' <= 0.82085: uncleaved (20.0)
|                 ZIMJ680101_S1' > 0.82085
|                   RACS820102_S3 <= 0.3587: uncleaved (4.0)
|                   RACS820102_S3 > 0.3587: cleaved (5.0)
|                 MITS020101_S1 > 0.35354: cleaved (2.0)
|             FAUJ880105_S1 > 0.57778
|               QIAN880137_S3' <= 0: cleaved (3.0)
|               QIAN880137_S3' > 0: uncleaved (37.0/1.0)
CHOP780207_S2' > 0.41765
|   ZIMJ680101_S1' <= 0.58306: uncleaved (145.0/2.0)
|   ZIMJ680101_S1' > 0.58306
|     PRAM900101_S1 <= 0.27463
|       FAUJ880105_S1 <= 0.57778
|         FAUJ880105_S1 <= 0: uncleaved (2.0)
|         FAUJ880105_S1 > 0
|           RACS820103_S3 <= 0.72378
|             WILM950104_S2 <= 0.44834: uncleaved (5.0)
|             WILM950104_S2 > 0.44834
|               PRAM820101_S2' <= 0.77041: cleaved (8.0)
|               PRAM820101_S2' > 0.77041: uncleaved (4.0/1.0)
|             RACS820103_S3 > 0.72378: cleaved (9.0)
|       FAUJ880105_S1 > 0.57778: uncleaved (4.0)
|     PRAM900101_S1 > 0.27463: uncleaved (12.0)

```

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, 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 CHOP780207_S2 <= 0.41765 and FAUJ880105_S1 > 0.57778 and 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.

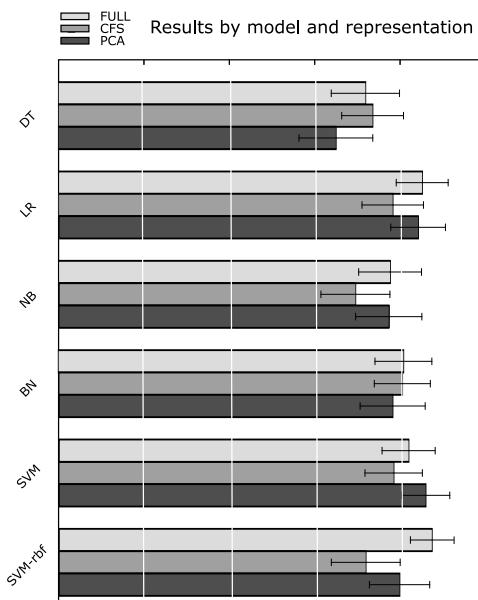


Figure 4-12: Classification results for all amino acid representations and model types. The three different amino acid representations are shown in shades of gray: “FULL” is the full physiochemical, 1241-dimensional representation; “CFS” is the feature-selected, 55-dimensional representation; and “PCA” is the 129-dimensional representation 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 frequently hampered 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 DH₅ α (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 (11) 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 μ g/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 μ M 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 [158] along with the primers PL_sense_AatII, TCCGACGTCTAAGAAACCATTATTATC and PL_anti_EcoRI, CCGGAATTCTGGTCAGTGCCTGCTGAT. 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,

AGATCCTTGGCGGCAAGAAA

and PL_Anti_Seq,

GCCATGGAACAGGTAGTTTCCAG.

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 MERA-maid 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-TET_{O1} 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-TET_{O1} 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 — P₁, P₂, …, P_M — such that there are n₁, n₂, …, n_M, mutants in each class and

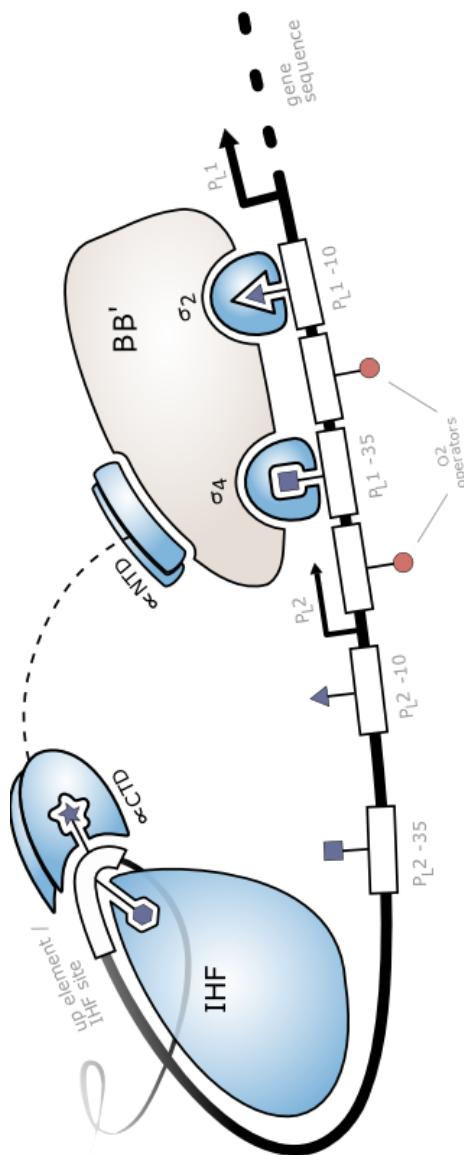


Figure 4-13: Structure of the PL-TETO1 promoter. There are numerous functional sites on the PL-TETO1 promoter that are known to effect the rate of complex formation between the promoter and RNA polymerase [95–97]. The promoter region contains two tandem promoters PL-1 and PL-2, each of which contain -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-TETO1 promoter has two tetO₂ operators from the Tn10 tetracycline resistance operon [158].

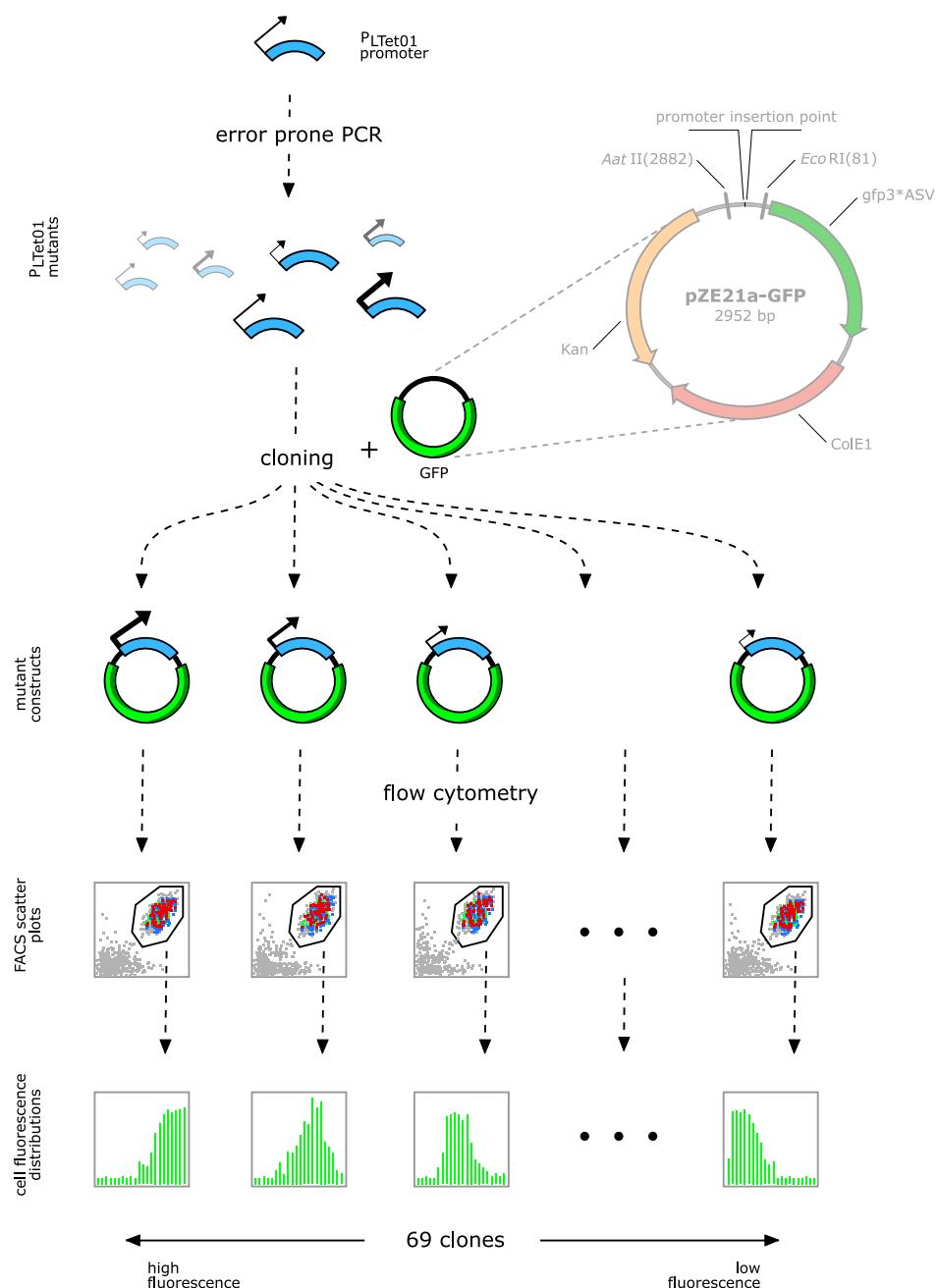


Figure 4-14: Schematic of the experimental procedure. A variant of the constitutive bacteriophage PL- λ promoter ($PL-TETO_1$) was mutated through error-prone PCR to create mutated fragments of promoters. These fragments were then ligated into plasmid constructs and used to drive the expression of *gfp* in *E. coli*. These cells were then analyzed using flow cytometry to quantify the fluorescence of GFP and output capacity of the promoter.

$\sum n_i = N$. 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, \dots, x_M will take on the particular set of values y_1, y_2, \dots, y_M is

$$\Pr(x_1 = y_1, x_2 = y_2, \dots, x_M = y_M) = \binom{X}{y_1, y_2, \dots, y_M} \prod_{i=1}^M \frac{n_i}{N} \quad (4.1)$$

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

$$\binom{X}{y_1, y_2, \dots, y_M} \prod_{i=1}^M \frac{n_i}{N} \quad (4.2)$$

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

$$\binom{X}{y_1, y_2, \dots, y_M} = \frac{X!}{y_1! y_2! \dots y_M!}. \quad (4.3)$$

The coefficient is the number of ways sets of size y_1, y_2, \dots, 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

$$\Pr(x_i \geq q) = \sum_{k=q}^X \binom{X}{q} \left(\frac{n_i}{N}\right)^k \left(1 - \frac{n_i}{N}\right)^{N-k}. \quad (4.4)$$

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 ($p\text{-value} \sim 0.5$) 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

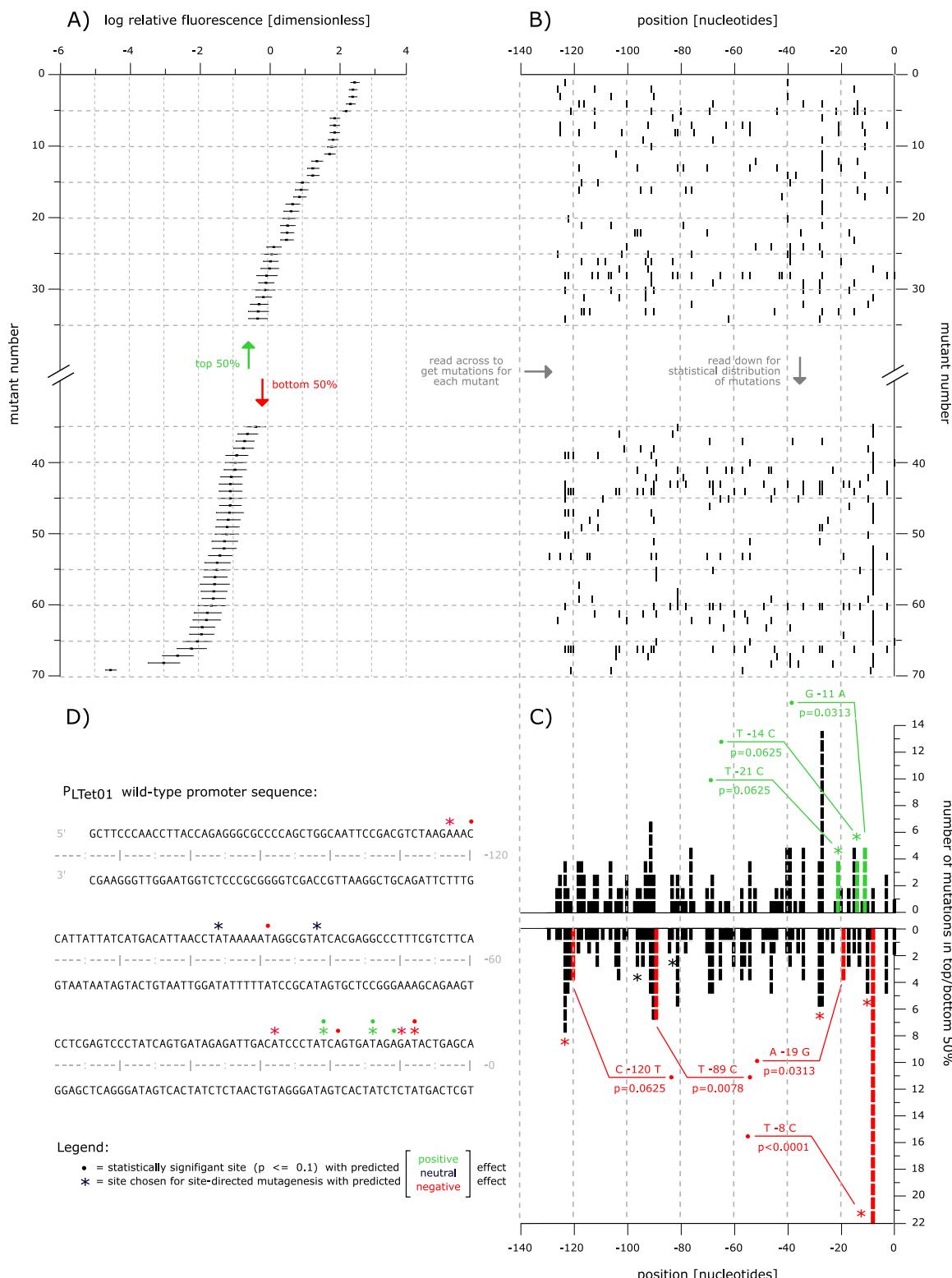


Figure 4-15: Statistical distribution of mutations and their effects on mutant fluorescence. See text for a description.

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.

Site	Predicted activity	P-value	Observations	Confidence	Relative Fluorescence	Log Relative Fluorescence	Agreement?
-8	Low	<0.0001	22	High	0.036	-3.32	Yes
-10	Low	0.1094	6	Med	0.011	-4.52	Yes
-14	High	0.0625	4	High	1.428	0.35	Yes
-21	High	0.0625	4	High	1.585	0.46	Yes
-28	Low	0.3770	10	Low	0.756	-2.58	Yes
-82	No effect	0.5000	2	Low	0.926	-0.08	Yes
-96	No effect	0.5000	5	Med	0.046	-3.08	No
-123	Low	0.1938	12	Med	0.087	-2.45	Yes

Table 4.9: Summary of double and triple mutants constructed by site-directed mutagenesis.

Sites	Predicted activity	Relative Fluorescence	Log Relative Fluorescence	Agreement?
-14, -21	High	1.924	0.65	Yes
-14, -82	High	0.954	-0.04	No
-21, -82	High	1.433	0.36	Yes
-96, -123	Low	0.274	-1.43	Yes
-82, -14, -21	High	0.140	-1.97	No
-8, -10, -28	Low	0.018	-4.03	Yes

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.

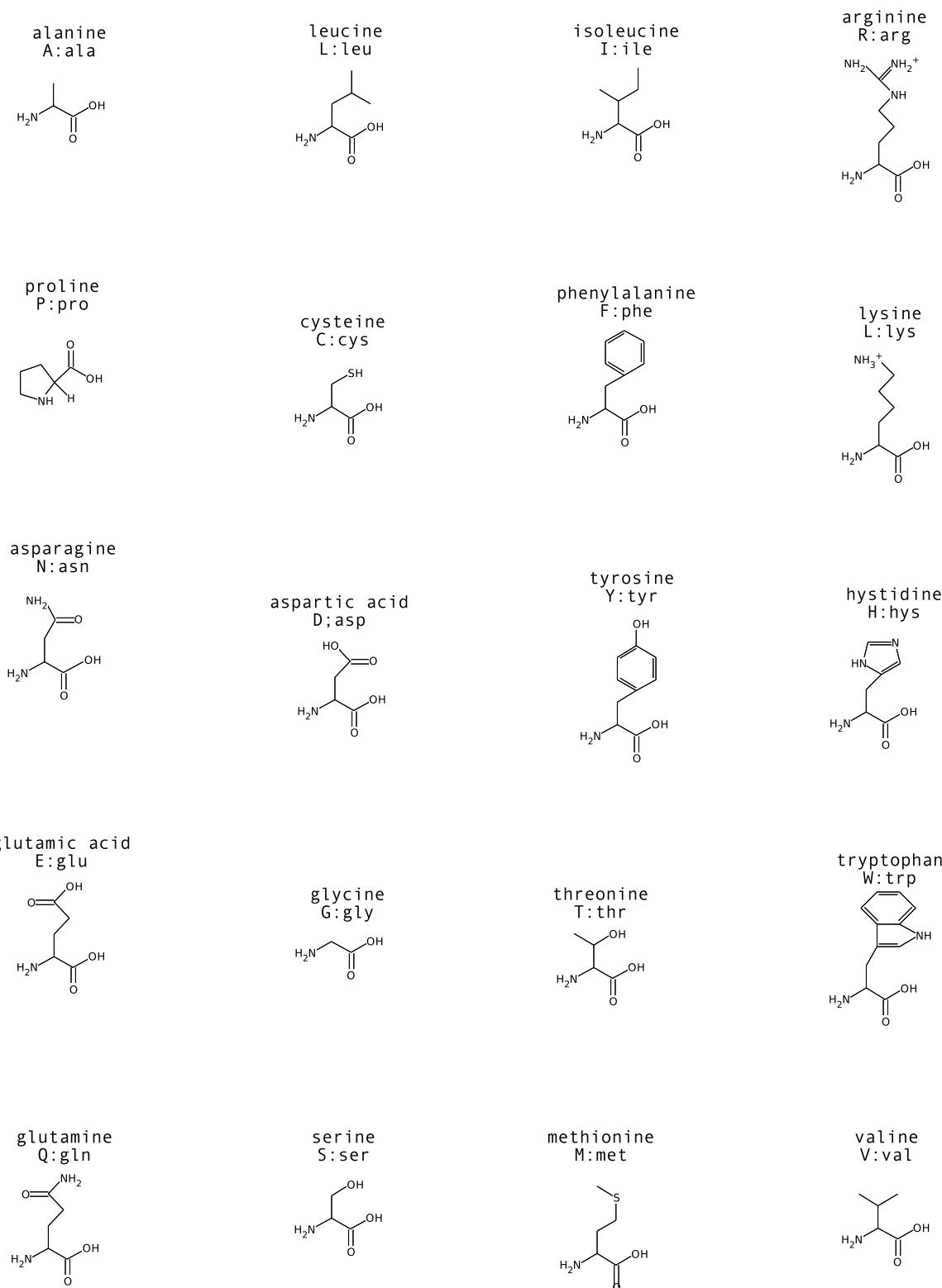


Figure A-1: Amino acid structures and abbreviations. The figure shows the chemical structure of the 20 naturally occurring amino acids and their three letter and one letter abbreviations.

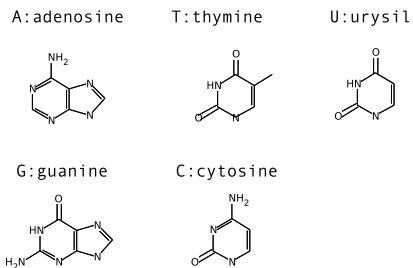
**Figure A-2:** Nucleotide base structures and abbreviations.

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.

		A	C	G	U
	AA	Lys	Asn	Lys	Asn
F	AC	Thr	Thr	Thr	Thr
i	AG	Arg	Ser	Arg	Ser
r	AU	Ile	Ile	MET	Ile
s P	CA	Gln	His	Gln	His
t o	CC	Pro	Pro	Pro	Pro
s	CG	Arg	Arg	Arg	Arg
& i	CU	Leu	Leu	Leu	Leu
t	GA	Glu	Asp	Glu	Asp
S i	GC	Ala	Ala	Ala	Ala
e o	GG	Gly	Gly	Gly	Gly
c n	GU	Val	Val	Val	Val
o	UA	.	Tyr	.	Tyr
n	UC	Ser	Ser	Ser	Ser
d	UG	.	Cys	Trp	Cys
	UU	Leu			

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.wid()):
            for i in range(self.len()):
```

```

        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):
            self.entropy[pos] = self.entropy[pos] \
                - freq * math.log(freq, base)

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

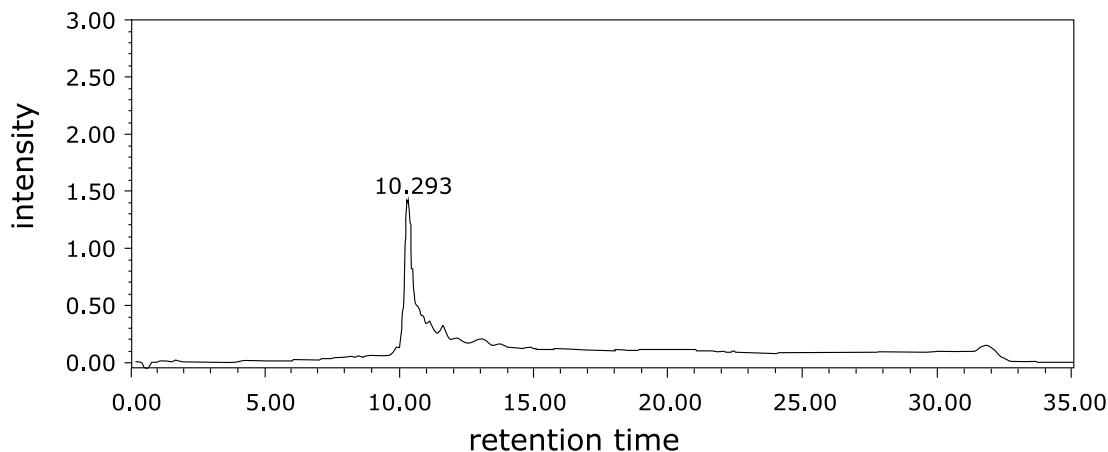
Total Information Content = 9.834

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 TTTAAC 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)



B)

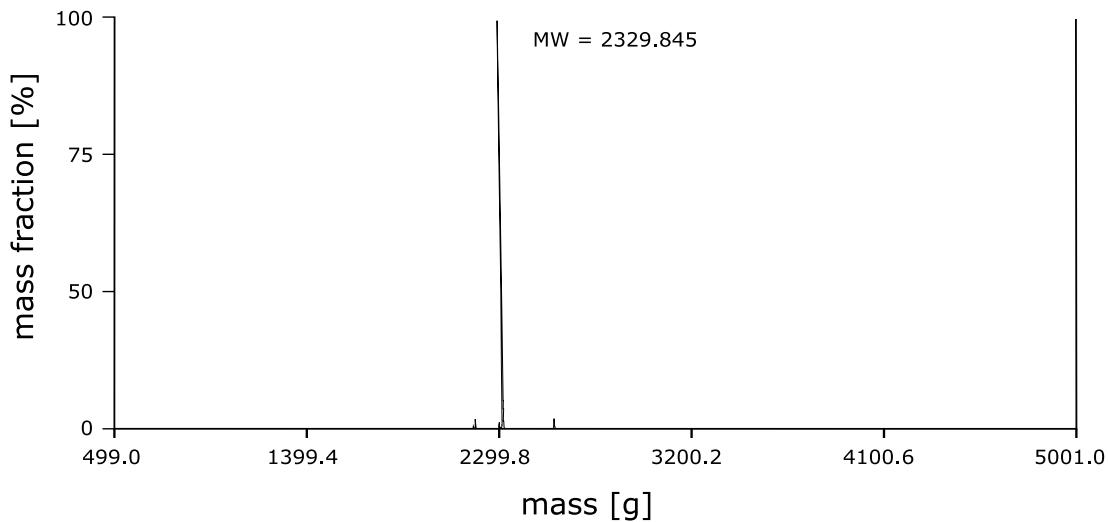


Figure A-3: Gas and mass spectra for the synth-1 peptide. As the figure shows, the peptide appears well above the 85% purity threshold. This peptide was designed using our preliminary, sensitive approach for designing antimicrobial peptides. However, the peptide was shown to have undetectable activity 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.

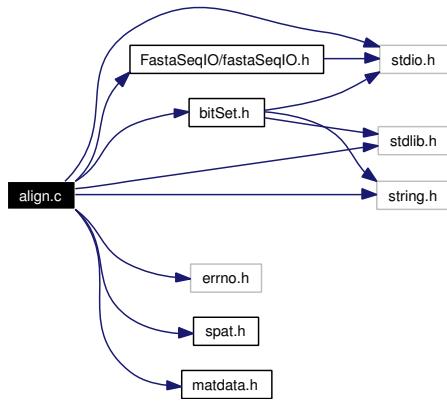
There are many other options for the `configure` script. To see a list of available options, use the optional flag `-help`.

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

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

Include dependency graph for align.c:



Defines

- #define ALIGN_ALPHABET 256

Functions

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

Variables

- const int aaOrder []

Detailed Description

This file defines functions that are used to create a similarity graph, or adjacency matrix via the comparison of small windows within a set of sequences. This file is only used for string based sequences, and not real valued data. Usually, the adjacency matrix is created via a the alignment of the windows within the sequence set. Thus, the name of this file. However, other functions can certainly be defined for creating the adjacency matrix.

Definition in file [align.c](#).

Define Documentation

B.2.0.1 #define ALIGN_ALPHABET 256

Definition at line 24 of file align.c.

Function Documentation

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

This function takes as its arguments two pointers to strings, a length, and a scoring matrix. The function computes the score, or degree of similarity, between the two strings by comparing each character in the strings from zero to $L - 1$. Each character receives a score that is looked up in the scoring matrix. This is most commonly used for amino acid sequences or DNA sequences; however, it is applicable to any series of characters. This function returns a single integer, which is the score between the two words.

Definition at line 44 of file align.c.

References aaOrder, and mat.

Referenced by alignWordsMat_bit().

```

45 {
46     int i;
47     int points = 0;
48     int x, y;
49
50     // Go over each character in the L-length window
51     for (i = 0; i < L; i++)
52     {
53
54         // The integer corresponding to the character in
55         // the first string, so that we can look it up
56         // in one of our scoring matrices.
57         x = aaOrder[(int) s1[i]];
58
59         // And for the second character
60         y = aaOrder[(int) s2[i]];
61
62         // If the characters aren't going to be in the scoring
63         // matrix, they get a -1 value...which we'll give zero
64         // points to here.
65         if (x != -1 && y != -1)
66         {
67
68             // Otherwise, they get a score that is looked up
69             // in the scoring matrix
70             points += mat[x][y];
71         }
72     }
73     return points;
74 }
```

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

This uses the function above. Here, we have an array of words ([sPat_t](#) objects) and we compare (align) them all. If their score is above 'threshold' then we will set a bit to 'true' in a [bitGraph_t](#) that we create. A [bitGraph_t](#) is essentially an adjacency matrix, where each member of the matrix contains only a single bit: are the words equal, true or false? The function traverses the words by doing all by all comparison; however, we only do the upper diagonal. The

function makes use of alignMat and needs to be passed a scoring matrix that the user has chosen which is appropriate for the context of whatever data sent the user is looking at.

Definition at line 88 of file align.c.

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

Referenced by main().

```

90 {
91     bitGraph_t * sg = NULL;
92     int score;
93     int i, j;
94
95     // Assign a new bitGraph_t object, with (wc x wc) possible
96     // true/false values
97     sg = newBitGraph (wc);
98     for (i = 0; i < wc; i++)
99     {
100         for (j = i; j < wc; j++)
101     {
102
103         // Get the score for the alignment of word i and word j
104         score =
105         alignMat (words[i].string, words[j].string, words[i].length, mat);
106
107         // If that score is greater than threshold, set
108         // a bit to 'true' in our bitGraph_t object
109         if (score >= threshold)
110         {
111
112             // We use 'bitGraphSetTrueSym' because, if i=j,
113             // then j=i for most applications. However, this
114             // can be relaxed for masochists.
115             bitGraphSetTrueSym (sg, i, j);
116         }
117     }
118 }
119
120     // Return a pointer to this new bitGraph_t object
121     return sg;
122 }
```

Variable Documentation

B.2.0.4 const int aaOrder[]

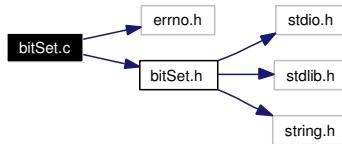
Definition at line 32 of file matrices.h.

Referenced by alignMat().

B.3 bitSet.c File Reference

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

Include dependency graph for bitSet.c:



Functions

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

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

Detailed Description

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

Definition in file `bitSet.c`.

Function Documentation

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`.

```

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
361     tmp = n - ((n >> 1) & 033333333333) - ((n >> 2) & 011111111111);
362     return ((tmp + (tmp >> 3)) & 030707070707) % 63;
364 }
```

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().

```

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

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.

```

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

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().

```

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

B.3.o.9 int bitGraphRowIntersection ([bitGraph_t](#) * *bg*, int *row1*, int *row2*, [bitSet_t](#) * *si*)

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 o (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], si);
601     return 0;
602 }
```

B.3.o.10 int bitGraphRowUnion ([bitGraph_t](#) * *bg*, int *row1*, int *row2*, [bitSet_t](#) * *si*)

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 o (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], si);
587     return 0;
588 }
```

B.3.o.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 o (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 }
```

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

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

Definition at line 714 of file bitSet.c.

References *bitGraph_t*::*graph*, and *setFalse()*.

Referenced by *convolve()*.

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

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

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

Definition at line 669 of file bitSet.c.

References *bitGraph_t*::*graph*, and *setFalse()*.

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

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

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

Definition at line 641 of file bitSet.c.

References *bitGraph_t*::*graph*, and *setTrue()*.

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

B.3.o.15 int bitGraphSetTrueDiagonal (*bitGraph_t* * *bg*)

Sets the main diagonal of a bitGraph true. Input: a *bitGraph_t* object. Output: integer success value of o (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.o.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 o (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.o.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 o (and an altered destination *bitSet_t* object with a true where all three source bitSets had a true.)

Definition at line 327 of file bitSet.c.

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

```

329 {
330     int i;
331     if ((s1->slots != s2->slots) || (s1->slots != s3->slots)
332     || (s1->slots != s4->slots))
333     {
334         fprintf (stderr, "Sets aren't same size!\n");
335         fflush (stderr);
336         exit (0);
337     }
338     for (i = 0; i < s1->slots; i++)
339     {
340         s4->tf[i] = BSINTERSECTION (s1->tf[i], s2->tf[i]);
341         s4->tf[i] = BSINTERSECTION (s3->tf[i], s4->tf[i]);
342     }
343     return 0;
344 }
```

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](#).

```

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

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     {
```

```

187     fprintf (stderr, "Sets aren't same size!\n");
188     fflush (stderr);
189     exit (0);
190   }
191   for (i = 0; i < s1->slots; i++)
192   {
193     s3->tf[i] = BSUNION (s1->tf[i], s2->tf[i]);
194   }
195   return 0;
196 }
```

B.3.0.22 int checkBit ([bitSet_t * s1](#), [int x](#))

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

Definition at line 148 of file bitSet.c.

References BSTEST, and bitSet_t::tf.

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

```

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

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\(\)](#).

```

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

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](#).

```

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

B.3.0.26 int countSet ([bitSet_t](#) * *s1*)

Counts the number of true values in a bitSet. Input: a [bitSet_t](#) object. Output: number of true values in that [bitSet_t](#) object.

Definition at line 437 of file bitSet.c.

References [bitcount32_precomp\(\)](#), and [bitSet_t::tf](#).

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

```

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

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.3.0.28 int deleteBitSet ([bitSet_t](#) * *sl*)

Performs memory management for the deletion of a [bitSet_t](#) structure. Input: a [bitSet_t](#) object. Output: integer success value of 1.

Definition at line 159 of file bitSet.c.

References [bitSet_t::tf](#).

Referenced by [convolve\(\)](#), [deleteBitGraph\(\)](#), [filterGraph\(\)](#), [findCliques\(\)](#), [getStatMat\(\)](#), [oldGetStatMat\(\)](#), [wholeCliqueConv\(\)](#), and [wholeRoundConv\(\)](#).

```

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

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](#).

```

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

B.3.0.31 int emptySet ([bitSet_t](#) * *st*)

Sets all values in a bitSet to false. Input: a [bitSet_t](#) object. Output: integer success value of 1.

Definition at line 136 of file bitSet.c.

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

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

```

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

B.3.0.32 int fillBitGraph ([bitGraph_t](#) * *bgr*)

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](#) * *st*)

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

Definition at line 124 of file bitSet.c.

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

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

```

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

B.3.o.34 int flipBits ([bitSet_t](#) * *sl*)

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 < sl->slots; i++)
112     {
113         sl->tf[i] = ~sl->tf[i];
114     }
115     return 0;
116 }
```

B.3.o.35 int maskBitGraph ([bitGraph_t](#) * *bg1*, [bitSet_t](#) * *bs*)

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

Definition at line 752 of file bitSet.c.

References [bitSetIntersection\(\)](#), [checkBit\(\)](#), [emptySet\(\)](#), and [bitGraph_t::graph](#).

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

B.3.o.36 [bit_t*](#) newBitArray (int *bytes*)

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

Definition at line 20 of file bitSet.c.

Referenced by newBitSet().

```

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

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()**.

```

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.o.38 `bitSet_t*` newBitSet (`int size`)

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

Definition at line 43 of file bitSet.c.

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

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

```

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

B.3.o.39 `int` nextBitBitSet (`bitSet_t * s1, int start`)

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

Definition at line 468 of file bitSet.c.

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

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

```

469 {
470     // slot is our starting slot, the
471     // slot containing bit 'start'
472     int slot = BITSLOT (start);
473     int i;
474     // stop is the bit to stop it --- it is equal to max, and it is
475     // the index of a bit that does NOT belong to the bitset
476     int stop;
477     bit_t bitFalse;
478     memset (&bitFalse, 0, sizeof (bit_t));
479
480     // s1->max is the number of bits in s1
481     // test to see if we're looking too high
482     if (start >= s1->max)
```

```

484      {
485          return -1;
486      }
487 // s1->slots is the number of available slots
488 // skip over empty slots
489 while (slot < s1->slots)
490 {
491     /*
492         printf("w");
493     */
494     if (s1->tf[slot] != bitFalse)
495     {
496         // this slot is not empty
497
498         // if each slot is, say 32 bits and
499         // we asked for nextBitBitSet(s1, 5),
500         // then slot 0 will be non-zero. but,
501         // instead of starting at 0, start at 5!
502         if (BSBITSIZE * slot > start)
503         {
504             // set start to index of first
505             // bit in this slot
506             start = BSBITSIZE * slot;
507         }
508         // set the stop, with a check against the 'max'
509         // element of the bitSet_t object
510         if (BSBITSIZE * (slot + 1) > s1->max)
511         {
512             stop = s1->max;
513         }
514         else
515         {
516             stop = BSBITSIZE * (slot + 1);
517         }
518         for (i = start; i < stop; i++)
519         {
520             if (checkBit (s1, i))
521             {
522                 return i;
523             }
524         }
525     }
526     slot++;
527 }
528 return -1;
529 }
```

B.3.0.40 int printBinaryBitSet ([bitSet_t](#) * *si*)

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

Definition at line 611 of file bitSet.c.

References BSTEST, and [bitSet_t::tf](#).

Referenced by [printBitGraph\(\)](#).

```

612 {
613     int i;
614     for (i = 0; i < s1->max; i++)
615     {
```

```

616     printf ("%d", (BSTEST (s1->tf, i) ? 1 : 0));
617 }
618 return 0;
619 }
```

B.3.0.41 int printBitGraph ([bitGraph_t](#) * *bg*)

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

Definition at line 730 of file bitSet.c.

References [bitGraph_t::graph](#), and [printBinaryBitSet\(\)](#).

```

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

B.3.0.42 int printBitSet ([bitSet_t](#) * *si*)

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

Definition at line 555 of file bitSet.c.

References [BSTEST](#), and [countSet\(\)](#).

```

556 {
557     int i;
558     printf ("bitSet (addr = %d; %d members)\n", (int) s1, countSet (s1));
559     printf ("\tmax = %d\n", s1->max);
560     printf ("\tslots = %d\n", s1->slots);
561     printf ("\tbytes = %d\n", s1->bytes);
562     printf ("\tmembers = ");
563
564     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 }
574 }
```

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().

```

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().

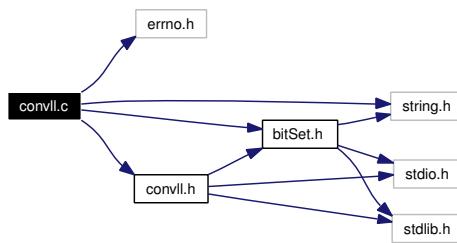
```

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

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

Include dependency graph for convll.c:



Functions

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

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

Variables

- `int cliquecounter = 0`

Detailed Description

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

Definition in file `convll.c`.

Function Documentation

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, "\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 212 of file convll.c.

References [countSet\(\)](#), [cSet_t::members](#), [nextBitBitSet\(\)](#), and [cSet_t::size](#).

Referenced by [pushClique\(\)](#), and [wholeCliqueConv\(\)](#).

```

213 {
214     int cliqueSize = countSet (clique);
215     int i = 0, start = 0;
216     cSet_t *holder = (cSet_t *) malloc (sizeof (cSet_t));
217
218     // Memory error checking
219     if (holder == NULL)
220     {
221         fprintf (stderr, "\nMemory Error - bitSetToCSet - [1]\n%s\n",
222                 strerror (errno));
223         fflush (stderr);
224         exit (0);
225     }
226     // More memory checking
227     holder->members = (int *) malloc (cliqueSize * sizeof (int));
228     if (holder->members == NULL)
229     {
230         fprintf (stderr, "\nMemory Error - bitSetToCSet - [2]\n%s\n",
231                 strerror (errno));
232     }
233 }
```

```

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 checkCliqueset ([cSet_t](#) * *cliqueset*, int * *indexToSeq*, int *p*)

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

Definition at line 266 of file convll.c.

References [cSet_t::members](#), and [cSet_t::size](#).

Referenced by [pushClique\(\)](#).

```

267 {
268     int *seqNums = NULL;
269     int thisSeq = 0, i = 0, j = 0;
270     seqNums = (int *) malloc (p * sizeof (int));
271
272     if (seqNums == NULL)
273     {
274         fprintf (stderr, "Memory error - checkCliqueset\n%s\n",
275                 strerror (errno));
276         fflush (stderr);
277         exit (0);
278     }
279     // Initialize an array of integers of size p to sentinel values of -1
280     for (i = 0; i < p; i++)
281     {
282         seqNums[i] = -1;
283     }
284     j = 0;
285
286     if (cliqueset->size < 1)
287     {
288         fprintf (stderr, "\nClique of zero size! - checkCliqueset\n");
289         fflush (stderr);
290         exit (0);
```

```

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

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

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

Definition at line 1417 of file convll.c.

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

Referenced by convolve().

```

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

```

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

After we have performed a round of convolution, this "empties" the member stacks by popping all nodes off each member linked list. Input: array of member linked lists, the size of that array (total number of offsets). Output: the array of now-empty member linked lists.

Definition at line 538 of file convll.c.

References `popWholeMemStack()`.

Referenced by `completeConv()`.

```

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

B.4.0.50 `mll_t** fillMemberStacks (cll_t * head, mll_t ** memberStacks)`

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

Definition at line 517 of file `convll.c`.

References `addToStacks()`, and `cnode::next`.

Referenced by `completeConv()`.

```

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

B.4.0.51 `cll_t* initheadCll (cll_t * head, cSet_t * newset)`

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

Definition at line 172 of file `convll.c`.

References `cnode::set`.

Referenced by `pushcSet()`.

```

173 {
174     // Check to make sure that the head is not already initialized.
```

```

175     if (head->set != NULL)
176     {
177         printf ("Stack head already initialized!");
178         exit (0);
179     }
180 // Make the head's set pointer point to the new set.
181 head->set = newset;
182 return head;
183 }
```

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

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

Definition at line 759 of file convll.c.

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

Referenced by singleCliqueConv().

```

761 {
762
763     int i = 0, j = 0, status = 0;
764
765     // Make sure we are still in-bounds, otherwise we bail out
766     // We'll refer to the offset currently being analyzed from the
767     // first clique as the 'first offset' and the offset currently
768     // being analyzed from the second clique as the 'second offset'
769     while ((i < first->set->size) && (j < second->set->size))
770     {
771         // If the second offset is earlier than the first offset plus
772         // one, then we move on to the next possible second offset
773         if ((first->set->members[i] + 1) > second->set->members[j])
774     {
775         j++;
776     }
777     // If the second offset is later than the first offset plus
778     // one, then we move on the next possible first offset
779     else if ((first->set->members[i] + 1) < second->set->members[j])
780     {
781         i++;
782     }
783     // Otherwise, the second offset is equal to the first offset
784     // plus one, so we have an extendable node. Push that on
785     // to the intersection stack, move both the first and second
786     // offsets to their respective next possible offsets, and
787     // increment the support counter for the new clique (status)
788     else
789     {
790         intersection = pushMemStack (intersection, first->set->members[i]);
791         i++;
792     }
793 }
```

```

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.4.0.53 `cSet_t* mllToCSet (mll_t * clique)`

Turns a member linked list used to store the intersection of two cliques into something more useful: a `cSet_t` structure. Input: a clique in `mll_t` form. Output: a clique in `cSet_t` form.

Definition at line 1145 of file convll.c.

References `mnode::cliqueMembership`, `cSet_t::members`, `mnode::next`, and `cSet_t::size`.

Referenced by `pushConvClique()`.

```

1146 {
1147     int sizecount = 0, i = 0;
1148     cSet_t *cliqueCset = malloc (sizeof (cSet_t));
1149     mll_t *head = clique;
1150     if (cliqueCset == NULL)
1151     {
1152         fprintf (stderr, "Memory error - mllToCSet cSet\n%s\n",
1153                 strerror (errno));
1154         fflush (stderr);
1155         exit (0);
1156     }
1157     // First count up how many members there are in the member linked list
1158     while (head != NULL)
1159     {
1160         sizecount++;
1161         head = head->next;
1162     }
1163
1164     head = clique;
1165     cliqueCset->size = sizecount;
1166     cliqueCset->members = (int *) malloc (sizecount * sizeof (int));
1167
1168     if (cliqueCset->members == NULL)
1169     {
1170         fprintf (stderr, "Memory error - mllTlCSet cliqueMembers\n%s\n",
1171                 strerror (errno));
1172         fflush (stderr);
1173         exit (0);
1174     }
1175     // In order to stay in the same format as with bitSet translation to
1176     // cSet, we ensure that the ids of the members are ascending with
1177     // ascending index number in the cSet. This is accomplished by noting
1178     // that since the intersection members are pushed onto the stack,
1179     // a LIFO operation, that the first intersected nodes off the stack
1180     // will have the highest ids, so we will put them at the end of
1181     // the members array with the higher index values.
1182     for (i = sizecount - 1; i >= 0; i--)
1183     {
1184         cliqueCset->members[i] = head->cliqueMembership;
1185         head = head->next;
1186     }

```

```

1187
1188     return cliqueCset;
1189 }
```

B.4.o.54 ***cll_t* popAllCll (cll_t * head)***

Shortcut function to pop all of the members of a linked list. Input: head of a linked list. Output: head of a now-empty linked list.

Definition at line 109 of file convll.c.

References popCll().

Referenced by completeConv(), and main().

```

110 {
111     while (head != NULL)
112     {
113         head = popCll (head);
114     }
115     return head;
116 }
```

B.4.o.55 ***cll_t* popCll (cll_t * head)***

Removes the head of the clique linked list, returns the new head of the clique linked list, and frees the memory occupied by the old head. Input: head of a linked list. Output: head of a linked list.

Definition at line 66 of file convll.c.

References cSet_t::members, cnode::next, and cnode::set.

Referenced by popAllCll().

```

67 {
68     // by default the new head is NULL...is important later
69     cll_t *newHead = NULL;
70     if (head == NULL)
71     {
72         fprintf (stderr, "\nCan't pop a null linked list\n");
73         fflush (stderr);
74         exit (0);
75     }
76     // unless this is the end of the linked list, set the new head
77     // to the next member of the list. Otherwise, since by default the
78     // new head is NULL, it will properly return an empty list
79     if (head->next != NULL)
80     {
81         newHead = head->next;
82     }
83     // Check to see if there is a set. If there is, and there are members,
84     // then first free the members. And if there is a set, then free it.
85     if (head->set != NULL)
86     {
87         if (head->set->members != NULL)
```

```

88     {
89         free (head->set->members);
90         head->set->members = NULL;
91     }
92     free (head->set);
93     head->set = NULL;
94 }
95 // Both the members and set have been freed, so now can free the cll_t
96 // without leaking anything.
97
98 free (head);
99 head = NULL;
100 return newHead;
101 }
```

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, "\nCan't pop a null linked list - popMemStack\n");
447         fflush (stderr);
448         exit (0);
449     }
450     if (head->next != NULL)
451     {
452         newHead = head->next;
453     }
454     free (head);
455     head = NULL;
456     return newHead;
457 }
```

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)
```

```

468     {
469         head = popMemStack (head);
470     }
471     return head;
472 }
```

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.

```

129 {
130     int i = 0;
131     cll_t *curr = head;
132     while (curr != NULL)
133     {
134         printf ("id = %d\n", curr->id);
135         // Make sure the clique is nonzero in size before attempting
136         // to print it
137         if ((curr->set != NULL) && (curr->set->size > 0))
138     {
139         if (curr->length >= 0)
140         {
141             printf ("Length = %d\n", curr->length);
142         }
143         printf ("Size = %d\n", curr->set->size);
144         printf ("Members = \n");
145         for (i = 0; i < curr->set->size; i++)
146         {
147             printf ("\t%d\n", curr->set->members[i]);
148         }
149         printf ("*****\n");
150     }
151     else
152     {
153         fprintf (stderr, "\nClique has no members! -- printCll\n");
154         fflush (stderr);
155         exit (0);
156     }
157     curr = curr->next;
158 }
159     return EXIT_SUCCESS;
160 }
```

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.

Definition at line 1482 of file convll.c.

References cnode::id, cSet_t::members, cnode::set, and cSet_t::size.

```

1483 {
1484     int i = 0;
1485
1486     printf ("\nSupport = %d\t(id = %d)\n", node->set->size, node->id);
1487     printf ("Members = \n");
1488     for (i = 0; i < node->set->size; i++)
1489     {
1490         printf ("\t%d\n", node->set->members[i]);
1491     }
1492     return 1;
1493 }
```

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 }
```

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.

```

558 {
559     int i = 0;
560     mll_t *curr = NULL;
561
562     for (i = 0; i < size; i++)
563     {
564         curr = memberStacks[i];
565         printf ("Offset %d: ", i);
566         while (curr != NULL)
567         {
568             printf ("%d,", curr->cliqueMembership);
569             curr = curr->next;
570         }
571         printf ("\n");
572     }
573 }
```

B.4.o.62 `cll_t* pruneCll (cll_t * head, int * indexToSeq, int p)`

Prunes a motif linked list of all motifs without support in at least unique source sequences. Input: head of a motif linked list, pointer to a structure that dereferences offset indices to sequence numbers, minimum number of unique source sequences in which a motif must occur. Output: head of a (potentially altered) motif linked list.

Definition at line 514 of file newConv.c.

References cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by completeConv(), and convolve().

```

515 {
516     int i = 0, j = 0, thisSeq = 0;
517     int *seqNums = NULL;
518     cll_t * curr = head;
519     cll_t * prev = NULL;
520     cll_t * storage = NULL;
521
522     // We'll do this similar to the pruneBitGraph function... we will
523     // keep track of which source sequence each motif occurrence was in.
524     // Again, since the occurrences are listed monotonically, we only
525     // need to compare the last non-sentinel index to the current
526     // sequence number.
527     seqNums = (int *) malloc (p * sizeof (int));
528     if (seqNums == NULL)
529     {
530         fprintf (stderr, "Memory error - pruneCll\n%s\n", strerror (errno));
531         fflush (stderr);
532         exit (0);
533     }
534     while (curr != NULL)
535     {
536
537         // First make sure the set size is at least p.
538         // This is redundant, but extremely simple and not expensive,
539         // so we'll leave it in just as a check.
540         if (curr->set->size < p)
541         {
542             if (prev != NULL)
543             {
544                 prev->next = curr->next;
545             }
546         }
547     }
548 }
```

```
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
// p must be 2 or greater, so we can start at 1 without
// worrying about segfaulting
567 for (i = 1; i < curr->set->size; i++)
568 {
569     thisSeq = indexToSeq[curr->set->members[i]];
570     if (thisSeq != seqNums[j])
571     {
572         j++;
573         seqNums[j] = thisSeq;
574         if (j == p - 1)
575         {
576             break;
577         }
578     }
579 }
580
// Same story as before... if the last number is -1,
// then we didn't have enough to fill up the <p> different
// slots, so this doesn't meet our criterion.
584 if (seqNums[p - 1] == -1)
585 {
586     if (prev != NULL)
587     {
588         prev->next = curr->next;
589     }
590     else
591     {
592         head = curr->next;
593     }
594     storage = curr->next;
595     free (curr->set->members);
596     free (curr->set);
597     free (curr);
598     curr = storage;
599 }
600 else
601 {
602     prev = curr;
603     curr = curr->next;
604 }
605 }
606 free (seqNums);
607 return (head);
608 }
```

B.4.o.63 ***cll_t** pushClique (*bitSet_t* * *clique*, *cll_t* * *head*, *int* * *indexToSeq*, *int* *p*)**

Pushes a bitSet onto a clique linked list, performing all necessary manipulations in order to do so. Input: new clique in the form of a *bitSet_t*, head of a linked list, pointer to the index/sequence number data structure, integer value of the -p flag. Output: head of an updated clique linked list.

Definition at line 345 of file convll.c.

References *bitSetToCSet()*, *checkCliquecSet()*, *cliquecounter*, and *pushcSet()*.

Referenced by *findCliques()*, and *singleLinkage()*.

```

346 {
347     cSet_t *cliquecSet = NULL;
348
349     // Change the bitSet_t to a cSet_t
350     cliquecSet = bitSetToCSet (clique);
351     // If the -p flag has been assigned a value, then check the clique
352     // and only proceed if that criterion is met. Otherwise, free the
353     // memory that we had allocated up to this point.
354     if (p > 1)
355     {
356         if (checkCliquecSet (cliquecSet, indexToSeq, p))
357         {
358             cliquecounter++;
359             /*
360                 printf("%d\n",cliquecounter);
361             */
362             /*
363                 fflush(stdout);
364             */
365             head = pushcSet (head, cliquecSet);
366         }
367         else
368         {
369             free (cliquecSet->members);
370             free (cliquecSet);
371         }
372         // If the -p flag wasn't set, then just push the cSet onto the linked
373         // list.
374     }
375     else
376     {
377         cliquecounter++;
378         /*
379             printf("%d\n",cliquecounter);
380         */
381         /*
382             fflush(stdout);
383         */
384         head = pushcSet (head, cliquecSet);
385     }
386     return head;
387 }
```

B.4.o.64 ***cll_t** pushCll (*cll_t* * *head*)**

Pushes a new, empty head onto a linked list of cliques. Note: this should always be followed by a call to *initheadCll*, as the head pushed on here is empty and will be meaningless without

any members. This function should NOT be used by the user; see pushcSet. Input: head of a linked list. Output: head of a linked list.

Definition at line 28 of file convll.c.

References cnode::id, cnode::length, cnode::next, cnode::set, and cnode::stat.

Referenced by pushcSet().

```

29 {
30     // Make a pointer, verify memory
31     cll_t *a = NULL;
32     a = (cll_t *) malloc (sizeof (cll_t));
33     if (a == NULL)
34     {
35         fprintf (stderr, "\nMemory Error - pushCll\n%s\n", strerror (errno));
36         fflush (stderr);
37         exit (0);
38     }
39     // Initialize id (sequential) and pointer to next item, but not
40     // the cSet with the clique members
41     if (head == NULL)
42     {
43         a->id = 0;
44         a->next = NULL;
45     }
46     else
47     {
48         a->next = head;
49         a->id = head->id + 1;
50     }
51     a->set = NULL;
52     a->length = -1;
53     a->stat = -1;
54     return a;
55 }
```

B.4.0.65 ***cll_t** pushConvClique (*mll_t* clique, cll_t* head*)**

Pushes a freshly-convolved clique, currently in mll_t form, onto the clique linked list for the next level. Also checks to make sure that the convolved clique is unique, and if it isn't, it takes appropriate action. Input: a convolved clique in mll_t form, the head of a clique linked list for the next level. Output: (potentially new) head of the clique linked list for the next level.

Definition at line 1099 of file convll.c.

References cSet_t::members, mllToCSet(), pushcSet(), removeSupers(), swapNodecSet(), and uniqClique().

Referenced by singleCliqueConv().

```

1100 {
1101     int status = 0;
1102     cSet_t *cliquecSet = NULL;
1103
1104     // First change the clique to something we can used more easily
1105     cliquecSet = mllToCSet (clique);
1106     // Then check to make sure it's unique by finding out its status
1107     status = uniqClique (cliquecSet, head);
```

```

1108
1109 // printf("Candidate:\n");
1110 // printCSet(cliquecSet);
1111
1112 // If we get -2, then this clique is a subset, so just free
1113 // the cSet we just made and move on.
1114 if (status == -2)
1115 {
1116     free (cliquecSet->members);
1117     free (cliquecSet);
1118     cliquecSet = NULL;
1119 }
1120 // If we get -1, then this is a unique clique, so push it on.
1121 else if (status == -1)
1122 {
1123     head = pushcSet (head, cliquecSet);
1124 }
1125 // Otherwise, this clique is a superset, so we'll first remove
1126 // all of the other cliques of which this is a superset. Then
1127 // we'll swap out the first clique of which this is a superset
1128 // with this current clique. The clique being removed is free'd
1129 // within the swapNode function.
1130 else
1131 {
1132     head = removeSupers (head, status, cliquecSet);
1133     head = swapNodecSet (head, status, cliquecSet);
1134 }
1135 return head;
1136 }
```

B.4.o.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.o.67 ***mll_t** *pushMemStack* (*mll_t * head*, *int cliqueNum*)**

This begins code for the member linked lists. A single one of these linked lists functions somewhat similarly to the clique linked lists, though with less information stored. Functionally, an array of member linked lists is used to access the "inverse" of what is contained in the clique linked lists. That is, we would like to be able to look up the cliques that a given node is a member of, so we have an array of member linked lists of size equal to the number of nodes.

This function pushes a single clique membership onto a node's member stack. Input: the head of a single member linked list, a clique number to be added. Output: the head of a single member linked list.

Definition at line 404 of file convll.c.

References mnode::cliqueMembership, and mnode::next.

Referenced by addToStacks(), and mergeIntersect().

```

405 {
406     mll_t *a = NULL;
407     a = (mll_t *) malloc (sizeof (mll_t));
408     // Memory error checking
409     if (a == NULL)
410     {
411         fprintf (stderr, "\nMemory Error - pushMemStack: %s\n",
412                 strerror (errno));
413         fflush (stderr);
414         exit (0);
415     }
416     if (head == NULL)
417     {
418         a->next = NULL;
419     }
420     else
421     {
422         a->next = head;
423     }
424     // Store the number of the clique of which the node is a member.
425     // Note that we assume no duplication, which is guaranteed
426     // by our method of filling the member stacks, which is quite simple:
427     // go through all members of a clique (which have no duplicates
428     // because they are constructed from merge-intersections or from
429     // bitSet_t's) and add that clique to each node's membership list.
430     a->cliqueMembership = cliqueNum;
431     return a;
432 }
```

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().

```
954     int foundStatus = 0;
955     cll_t *curr = head;
956     cll_t *prev = NULL;
957     int i = 0, j = 0, breakFlag = 0;
958
959     while (curr != NULL)
960     {
961         if (curr->id == node)
962         {
963             foundStatus = 1;
964             break;
965         }
966         curr = curr->next;
967     }
968
969     if (foundStatus == 0)
970     {
971         fprintf (stderr, "\nFirst clique not found! (removeSupers)\n");
972         fflush (stderr);
973         exit (0);
974     }
975     // Now this is trickier, to remove nodes from the middle of a linked
976     // list; this means that we need to remember which node we were just
977     // at so that we can connect it to the node after the one we are
978     // about to delete.
979     prev = curr;
980     curr = curr->next;
981
982     // This code is similar to that in uniqClique.
983     // Descend through all members of the next level's linked list.
984     while (curr != NULL)
985     {
986         i = 0;
987         j = 0;
988         breakFlag = 0;
989         // The proposed convolved clique will be referred to as the
990         // 'first' clique, and the current clique being analyzed
991         // in the next level is the 'second' clique.
992         // Continue if we have more members in both cliques. We will
993         // have already broken out if it is not possible for this
994         // second clique to be a subset of the first.
995         while ((i < newClique->size) && (j < curr->set->size))
996         {
997             // If the current member of the first clique is
998             // less than the current member of the second clique
999             // then it is still possible that the first is a
1000             // superset of the second, so move on to the next
1001             // member.
1002             if (newClique->members[i] < curr->set->members[j])
1003             {
1004                 i++;
1005             }
1006             // If the current member of the first clique is greater
1007             // than the current member of the second clique, then
1008             // the proposed second clique cannot be a subset since
1009             // its members are all in ascending order. We also
1010             // know that since the first clique already has
1011             // a subset in this linked list, the current node
1012             // cannot possibly be a superset of the proposed
1013             // clique, so we can just disregard that. Thus,
1014             // we make a flag signifying this and break out.
1015             else if (newClique->members[i] > curr->set->members[j])
1016             {
1017                 breakFlag = 1;
1018                 break;
1019             }
1020             else
1021             {
```

```

1022         i++;
1023         j++;
1024     }
1025 }
1026 // If the breakflag is 1, then we know
1027 // that there is a member of the second clique not in the
1028 // first, and so the second is not a subset. If the breakflag
1029 // is 0 but j is less than the second clique's size, then
1030 // we must have broken because we ran out of members in the
1031 // first clique... thus, there is a member of the second
1032 // clique not in the first. Thus, only if the breakflag is
1033 // 0 and j is equal to the size of the second clique do we
1034 // know that every member of the second clique is in the first
1035 // and that the second clique can thus be removed.
1036 if ((breakFlag == 0) && (j == curr->set->size))
1037 {
1038     // Make the previous clique point to the next one
1039     // instead of the current one.
1040     prev->next = curr->next;
1041     // Free all of the memory used by the current clique.
1042     free (curr->set->members);
1043     free (curr->set);
1044     free (curr);
1045     curr = prev->next;
1046 }
1047 else
1048 {
1049     // Otherwise, the current second clique is not a
1050     // subset of the first, and we advance the prev and
1051     // curr pointers.
1052     prev = curr;
1053     curr = curr->next;
1054 }
1055 }
1056 return head;
1057 }

```

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().

```

613 {
614     int i = 0;
615     emptySet (queue);
616
617     // Go through each offset in the list
618     for (i = 0; i < listszie; i++)
619     {
620         // Check to make sure that's a valid offset number, and if so

```

```

621      // then set its stack true in the queue.
622      if (list[i] + 1 < numOffsets)
623      {
624          queue = setStackTrue (memList, list[i] + 1, queue);
625      }
626      else
627      {
628          fprintf (stderr, "\nInvalid offset number! - searchMemsWithList\n");
629          fprintf (stderr, "\nlist[i]+1 (%d) >= numOffsets (%d)\n",
630                  list[i] + 1, numOffsets);
631          fflush (stderr);
632          exit (0);
633      }
634  }
635
636  return queue;
637 }
```

B.4.0.70 **bitSet_t*** *setStackTrue* (**mll_t** ** *memList*, **int** *i*, **bitSet_t** * *queue*)

Adds all of the members of a given stack to a "queue" in the form of a **bitSet_t** data structure. That is, for each clique in the member linked list, it sets the corresponding bit in the **bitSet_t** true. Input: array of member linked lists, an integer indicating a specific member linked list, and a **bitSet_t** of length \geq the number of cliques in the current clique linked list. Ouput: the updated **bitSet_t** object.

Definition at line 585 of file convll.c.

References mnode::cliqueMembership, mnode::next, and setTrue().

Referenced by searchMemsWithList().

```

586 {
587     mll_t *curr = memList[i];
588
589     // Traverse down the member linked list
590     while (curr != NULL)
591     {
592         // Set the bit in queue corresponding to the current clique
593         // membership true
594         setTrue (queue, curr->cliqueMembership);
595         curr = curr->next;
596     }
597
598     return queue;
599 }
```

B.4.0.71 **cll_t*** *singleCliqueConv* (**cll_t** * *head*, **int** *firstClique*, **cll_t** ** *firstGuess*, **int** *secondClique*, **cll_t** ** *secondGuess*, **cll_t** * *nextPhase*, **bitSet_t** * *printStatus*, **int** *support*)

Convolves one single clique against one other single clique. Note that this is non-commutative, so exchanging *firstClique* and *secondClique* will not give the same results. The "guess" pointers keep the location of the previous clique in the linked list so that we don't have to search the

linked list from the beginning/end every time. We exploit our earlier tidiness in that we can reasonably guess that we will monotonically traverse down cliques. Input: head of the current clique linked list, the id number of the first clique, a pointer to a guess at the first clique, the id number of the second clique, a pointer to a guess at the second clique, the head of the clique linked list for the next round of convolution, a bitSet indicating which cliques should be output as maximal, and the minimum support flag. Output: the head of clique linked list for the next round of convolution (which may have changed if the two cliques could be convolved).

Definition at line 657 of file convll.c.

References cnode::id, mergeIntersect(), cnode::next, popWholeMemStack(), pushConvClique(), cnode::set, setFalse(), and cSet_t::size.

Referenced by wholeCliqueConv().

```

660 {
661     cll_t *first = NULL, *second = NULL;
662     mll_t *survivingMems = NULL;
663     // int flag = 0;
664     int newSupport = 0;
665     // cll_t *checker = head;
666
667     // Check to make sure we're looking for legitimate cliques.
668     if ((firstClique > head->id) || (secondClique > head->id))
669     {
670         fprintf (stderr, "\nNonexistent clique! - singleCliqueConv\n");
671         fflush (stderr);
672         exit (0);
673     }
674     // Our guesses depend on monotonic traversal. If we don't find
675     // the first clique, then bail out.
676     while ((*firstGuess)->id != firstClique)
677     {
678         if ((*firstGuess)->next != NULL)
679         {
680             *firstGuess = (*firstGuess)->next;
681         }
682         else
683         {
684             fprintf (stderr, "\nFirst clique not found! - singleCliqueConv\n");
685             fflush (stderr);
686             exit (0);
687         }
688     }
689     first = *firstGuess;
690
691     // Our guesses depend on monotonic traversal. If we don't find
692     // the second clique, then bail out.
693     while ((*secondGuess)->id != secondClique)
694     {
695         if ((*secondGuess)->next != NULL)
696         {
697             (*secondGuess) = (*secondGuess)->next;
698         }
699         else
700         {
701             fprintf (stderr, "\nSecond clique not found! - singleCliqueConv\n");
702             fflush (stderr);
703             exit (0);
704         }
705     }
706     second = *secondGuess;
707     // Find out what the surviving members are when the first clique

```

```

708 // is convolved with the second clique
709 survivingMems =
710     mergeIntersect (first, second, survivingMems, printStatus, &newSupport);
711
712 // If the first clique is subsumed by the second, then it is not
713 // maximal, so don't print it.
714 // printStatus true means print it!
715 if (newSupport == first->set->size)
716 {
717     setFalse (printStatus, first->id);
718 }
719 // If the second clique is subsumed by the first, then it is not
720 // maximal, so don't print it.
721 if (newSupport == second->set->size)
722 {
723     setFalse (printStatus, second->id);
724 }
725
726 // If the support of the clique just formed by convolution meets the
727 // support criterion, then push it on to the linked list for
728 // the next phase of convolution.
729 if (newSupport >= support)
730 {
731     // printf("Push %d and %d\n",first->id,second->id);
732     nextPhase = pushConvClique (survivingMems, nextPhase);
733     // printf("-----\n");
734     // printCll(nextPhase);
735     // printf("-----\n");
736 }
737 // Pop the surviving members; they are no longer needed, as they
738 // either didn't meet the support criterion or have been pushed on
739 // already
740 survivingMems = popWholeMemStack (survivingMems);
741
742 return nextPhase;
743 }

```

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;

```

```

916     }
917     curr = curr->next;
918 }
919
920 // If we can't find it, then we get upset and exit.
921 if (foundflag == 0)
922 {
923     fprintf (stderr, "\nClique not found! (in swapNode)\n");
924     fflush (stderr);
925     exit (0);
926 }
927 // Then we free the useless clique's members and its set data structure
928 // before pointing its set to the new clique.
929 free (curr->set->members);
930 free (curr->set);
931 curr->set = newClique;
932 return head;
933
934 }

```

B.4.0.73 int uniqClique (*cSet_t * cliquecSet, cll_t * head*)

Before we push a convolved clique onto the stack for the next level, this function ensures that it is not subsumed by and does not subsume any other clique currently on that stack. Input: a candidate clique for the next level in *cSet_t* form, and the head of the clique linked list for the next level. Output: an integer indicating the status of the proposed clique with respect to the next level: -1 if the clique is unique, -2 if the clique is a subset/duplicate of an existing clique, or a clique id in the range [0,numcliques) representing the first clique of which the proposed one is a superset. Note that by executing this each time a clique is added to the next level, we ensure that if the new clique is not unique, it can only be a superset or a subset of some other clique; it cannot be both a strictly superset of one and a strictly subset of another. One of those other two cliques would have been identified in previous steps as being super- or sub-sets, so it is impossible for one clique now to be both a super and a subset.

Definition at line 821 of file convll.c.

References *cnode::id*, *cSet_t::members*, *cnode::next*, *cnode::set*, and *cSet_t::size*.

Referenced by *pushConvClique()*.

```

822 {
823     int i = 0, j = 0;
824     int asubbflag = 1, bsubaflag = 1;
825
826     // Descend through all members of the next level's linked list
827     while (head != NULL)
828     {
829         asubbflag = 1;
830         bsubaflag = 1;
831         i = 0;
832         j = 0;
833         // The proposed convolved clique will be referred to as the
834         // "first" clique, and the current clique being analyzed
835         // in the next level is the "second" clique.
836         // Continue if we have more members in both cliques AND if it
837         // is still possible for one clique to be a subset of
838         // the other.
839         while ((i < cliquecSet->size) && (j < head->set->size) &&

```

```

840      ((asubbflag == 1) || (bsubaflag == 1)))
841  {
842      // If the current member of the first clique is less
843      // than the current member of the second clique,
844      // it is impossible for the first clique to be a
845      // subset of the second (since the members are
846      // traversed in ascending order.
847      if (cliquecSet->members[i] < head->set->members[j])
848      {
849          i++;
850          asubbflag = 0;
851      }
852      // Similarly, if the current member of the second
853      // clique is less than the current member of the
854      // second clique, the second can't be a subset
855      // of the first.
856      else if (cliquecSet->members[i] > head->set->members[j])
857      {
858          j++;
859          bsubaflag = 0;
860      }
861      // Otherwise, they matched this time, so move them
862      // both on.
863      else
864      {
865          i++;
866          j++;
867      }
868  }
869
870      // If the proposed clique is a subset of some other clique
871      // in the next level, then return -2, and it won't be added.
872      // (Note, this also is how exact duplicates are handled.)
873      if ((asubbflag == 1) && (i == cliquecSet->size))
874  {
875      return (-2);
876  }
877      // If the proposed clique is a superset of some other clique(s)
878      // in the next level, then return the id of the first clique
879      // of which it is a superset.
880      if ((bsubaflag == 1) && (j == head->set->size))
881  {
882      return (head->id);
883  }
884      // If the proposed clique has not been found to be a superset
885      // or a subset yet, then move on to the next clique in
886      // the next level.
887      head = head->next;
888  }
889      // If we've gotten here, we've checked all cliques in the previous
890      // level and haven't found the proposed clique to be a superset or
891      // a subset... if so, then we're all good, so return a -1.
892      return (-1);
893 }

```

B.4.0.74 ***cll_t* wholeCliqueConv (cll_t * head, cll_t * node, cll_t **firstGuess, mll_t **memList, int numOffsets, cll_t * nextPhase, bitSet_t * printStatus, int support)***

Convolves one single clique against all possible cliques that could possibly be convolved. It does not attempt to convolve all other cliques, but prunes that set by first looking at the offsets that are in the clique, then collecting all of the cliques who have members that are one greater than the offsets in this clique, and then convolving those cliques in a sort of "queue" using the

[bitSet_t](#) data structure. Input: the head of the clique linked list for the current level, the current node being convolved against in the linked list, the location of the previous node in the form of a pointer to a "guess", an array of member linked lists, the length of that array, the head of the clique linked list for the next level, a [bitSet_t](#) for the printStatus of maximality, and the support criterion. Output: the head of the (possibly modified) clique linked list for the next level.

Definition at line 1208 of file convll.c.

References [bitSetToCSet\(\)](#), [countSet\(\)](#), [deleteBitSet\(\)](#), [cnode::id](#), [cSet_t::members](#), [newBitSet\(\)](#), [searchMemsWithList\(\)](#), [cnode::set](#), [singleCliqueConv\(\)](#), and [cSet_t::size](#).

Referenced by [wholeRoundConv\(\)](#).

```

1211 {
1212     bitSet_t *queue = NULL;
1213     cSet_t *cliquesToSearch = NULL;
1214     int i = 0;
1215     cll_t **secondGuess = NULL;
1216
1217     // This bitSet will be used to create a "queue" of the different
1218     // cliques that must be convolved against the current primary clique.
1219     // A bitset is used to make it easy to deal with duplicates, where
1220     // multiple clique members' next offsets
1221     // are all members of some other specific clique.
1222     queue = newBitSet (head->id + 1);
1223     queue =
1224         searchMemsWithList (node->set->members, node->set->size, mList,
1225         numOffsets, queue);
1226     // We'll use this "secondGuess" to store where the previous clique
1227     // being convolved was... since we will progressing monotonically
1228     // in descending order, this will save us some time in traversing the
1229     // linked list looking for the clique that we want.
1230     secondGuess = (cll_t **) malloc (sizeof (cll_t *));
1231     if (secondGuess == NULL)
1232     {
1233         fprintf (stderr, "Memory error - wholeCliqueConv\n%s\n",
1234                 strerror (errno));
1235         fflush (stderr);
1236         exit (0);
1237     }
1238     // If the offsets that we are looking for are in no other cliques,
1239     // we can just bail out now.
1240     if (countSet (queue) == 0)
1241     {
1242         deleteBitSet (queue);
1243         return nextPhase;
1244     }
1245     // Otherwise, we start our secondGuess at the head and get going.
1246     *secondGuess = head;
1247
1248     // We change the bitSet to something more useful.
1249     cliquesToSearch = bitSetToCSet (queue);
1250
1251     // Note that we start from the end of the cSet member list so that
1252     // we can convolve the highest-id cliques first, which are at the
1253     // beginning of our stack of cliques.
1254     for (i = cliquesToSearch->size - 1; i >= 0; i--)
1255     {
1256         nextPhase = singleCliqueConv (head, node->id, firstGuess,
1257                                         cliquesToSearch->members[i], secondGuess,
1258                                         nextPhase, printStatus, support);
1259     }
1260
1261     // And then we free everything that we created

```

```

1262     deleteBitSet (queue);
1263     free (cliquesToSearch->members);
1264     free (cliquesToSearch);
1265     free (secondGuess);
1266     return nextPhase;
1267 }
```

B.4.0.75 ***cll_t* wholeRoundConv (cll_t ** head, mll_t ** mList, int numOffsets, int support, int length, cll_t ** allCliques)***

Performs convolution on all cliques in a linked list by repeatedly calling wholeCliqueConv. Input: pointer to the head of a clique linked list for the current level, array of member linked lists, length of that array, minimum support threshold, the current length of motifs, and a pointer to a linked list containing all cliques that will be printed out. Output: the head of the clique linked list for the next level of convolution.

Definition at line 1279 of file convll.c.

References checkBit(), deleteBitSet(), fillSet(), cnode::id, newBitSet(), cnode::next, wholeCliqueConv(), and yankCll().

Referenced by completeConv().

```

1281 {
1282     bitSet_t *printStatus = NULL;
1283     cll_t *curr = *head;
1284     cll_t *prev = NULL;
1285     cll_t *nextPhase = NULL;
1286     cll_t **firstGuess = NULL;
1287
1288     // Create a bitset to keep track of print status for this level.
1289     // It starts off all true, and gets changed to false if the patterns
1290     // are not maximal.
1291     printStatus = newBitSet ((*head)->id + 1);
1292     fillSet (printStatus);
1293     firstGuess = (cll_t **) malloc (sizeof (cll_t *));
1294     if (firstGuess == NULL)
1295     {
1296         fprintf (stderr, "Memory error - wholeRoundConv\n%s\n",
1297                  strerror (errno));
1298         fflush (stderr);
1299         exit (0);
1300     }
1301     // Start off at the head.
1302     *firstGuess = *head;
1303     // Convolve a whole clique at a time, traversing the linked list.
1304     // Note that firstGuess gets altered within the function.
1305     while (curr != NULL)
1306     {
1307         nextPhase =
1308         wholeCliqueConv (*head, curr, firstGuess, mList, numOffsets,
1309                         nextPhase, printStatus, support);
1310         curr = curr->next;
1311     }
1312
1313     // Now go back to the head for printing output
1314     curr = *head;
1315
1316     // printf("\n*****\n");
1317     // printf("Length = %d", length);
```

```

1318 // printf("\n*****\n");
1319
1320 // For each clique that is still 'true' in printStatus and is thus
1321 // maximal, perform some sort of output. Yankcll will pull out the
1322 // clique and save it for printing at a later time.
1323 while (curr != NULL)
1324 {
1325     if (checkBit (printStatus, curr->id))
1326     {
1327         // This is the line that makes the allCliques output.
1328         // Can either printcll, or add to allCliques.
1329         // printClIPattern(curr, length);
1330         yankClI (head, prev, &curr, allCliques, length);
1331     }
1332     else
1333     {
1334         prev = curr;
1335         curr = curr->next;
1336     }
1337 }
1338
1339 // And clean up.
1340 deleteBitSet (printStatus);
1341 free (firstGuess);
1342 return nextPhase;
1343 }

```

B.4.o.76 int yankCll (**cll_t** ** *head*, **cll_t** * *prev*, **cll_t** ** *curr*, **cll_t** ** *allCliques*, int *length*)

Removes a clique from within a linked list in order to save it for later printing. This is done so that the cliques are not printed as they are convolved, but rather after all rounds of convolution are complete. Input: a pointer to the head of the current linked list, the clique prior to the one that is to be yanked (NULL if the clique to be yanked is the head), the clique that is to be yanked, a pointer to the head of the list with all cliques that are to be printed, and the length of the current motif. Output: Nothing is returned beyond a success integer, but it alters the current level `cll_t`, the value of `curr`, and the linked list of all cliques that are to be printed.

Definition at line 1359 of file convl.c.

References `cnode::id`, and `cnode::next`.

Referenced by convolve(), and wholeRoundConv().

```
1361 {
1362     if (*curr == NULL)
1363     {
1364         fprintf (stderr, "\nCan't yank from end of cll!\n");
1365         fflush (stderr);
1366         exit (0);
1367     }
1368 // If we're not on the head, change the previous node's "next".
1369 // If we are on the head, make the new head be our current node's "next".
1370 if (prev != NULL)
1371 {
1372     prev->next = (*curr)->next;
1373 }
1374 else
1375 {
```

```
1376     *head = (*curr)->next;
1377 }
1378
1379 // Change next in curr, then change id and length information in curr
1380 (*curr)->next = *allCliques;
1381
1382 if (*allCliques != NULL)
1383 {
1384     (*curr)->id = (*allCliques)->id + 1;
1385 }
1386 else
1387 {
1388     (*curr)->id = 0;
1389 }
1390
1391 (*curr)->length = length;
1392
1393 *allCliques = *curr;
1394
1395 if (prev != NULL)
1396 {
1397     *curr = prev->next;
1398 }
1399 else
1400 {
1401     *curr = *head;
1402 }
1403 return (1);
1404 }
```

Variable Documentation

B.4.o.77 int cliquecounter = 0

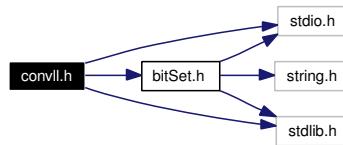
Definition at line 335 of file convll.c.

Referenced by pushClique().

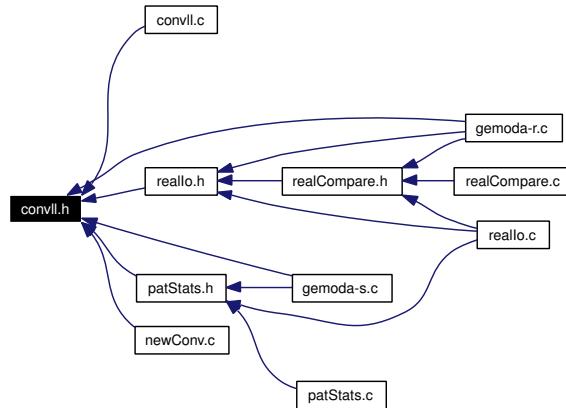
B.5 convll.h File Reference

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

Include dependency graph for convll.h:



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



Data Structures

- struct `cSet_t`
- struct `cnode`
- struct `mnode`

Typedefs

- typedef `cnode cll_t`
- typedef `mnode mll_t`

Functions

- `cll_t * pushClL (cll_t *head)`

- `cll_t * popCll (cll_t *head)`
- `cll_t * popAllCll (cll_t *head)`
- `int printCll (cll_t *head)`
- `cll_t * initheadCll (cll_t *head, cSet_t *newset)`
- `cll_t * pushcSet (cll_t *head, cSet_t *newset)`
- `cll_t * pushClique (bitSet_t *clique, cll_t *head, int *indexToSeq, int p)`
- `mll_t * pushMemStack (mll_t *head, int cliqueNum)`
- `mll_t * popMemStack (mll_t *head)`
- `mll_t * popWholeMemStack (mll_t *head)`
- `mll_t ** addToStacks (cll_t *node, mll_t **memberStacks)`
- `mll_t ** fillMemberStacks (cll_t *head, mll_t **memberStacks)`
- `mll_t ** emptyMemberStacks (mll_t **memberStacks, int size)`
- `void printMemberStacks (mll_t **memberStacks, int size)`
- `bitSet_t * searchMemsWithList (int *list, int listsize, mll_t **memList, int numOffsets, bitSet_t *queue)`
- `bitSet_t * setStackTrue (mll_t **memList, int i, bitSet_t *queue)`
- `cll_t * singleCliqueConv (cll_t *head, int firstClique, cll_t **firstGuess, int secondClique, cll_t **secondGuess, cll_t *nextPhase, bitSet_t *printStatus, int support)`
- `mll_t * mergeIntersect (cll_t *first, cll_t *second, mll_t *intersection, bitSet_t *printStatus, int *newSupport)`
- `cll_t * pushConvClique (mll_t *clique, cll_t *head)`
- `cSet_t * mllToCSet (mll_t *clique)`
- `cSet_t * bitSetToCSet (bitSet_t *clique)`
- `cll_t * wholeCliqueConv (cll_t *head, cll_t *node, cll_t **firstGuess, mll_t **memList, int numOffsets, cll_t *nextPhase, bitSet_t *printStatus, int support)`
- `cll_t * wholeRoundConv (cll_t **head, mll_t **memList, int numOffsets, int support, int length, cll_t **allCliques)`
- `cll_t * completeConv (cll_t **head, int support, int numOffsets, int minLength, int *indexToSeq, int p)`
- `int printCllPattern (cll_t *node, int length)`
- `int uniqClique (cSet_t *clique, cll_t *head)`
- `cll_t * swapNodecSet (cll_t *head, int node, cSet_t *newClique)`
- `int yankCll (cll_t **head, cll_t *prev, cll_t **curr, cll_t **allCliques, int length)`
- `cll_t * removeSupers (cll_t *head, int node, cSet_t *newClique)`

Detailed Description

This header file contains declarations and definitions for dealing with different kinds of sets that are used throughout the convolution stage of Gemoda.

Definition in file `convll.h`.

Typedef Documentation

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.o.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.o.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.o.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.o.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 `mlt_t* popMemStack (mlt_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 `mlt_t* popWholeMemStack (mlt_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 `int printCll (cll_t * head)`

Prints the members (cliques) of a linked list in the format: *id* = unique id number of clique within linked list; *Length* = number of members of clique, if available; *Size* = length of each member of clique; *Members* = newline-separated list of members of the clique. Input: head of a linked list. Output: Gives text output, returns (meaningless) exit value.

Definition at line 118 of file convll.c.

References cnode::id, cnode::length, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

B.5.0.93 `int printCllPattern (cll_t * node, int length)`

Prints out the contents of a clique linked list node in this format: *support* = number of motif occurrences (*id* = some id number); *members* = newline-separated list of offsets. Input: a specific node to be output, the length of the motif inside it. Output: text per above, and an integer success value.

Definition at line 1328 of file convll.c.

References cnode::id, cSet_t::members, cnode::set, and cSet_t::size.

B.5.0.94 `void printMemberStacks (mlt_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.o.99 `mll_t*` `pushMemStack` (`mll_t * head`, `int cliqueNum`)

This begins code for the member linked lists. A single one of these linked lists functions somewhat similarly to the clique linked lists, though with less information stored. Functionally, an array of member linked lists is used to access the "inverse" of what is contained in the clique linked lists. That is, we would like to be able to look up the cliques that a given node is a member of, so we have an array of member linked lists of size equal to the number of nodes.

This function pushes a single clique membership onto a node's member stack. Input: the head of a single member linked list, a clique number to be added. Output: the head of a single member linked list.

Definition at line 358 of file convll.c.

References mnode::cliqueMembership, and mnode::next.

Referenced by addToStacks(), and mergeIntersect().

B.5.o.100 `cll_t*` `removeSupers` (`cll_t * head`, `int node`, `cSet_t * newClique`)

This function finds all cliques in a linked list of which the proposed clique is a superset. It starts looking AFTER the first clique which has already been found to be a subset. In some senses, it is just a continuation of the uniqclique function in order to take advantage of the fact that though a proposed clique can only be a subset of one existing next-level clique, it can be a superset of many existing next- level cliques. Input: head of a clique linked list, the id of the first node found to be a subset of the proposed clique, and the proposed clique (in `cSet_t` form). Output: the head of the clique linked list with all but the first subset (which was passed as an argument) removed. This function is now ready for swapNode to be called.

Definition at line 849 of file convll.c.

References cnode::id, `cSet_t::members`, cnode::next, cnode::set, and `cSet_t::size`.

Referenced by pushConvClique().

B.5.o.101 `bitSet_t*` `searchMemsWithList` (`int * list`, `int listsize`, `mll_t ** memList`, `int numOffsets`, `bitSet_t * queue`)

Creates one large queue by calling "setStackTrue" for each member of a list of offsets. This then creates the union of clique membership for all offsets in the list being searched. Input: an array of offset numbers, the length of that array, an array of member linked lists, the length of that array (the total number of offsets), and a `bitSet_t` to store the union/queue. Output: the union/queue in a `bitSet_t` structure.

Definition at line 540 of file convll.c.

References emptySet(), and setStackTrue().

Referenced by wholeCliqueConv().

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 \geq the number of cliques in the current clique linked list. Ouput: the updated `bitSet_t` object.

Definition at line 516 of file convll.c.

References mnode::cliqueMembership, mnode::next, and setTrue().

Referenced by searchMemsWithList().

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 * cliquecSet, cll_t * head*)

Before we push a convolved clique onto the stack for the next level, this function ensures that it is not subsumed by and does not subsume any other clique currently on that stack. Input: a candidate clique for the next level in *cSet_t* form, and the head of the clique linked list for the next level. Output: an integer indicating the status of the proposed clique with respect to the next level: -1 if the clique is unique, -2 if the clique is a subset/duplicate of an existing clique, or a clique id in the range [0,numcliques) representing the first clique of which the proposed one is a superset. Note that by executing this each time a clique is added to the next level, we ensure that if the new clique is not unique, it can only be a superset or a subset of some other clique; it cannot be both a strictly superset of one and a strictly subset of another. One of those other two cliques would have been identified in previous steps as being super- or sub-sets, so it is impossible for one clique now to be both a super and a subset.

Definition at line 729 of file convll.c.

References cnode::id, cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by pushConvClique().

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.

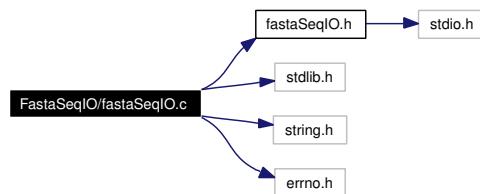
References cnode::id, and cnode::next.

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.o.109 #define BIG_BUFFER 1000000

Definition at line 11 of file fastaSeqIO.c.

B.6.o.110 #define BUFFER 100000

Definition at line 10 of file fastaSeqIO.c.

Function Documentation

B.6.o.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.o.112 long countLines (FILE * INPUT)

Definition at line 69 of file fastaSeqIO.c.

Referenced by ReadFile().

```

70 {
71     long start;
72     long count = 1;
73     int myChar;
74     int status = 0;
75     start = ftell(INPUT);
76     myChar = fgetc(INPUT);
77     while (myChar != EOF) {
78         if (myChar == '\n') {
79             count++;
```

```

80         status = 1;
81     } else {
82         status = 0;
83     }
84     myChar = fgetc(INPUT);
85 }
86 if (status == 1) {
87     count--;
88 }
89 fseek(INPUT, start, SEEK_SET);
90 return count;
91 }

```

B.6.o.113 int FreeFSeqs ([fSeq_t](#) * *arrayOfSequences*, int *numberOfSequences*)

Definition at line 304 of file fastaSeqIO.c.

References [fSeq_t::label](#), and [fSeq_t::seq](#).

Referenced by [main\(\)](#).

```

305 {
306     int i;
307     for (i = 0; i < numberOfSequences; i++) {
308         if (arrayOfSequences[i].label != NULL) {
309             free(arrayOfSequences[i].label);
310         }
311         arrayOfSequences[i].label = NULL;
312
313         if (arrayOfSequences[i].seq != NULL) {
314             free(arrayOfSequences[i].seq);
315         }
316         arrayOfSequences[i].seq = NULL;
317     }
318     if (arrayOfSequences != NULL) {
319         free(arrayOfSequences);
320     }
321     arrayOfSequences = NULL;
322     return EXIT_SUCCESS;
323 }

```

B.6.o.114 int initAoffFSeqs ([fSeq_t](#) * , int *numSeq*)

Definition at line 94 of file fastaSeqIO.c.

References [fSeq_t::label](#), and [fSeq_t::seq](#).

Referenced by [ReadFSeqs\(\)](#), and [ReadTxtSeqs\(\)](#).

```

95 {
96     int i;
97     for (i = 0; i < numSeq; i++) {
98         aos[i].seq = NULL;
99         aos[i].label = NULL;
100    }
101    return 1;
102 }

```

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!\n\n");
115         fflush(stderr);
116         return NULL;
117     }
118     buf = (char **) malloc ( (int)(nl+1) * sizeof(char *) );
```

```

119     if ( buf == NULL){
120         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
121         fflush(stderr);
122         exit(0);
123     }
124
125     // measure the first line
126     tls = measureLine(INPUT) + 1;
127     if(tls != 0){
128         buf[i] = (char *) malloc ( tls * sizeof(char));
129         if ( buf[i] == NULL){
130             fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
131             fflush(stderr);
132             exit(0);
133         }
134     }
135     fgets(buf[i], tls, INPUT);
136     do{
137         if(buf[i][ strlen(buf[i])-1 ] == '\n'){
138             buf[i][ strlen(buf[i])-1 ] = '\0';
139         }
140         tls = measureLine(INPUT) + 1;
141         if(tls != 0){
142             i++;
143             buf[i] = (char *) malloc ( tls * sizeof(char) );
144             if ( buf[i] == NULL){
145                 fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
146                 fflush(stderr);
147                 exit(0);
148             }
149         }
150     }while( fgets(buf[i], tls, INPUT) != NULL );
151     free(buf[i]);
152     buf = (char **) realloc ( buf, i * sizeof(char *));
153     if ( buf == NULL){
154         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
155         fflush(stderr);
156         return NULL;
157     }
158     // I think that 'i' might actually be the # of lines
159     // plus one here? somehow line 131 isn't being freed,
160     // or at least 2 bytes of it.
161     *n = i;
162     return buf;
163 }

```

B.6.o.118 [fSeq_t*](#) ReadFSeqs (FILE * *INPUT*, int * *numberOfSequences*)

Definition at line 199 of file fastaSeqIO.c.

References initAofFSeqs(), fSeq_t::label, ReadFile(), fSeq_t::seq, sSize_t::size, sSize_t::start, and sSize_t::stop.

Referenced by main().

```

199
200     int i,j,k;
201     int nl, ns=0;
202     char **buf = NULL;
203     fSeq_t *aos;
204     sSize_t *ss;
205     sSize_t *ll;

```

```

206
207     buf = ReadFile(INPUT, &nl);
208     if(buf == NULL){
209         return NULL;
210     }
211
212     // Count how many sequences we have
213     for( j=0 ; j<nl ; j++){
214         if(buf[j][0] == '>'){
215             ns++;
216         }
217     }
218     ss = (sSize_t *) malloc ( ns * sizeof(sSize_t) );
219     if(ss == NULL){
220         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
221         fflush(stderr);
222         exit(0);
223     }
224     ll = (sSize_t *) malloc ( ns * sizeof(sSize_t) );
225     if(ll == NULL){
226         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
227         fflush(stderr);
228         exit(0);
229     }
230
231     // find the first sequence
232     k=0;
233     while( buf[k][0] != '>' ){
234         k++;
235     }
236
237     // record how large each sequence is
238     i = -1;
239     for( j=k ; j<nl ; j++){
240         if(buf[j][0] == '>'){
241             i++;
242             ll[i].start = j;
243             ll[i].stop = j;
244             ll[i].size = strlen( buf[j] );
245             ss[i].start = j+1;
246             ss[i].size = 0;
247         }else{
248             ss[i].stop = j;
249             ss[i].size += strlen( buf[j] );
250         }
251     }
252
253     aos = (fSeq_t *) malloc ( ns * sizeof(fSeq_t));
254     if( aos == NULL){
255         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
256         fflush(stderr);
257         exit(0);
258     }
259     initAoffSeqs(aos, ns);
260
261     for ( i=0 ; i<ns ; i++ ){
262         if( ll[i].size > 0 ){
263             aos[i].label = (char *) malloc ( (ll[i].size+1) * sizeof(char) );
264             if( aos[i].label == NULL){
265                 fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
266                 fflush(stderr);
267                 exit(0);
268             }
269             aos[i].label[0] = '\0';
270             for ( j=ll[i].start ; j<=ll[i].stop ; j++ ){
271                 // both instances of strcat here are using
272                 // .label/.seq's that are NULL and that is

```

```

274             // throwing a memory error in valgrind
275             aos[i].label = strcat ( aos[i].label, buf[j] );
276         }
277     }
278     if( ss[i].size > 0 ){
279         aos[i].seq = (char *) malloc ( (ss[i].size+1) * sizeof(char) );
280         if( aos[i].seq == NULL){
281             fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
282             fflush(stderr);
283             exit(0);
284         }
285         aos[i].seq[0] = '\0';
286         for ( j=ss[i].start ; j<=ss[i].stop ; j++ ){
287             aos[i].seq = strcat ( aos[i].seq, buf[j] );
288         }
289     }
290     free(ll);
291     free(ss);
293     for ( i=0 ; i<nl ; i++ ){
295         free(buf[i]);
296     }
297     free(buf);
298
299     *numberOfSequences = ns;
300     return aos;
301 }
```

B.6.o.119 **fSeq_t*** ReadTxtSeqs (FILE * *INPUT*, int * *numberOfSequences*)

Definition at line 172 of file fastaSeqIO.c.

References initAofFSeqs(), ReadFile(), and fSeq_t::seq.

```

172
173     int i;
174     int nl;
175     char **buf = NULL;
176     fSeq_t *aos;
177
178     buf = ReadFile(INPUT, &nl);
179     if(buf == NULL){
180         return NULL;
181     }
182     aos = (fSeq_t *) malloc ( nl * sizeof(fSeq_t));
183     if( aos == NULL){
184         fprintf(stderr, "\nMemory Error\n%s\n", strerror(errno));
185         fflush(stderr);
186         exit(0);
187     }
188     initAofFSeqs(aos, nl);
189     for ( i=0 ; i<nl ; i++ ){
190         aos[i].seq = buf[i];
191     }
192     free(buf);
193     *numberOfSequences = nl;
194     return (aos);
195 }
```

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 }
```

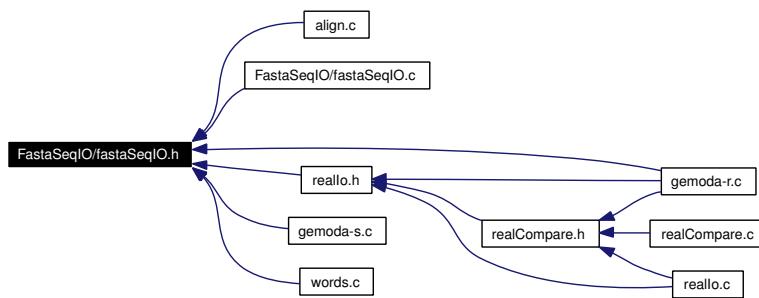
B.7 FastaSeqIO/fastaSeqIO.h File Reference

```
#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 initAoffFSeqs (*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 initAoffFSeqs(), 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 initAoffFSeqs(), ReadFile(), and fSeq_t::seq.

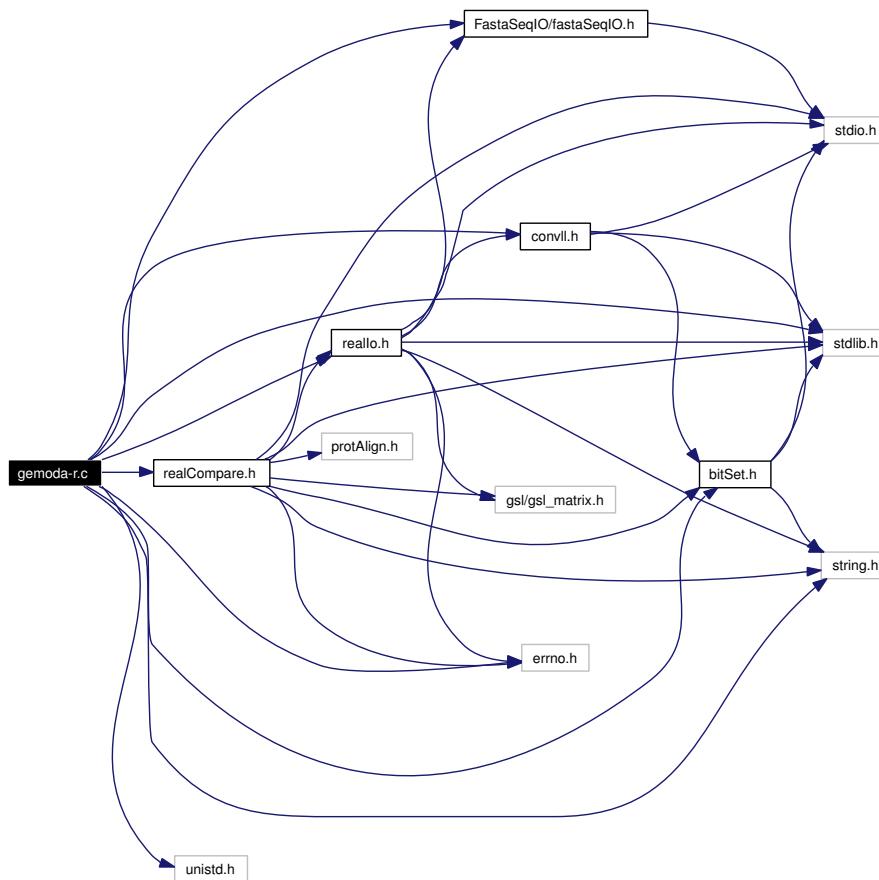
B.7.0.129 `int WriteFSeqA (FILE * MY_FILE, fSeq_t * arrayOfSequences, int start, int stop)`

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)

- `cll_t * convolve (bitGraph_t *bg, int support, int R, int *indexToSeq, int p, int clusterMethod, int **offsetToIndex, int numberOfSequences, int noConvolve, FILE *OUTPUT_FILE)`
- `bitGraph_t * pruneBitGraph (bitGraph_t *bg, int *indexToSeq, int **offsetToIndex, int numSeqs, int p)`
- `int countExtraParams (char *s)`
- `double * parseExtraParams (char *s, int numParams)`
- `int main (int argc, char **argv)`

Detailed Description

This file contains the main routine for the real valued version of Gemoda. There are also some accessory functions for printing information on how to use Gemoda and run it from the commandline.

Definition in file [gemoda-r.c](#).

Function Documentation

B.8.o.130 `cll_t* convolve (bitGraph_t * bg, int support, int R, int * indexToSeq, int p, int clusterMethod, int ** offsetToIndex, int numberOfSequences, int noConvolve, FILE * OUTPUT_FILE)`

Our outer convolution function. This function will call preliminary functions, cluster the data, and then call the main convolution function. This is the interface between the main gemoda-<x> code and the generic code that gets all of the work done. Input: the bitGraph to be clustered and convolved, the minimum support necessary for a motif to be returned, a flag indicating whether recursive filtering should be used, a pointer to the data structure that dereferences offset indices to sequence numbers, the number of unique source sequences that a motif must be present in, and a number indicating the clustering method that is to be used. Output: the final motif linked list with all motifs that are to be given as output to the user.

Definition at line 625 of file newConv.c.

Referenced by main().

```

629 {
630   bitSet_t * cand = NULL;
631   bitSet_t * mask = NULL;
632   bitSet_t * Q = NULL;
633   int size = bg->size;
634   cll_t * elemPats = NULL;
635   cll_t * allCliques = NULL;
636   cll_t * curr = NULL;
637
638   // contains indices (rows) containing the threshold value.
639   cand = newBitSet (size);
640   mask = newBitSet (size);
641   Q = newBitSet (size);
642   fillSet (cand);

```

```

643     fillSet (mask);
644
645     // Note that we prune based on p before setting the diagonal false.
646     if (p > 1)
647     {
648         bg =
649         pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
650     }
651
652     // Now we set the main diagonal false for clustering and filtering.
653     bitGraphSetFalseDiagonal (bg);
654     filterGraph (bg, support, R);
655     fprintf (OUTPUT_FILE, "Graph filtered!  Now clustering...\n");
656     fflush (NULL);
657     if (clusterMethod == 0)
658     {
659         findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
660     }
661     else
662     {
663         singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
664                         p);
665     }
666     fprintf (OUTPUT_FILE,
667             "Clusters found!  Now filtering clusters (if option set)...\\n");
668     fflush (NULL);
669     if (p > 1)
670     {
671         elemPats = pruneCll (elemPats, indexToSeq, p);
672     }
673     deleteBitSet (cand);
674     deleteBitSet (mask);
675     deleteBitSet (Q);
676
677     // Now let's convolve what we made.
678     if (noConvolve == 0)
679     {
680         fprintf (OUTPUT_FILE, "Now convolving...\n");
681         fflush (NULL);
682         allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
683     }
684
685     else
686     {
687         curr = elemPats;
688         while (curr != NULL)
689     {
690         yankCll (&elemPats, NULL, &curr, &allCliques, 0);
691     }
692     }
693     return allCliques;
694 }
```

B.8.o.131 int countExtraParams (char * s)

Definition at line 91 of file gemoda-r.c.

Referenced by main().

```

92 {
93     int i = 0;
94     int numParams = 1;
95     for (i = 0; i < strlen (s); i++)
```

```

96      {
97          if (s[i] == ',')
98          {
99              numParams++;
100         }
101     }
102     return numParams;
103 }
```

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(), popAllCll(), readRealData(), realComparison(), bitGraph_t::size, rdh_t::size, sortByStats(), and usage().

```

161 {
162     int inputOption = 0;
163     char *sequenceFile = NULL;
164     FILE *SEQUENCE_FILE = NULL;
165     char *outputFile = NULL;
166     FILE *OUTPUT_FILE = NULL;
167     int L = 0;
168     int status = 0;
169     double g = 0;
170     int sup = 2;
171     int R = 1;
172     int P = 0;
173     int compFunc = 0;
174     double *extraParams = NULL;
175     int numExtraParams = 0;
176     int i = 0, j = 0;
177     /*
178         int j, k, i, l;
179     */
180     int noConvolve = 0;
181     int samp = 1;
182     int supportDim = 0, lengthDim = 0;
183     bitGraph_t *oam = NULL;
184     unsigned int **d = NULL;
185     int oamSize = 0;
186     cll_t *allCliques = NULL;
187     /*
188         cll_t *curCliq = NULL;
189     */
190     /*
191         int curSeq;
192     */
193     /*
194     */
```

```

195     int curPos;
196     */
197     int clusterMethod = 0;
198     int joelOutput = 0;
199
200     // gemoda-r new stuff
201     rdh_t *data = NULL;
202
203     /*
204      Get command-line options
205     */
206     while ((inputOption = getopt (argc, argv, "p:m:e:i:o:l:g:k:c:njs:")) != EOF)
207     {
208         switch (inputOption)
209     {
210         // Comparison metric
211         case 'm':
212             compFunc = atoi (optarg);
213             break;
214         // Input file
215         case 'i':
216             sequenceFile = optarg;
217             break;
218         // Output file
219         case 'o':
220             outputFile =
221                 (char *) malloc ((strlen (optarg) + 1) * sizeof (char));
222             if (outputFile == NULL)
223             {
224                 fprintf (stderr, "Error allocating memory for options.\n");
225                 exit (EXIT_FAILURE);
226             }
227             else
228             {
229                 strcpy (outputFile, optarg);
230             }
231             break;
232         // Minimum motif length
233         case 'l':
234             L = atoi (optarg);
235             break;
236         // Minimum motif similarity score
237         case 'g':
238             g = atof (optarg);
239             status++;
240             break;
241         // Minimum support (number of motif occurrences)
242         case 'k':
243             sup = atoi (optarg);
244             break;
245
246     ****
247     * Recursive initial pruning: an option for clique finding.
248     * It takes all nodes with less than the minimum
249     * number of support and removes all of their nodes, and does this
250     * recursively so that nodes that are connected to many sparsely connected
251     * nodes will be removed and not left in the
252     * This option is deprecated as it is at worst no-gain and at best useful.
253     * It will be on by default for clique-finding, but can be turned
254     * back off with some
255     * minor tweaking. For almost all cases in which it does not speed
256     * up computations, it will have a trivial time to perform. Thus, if
257     * clique-finding is turned on, then R is set to 1 by default.
258         case 'r':
259             R = 1;
260             break;
261     ****
262     // Optional pruning parameter to require at motif occurrences

```

```
263     // in at least P distinct input sequences
264
265     case 'p':
266         P = atoi (optarg);
267         break;
268
269     // Clustering method.
270     case 'c':
271         clusterMethod = atoi (optarg);
272         break;
273     // Extra parameters for comparison function
274     case 'e':
275         numExtraParams = countExtraParams (optarg);
276         extraParams = parseExtraParams (optarg, numExtraParams);
277         break;
278     case 'n':
279         noConvolve = 1;
280         break;
281     case 'j':
282         joelOutput = 1;
283         break;
284     case 's':
285         samp = atoi (optarg);
286         break;
287     // Catch-all.
288     case '?':
289         fprintf (stderr, "Unknown option '-%c'.\n", optopt);
290         usage (argv);
291         return EXIT_SUCCESS;
292     default:
293         usage (argv);
294         return EXIT_SUCCESS;
295     }
296 }
297 // Require an input file, a nonzero length, and a similarity threshold
298 // to be set.
299 if (sequenceFile == NULL || L == 0 || status < 1)
300 {
301     usage (argv);
302     return EXIT_SUCCESS;
303 }
304 // Open the sequence file
305 if ((SEQUENCE_FILE = fopen (sequenceFile, "r")) == NULL)
306 {
307     fprintf (stderr, "Couldn't open file %s; %s\n", sequenceFile,
308             strerror (errno));
309     exit (EXIT_FAILURE);
310 }
311 // Open the output file
312 if (outputFile != NULL)
313 {
314     if ((OUTPUT_FILE = fopen (outputFile, "w")) == NULL)
315     {
316         fprintf (stderr, "Couldn't open file %s; %s\n", outputFile,
317                 strerror (errno));
318         exit (EXIT_FAILURE);
319     }
320 }
321 else
322 {
323     OUTPUT_FILE = stdout;
324 }
325
326
327
328 // Verbosity in output helps to distinguish output files.
329 fprintf (OUTPUT_FILE, "Input file = %s\n", sequenceFile);
330 fprintf (OUTPUT_FILE, "l = %d, k = %d, g = %f\n", L, sup, g);
```

```

331  if (P > 1)
332  {
333      fprintf (OUTPUT_FILE, "Minimum # of sequences with motif = %d\n", P);
334  }
335  if (R > 0)
336  {
337      fprintf (OUTPUT_FILE, "Recursive pruning is ON.\n");
338  }
339
340  data = readRealData (SEQUENCE_FILE);
341  fclose (SEQUENCE_FILE);
342  // printf("size = %d,indexSize = %d\n",data->size,data->indexSize);
343  // printf("size1 = %d,size2 = %d\n",data->seq[0]->size1,data->seq[0]->size2);
344  // for(i = 0; i < 2; i++) {
345  // for(j = 0; j < 3; j++) {
346  // printf("%lf,%lf,%lf\n",gsl_matrix_get(data->seq[i],j,0),
347  // gsl_matrix_get(data->seq[i],j,1),
348  // gsl_matrix_get(data->seq[i],j,2));}
349  oam = realComparison (data, L, g, compFunc, extraParams);
350  // printf("oam->size = %d\n", oam->size);
351  if ((samp > 0) && (clusterMethod == 0))
352  {
353      // We are currently using one gap per sequence, as done in
354      // realCompare.c's call to initRdhIndex in realComparison.
355      // Note that this is data->size, NOT oam->size.
356      d =
357      getStatMat (oam, sup, L, &supportDim, &lengthDim, data->size, samp,
358                  OUTPUT_FILE);
359  }
360  else
361  {
362      d = NULL;
363      supportDim = 0;
364  }
365
366  allCliques =
367  convolve (oam, sup, R, data->indexToSeq, P, clusterMethod,
368             data->offsetToIndex, data->size, noConvolve, OUTPUT_FILE);
369
370  oamSize = oam->size;
371  // Do some early memory cleanup since this is so big.
372  deleteBitGraph (oam);
373
374  if ((samp > 0) && (clusterMethod == 0))
375  {
376      cumDMatrix (d, allCliques, supportDim, lengthDim, oamSize, data->size);
377      calcStatAllCliqs (d, allCliques, oamSize - data->size);
378      allCliques = sortByStats (allCliques);
379  }
380
381  if (joelOutput == 0)
382  {
383      outputRealPats (data, allCliques, L, OUTPUT_FILE, d);
384  }
385  else
386  {
387      outputRealPatsWCentroid (data, allCliques, L, OUTPUT_FILE, extraParams,
388                               compFunc);
389  }
390
391  freeD (d, supportDim);
392  freeRdh (data);
393  allCliques = popAllCll (allCliques);
394  fclose (OUTPUT_FILE);
395
396  return 0;
397 }

```

B.8.o.133 double* parseExtraParams (char * s, int numParams)

This was borrowed from the old gemoda-p code, there it used to parse filenames, here we are parsing comma-separated lists of doubles that are useful for SpecConnect.

Definition at line 110 of file gemoda-r.c.

Referenced by main().

```

111 {
112     int i = 0, j = 0, k = 0;
113     int startLength = 0;
114     double *extraParams = NULL;
115     char *paramString = NULL;
116
117     extraParams = (double *) malloc (numParams * sizeof (double));
118     if (extraParams == NULL)
119     {
120         fprintf (stderr, "Can't allocate extra params!\n");
121         exit (0);
122     }
123     j = 0;
124     k = 0;
125     startLength = strlen (s);
126     for (i = 0; i < startLength; i++)
127     {
128         if (s[i] == ',')
129         {
130             // We've found an end. So point the pointer to
131             // the beginning of the previous string.
132             paramString = &s[k];
133             // Terminate the string where the comma used to be
134             s[i] = '\0';
135             // Update the location for the next string beginning
136             k = i + 1;
137             // Convert to a double and update the param number.
138             extraParams[j] = atof (paramString);
139             j++;
140         }
141     }
142     // Don't forget to do the last one, which isn't comma-terminated.
143     paramString = &s[k];
144     extraParams[j] = atof (paramString);
145     return (extraParams);
146 }
```

**B.8.o.134 bitGraph_t* pruneBitGraph (bitGraph_t * bg, int * indexToSeq, int **
offsetToIndex, int numSeqs, int p)**

Simple function (non-recursive) to prune off the first level of motifs that will not meet the "minimum number of unique sequences" criterion. This could have been implemented as above, but it may have gotten a little expensive with less yield, so only the first run through is done here. Input: a bit graph to be pruned, a pointer to the structure that dereferences offset indices to sequence numbers, a pointer to the structure that dereferences seq/position to offsets, the number of unique sequences in the input set, and the minimum number of unique sequences that must contain the motif. Output: a pruned bitGraph.

Definition at line 402 of file newConv.c.

Referenced by convolve().

```

404 {
405     int i = 0, j = 0, nextBit = 0;
406     int *seqNums = NULL;
407
408     // Since we don't immediately know which node is in which source
409     // sequence, we can't just count them up regularly. Instead, we'll
410     // need to keep track of which sequences they come from and
411     // increment _something_. What we chose to do here is just make
412     // an array of integers of length = <p>. Then, we try to put the
413     // source sequence number of each neighbor (including itself, since
414     // the main diagonal is still true at this time) into the next slot
415     // Since we will monotonically search the bitSet, we can just
416     // move on to the first bit in the next sequence using the
417     // offsetToIndex structure so that we know the next sequence number
418     // to be put in is always unique.
419     seqNums = (int *) malloc (p * sizeof (int));
420     if (seqNums == NULL)
421     {
422         fprintf (stderr, "Memory error - pruneBitGraph\n%s\n",
423                 strerror (errno));
424         fflush (stderr);
425         exit (0);
426     }
427
428     // So, for each row in the bitgraph...
429     for (i = 0; i < bg->size; i++)
430     {
431
432         // Make sure the whole array is -1 sentinels.
433         for (j = 0; j < p; j++)
434         {
435             seqNums[j] = -1;
436         }
437         j = 0;
438
439         // Find the first neighbor of this bit.
440         nextBit = nextBitBitSet (bg->graph[i], 0);
441         if (nextBit == -1)
442         {
443             continue;
444         }
445         else
446         {
447
448             // and put its sequence number in the array of ints.
449             seqNums[0] = indexToSeq[nextBit];
450         }
451
452         // If it's the last sequence, then bail out so that we don't
453         // segfault in the next step.
454         if (seqNums[0] >= numOfSeqs - 1)
455         {
456             emptySet (bg->graph[i]);
457             continue;
458         }
459
460         // Find the next neighbor of this bit, STARTING AT the first
461         // bit in the next sequence.
462         nextBit =
463         nextBitBitSet (bg->graph[i],
464                         offsetToIndex[indexToSeq[nextBit] + 1][0]);
465
466         // And iterate this until we run out of neighbors.
467         while (nextBit >= 0)
468         {

```

```

469     j++;
470     seqNums[j] = indexToSeq[nextBit];
471
472     // Or until this new neighbor will fill up the array
473     if (j == p - 1)
474     {
475         break;
476     }
477
478     // Or until this new neighbor is in the last sequence.
479     if (seqNums[j] >= numSeqs - 1)
480     {
481         break;
482     }
483
484     // Get the next neighbor!
485     nextBit =
486     nextBitBitSet (bg->graph[i],
487                     offsetToIndex[indexToSeq[nextBit] + 1][0]);
488 }
489
490 // If we didn't have enough unique sequences, and either a) we
491 // were in the nth-to-last sequence and there were no
492 // neighbors after it, or b) we were in the last sequence,
493 // then the last number will still be our sentinel, -1. If
494 // the last number is not a sentinel, then we have at least
495 // p distinct sequence occurrences, so we're OK.
496 if (seqNums[p - 1] == -1)
497 {
498     emptySet (bg->graph[i]);
499 }
500 }
501 free (seqNums);
502 return (bg);
503 }
```

B.8.o.135 void usage (char ** *argv*)

This function tells the user how to run Gemoda. The function displays all the available flags and gives an example of how to use the commandline to run the code.

Definition at line 35 of file gemoda-r.c.

Referenced by main().

```

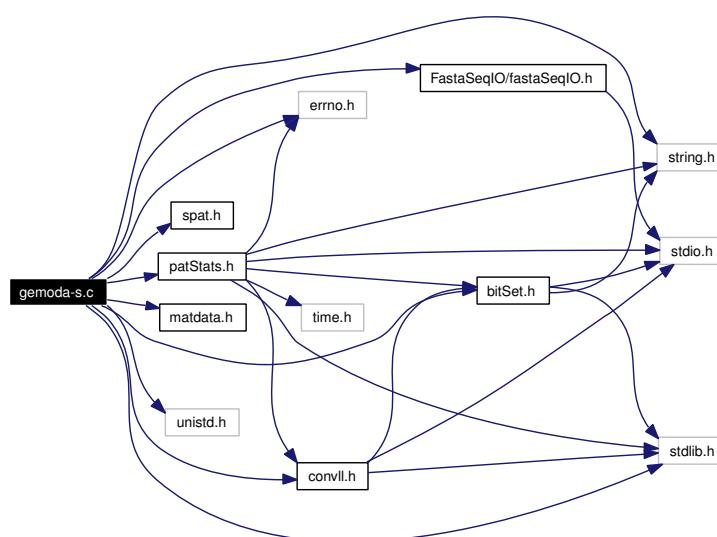
36 {
37     fprintf (stdout,
38             "Usage: %s -i <Fasta sequence file> "
39             "-l <word size> \n\t-k <support> -g <threshold> "
40             "-m <matrix name> [-z] \n\t[-c <cluster method [0|1]>]"
41             "[ -p <unique support>] \n\n"
42             "Required flags and input:\n\n"
43             "-i <Fasta sequence file>:\n\t"
44             "File containing all sequences to be searched, in Fasta format.\n\n"
45             "-l <word size>:\n\t"
46             "Minimum length of motifs; also the sliding window length\n\t"
47             "over which all motifs must meet the similarity criterion\n\n"
48             "-k <support>:\n\t"
49             "Minimum number of motif occurrences.\n\n"
50             "-g <threshold>:\n\t"
51             "Similarity threshold. Two windows, when scored with the\n\t"
52             "similarity matrix defined by the -m flag, must have at least\n\t"
```

```
51      " this score in order to be deemed 'connected'. This criterion\n\t"
52      " must be met over all sliding windows of length 1.\n\n"
53      "-c <cluster method [0|1]>:\n\t"
54      "The clustering method to be used after evaluating the "
55      "\n\ttsimilarity of the unique words in the input. Note that the "
56      "\n\tclustering method will have a significant impact on both the "
57      "\n\tresults that one obtains and the computation time.\n\n\t"
58      "0: clique-finding\n\t"
59      "Uses established methods to find all maximal cliques in the "
60      "\n\ttdata. This will give the most thorough results (that are "
61      "\n\ttprovable exhaustive), but will also give less-significant "
62      "\n\ttresults in addition to the most interesting and most\n\t"
63      "significant ones. The results are deterministic but may take some "
64      "\n\tttime on data sets with high similarity or if the similarity "
65      "\n\ttthreshold is set extremely low.\n\t"
66      "1: single-linkage clustering\n\t"
67      "Uses a single-linkage-type clustering where all nodes that "
68      "\n\ttare connected are put in the same cluster. This method is "
69      "\n\ttalso deterministic and will be faster than clique-finding, "
70      "\n\ttbut it loses guarantees of exhaustiveness in searching the "
71      "\n\ttdata set.\n\n",
72      "-p <unique support>:\n\t"
73      "A pruning parameter that requires the motif to occur in "
74      "\n\ttat least <unique support> different input sequences. Note "
75      "\n\ttthat this parameter must be less than or equal to the total "
76      "\n\ttsupport parameter set by the -k flag.\n\n", argv[0]);
77      fprintf (stdout, "\n");
78 }
```

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:



Functions

- void `usage` (char **argv)
- void `matrixlist` (void)
- void `getMatrixByName` (char name[], int mat[][][MATRIX_SIZE])
- `bitGraph_t` * `alignWordsMat_bit` (`sPat_t` *words, int wc, int mat[][][MATRIX_SIZE], int threshold)

- `sPat_t * countWords2 (fSeq_t *seq, int numSeq, int L, int *numWords)`
- `cll_t * convolve (bitGraph_t *bg, int support, int R, int *indexToSeq, int p, int clusterMethod, int **offsetToIndex, int numberofSequences, int noConvolve, FILE *OUTPUT_FILE)`
- `bitGraph_t * pruneBitGraph (bitGraph_t *bg, int *indexToSeq, int **offsetToIndex, int numofSeqs, int p)`
- int `main (int argc, char **argv)`

Detailed Description

This file houses the main routine for the sequence based Gemoda algorithm. In addition, there are a few helper functions which are used to inform the user how to run the software.

The Gemoda algorithm has three stages: comparison, clustering, and convolution. These three stages are called in serial from the main routine in this file.

Definition in file [gemoda-s.c](#).

Function Documentation

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 and all by all comparison; however, we only do the upper diagonal. The function makes use of alignMat and needs to be passed a scoring matrix that the user has chosen which is appropriate for the context of whatever data sent the user is looking at.

Definition at line 88 of file align.c.

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

Referenced by main().

```

90 {
91     bitGraph_t * sg = NULL;
92     int score;
93     int i, j;
94
95     // Assign a new bitGraph_t object, with (wc x wc) possible
96     // true/false values
97     sg = newBitGraph (wc);
98     for (i = 0; i < wc; i++)
99     {
100         for (j = i; j < wc; j++)
101         {
102             // Get the score for the alignment of word i and word j

```

```

104     score =
105     alignMat (words[i].string, words[j].string, words[i].length, mat);
106
107     // If that score is greater than threshold, set
108     // a bit to 'true' in our bitGraph_t object
109     if (score >= threshold)
110     {
111
112         // We use 'bitGraphSetTrueSym' because, if i=j,
113         // then j=i for most applications. However, this
114         // can be relaxed for masochists.
115         bitGraphSetTrueSym (sg, i, j);
116     }
117 }
118 }
119
120 // Return a pointer to this new bitGraph_t object
121 return sg;
122 }
```

B.9.o.137 *cll_t* convolve (bitGraph_t * bg, int support, int R, int * indexToSeq, int p, int clusterMethod, int ** offsetToIndex, int numberSequences, int noConvolve, FILE * OUTPUT_FILE)*

Our outer convolution function. This function will call preliminary functions, cluster the data, and then call the main convolution function. This is the interface between the main gemoda-<x> code and the generic code that gets all of the work done. Input: the bitGraph to be clustered and convolved, the minimum support necessary for a motif to be returned, a flag indicating whether recursive filtering should be used, a pointer to the data structure that dereferences offset indices to sequence numbers, the number of unique source sequences that a motif must be present in, and a number indicating the clustering method that is to be used. Output: the final motif linked list with all motifs that are to be given as output to the user.

Definition at line 625 of file newConv.c.

References bitGraphSetFalseDiagonal(), completeConv(), deleteBitSet(), fillSet(), filterGraph(), findCliques(), newBitSet(), pruneBitGraph(), pruneCll(), singleLinkage(), bitGraph_t::size, and yankCll().

```

629 {
630     bitSet_t * cand = NULL;
631     bitSet_t * mask = NULL;
632     bitSet_t * Q = NULL;
633     int size = bg->size;
634     cll_t * elemPats = NULL;
635     cll_t * allCliques = NULL;
636     cll_t * curr = NULL;
637
638     // contains indices (rows) containing the threshold value.
639     cand = newBitSet (size);
640     mask = newBitSet (size);
641     Q = newBitSet (size);
642     fillSet (cand);
643     fillSet (mask);
644
645     // Note that we prune based on p before setting the diagonal false.
646     if (p > 1)
```

```

647      {
648        bg =
649        pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
650      }
651
652      // Now we set the main diagonal false for clustering and filtering.
653      bitGraphSetFalseDiagonal (bg);
654      filterGraph (bg, support, R);
655      fprintf (OUTPUT_FILE, "Graph filtered!  Now clustering...\n");
656      fflush (NULL);
657      if (clusterMethod == 0)
658      {
659        findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
660      }
661    else
662    {
663      singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
664                  p);
665    }
666    fprintf (OUTPUT_FILE,
667            "Clusters found!  Now filtering clusters (if option set)...\\n");
668    fflush (NULL);
669    if (p > 1)
670    {
671      elemPats = pruneCll (elemPats, indexToSeq, p);
672    }
673    deleteBitSet (cand);
674    deleteBitSet (mask);
675    deleteBitSet (Q);
676
677    // Now let's convolve what we made.
678    if (noConvolve == 0)
679    {
680      fprintf (OUTPUT_FILE, "Now convolving...\n");
681      fflush (NULL);
682      allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
683    }
684
685  else
686  {
687    curr = elemPats;
688    while (curr != NULL)
689    {
690      yankCll (&elemPats, NULL, &curr, &allCliques, 0);
691    }
692  }
693  return allCliques;
694 }

```

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()**.

```

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     {
390         totalChars += strlen (seq[i].seq);
391     }
392
393     // Get a prime number for the size of the hash table
394     hashSize = sieve3 ((long) (2 * totalChars));
395     wordHash = initSHash (hashSize);
396
397     // Chop up each sequence and hash out the words of size L
398     for (i = 0; i < numSeq; i++)
399     {
400         prev = -1;
401
402         // skip sequences that are too short to have
403         // a pattern
404         if (strlen (seq[i].seq) < L)
405         {
406             continue;
407         }
408         for (j = 0; j < strlen (seq[i].seq) - L + 1; j++)
409         {
410
411             // Make a hash table entry for this word
412             newEntry.key = &(seq[i].seq[j]);
413             newEntry.data = 1;
414             newEntry.idx = wc;
415             newEntry.L = L;
416
417             // Check to see if it's already in the hash table
418             ep = searchSHash (&newEntry, &wordHash, 0);
419             if (ep == NULL)
420             {
421
422                 // If it's not, create an entry for it
423                 ep = searchSHash (&newEntry, &wordHash, 1);
424
425                 // Increase the size of our word array
426                 words = (sPat_t *) realloc (words, (wc + 1) * sizeof (sPat_t));
427                 if (words == NULL)
428                 {
429                     fprintf (stderr, "Error!\n");
430                     fflush (stderr);
431                 }
432
433                 // Add the new word
434                 words[wc].string = &(seq[i].seq[j]);
435                 words[wc].length = L;
436                 words[wc].support = 1;
437                 words[wc].offset =
438                 (sOffset_t *) malloc (1 * sizeof (sOffset_t));
439                 if (words[wc].offset == NULL)
440                 {
441                     fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));

```

```

442         fflush (stderr);
443         exit (0);
444     }
445     words[wc].offset[0].seq = i;
446     words[wc].offset[0].pos = j;
447     words[wc].offset[0].prev = prev;
448     words[wc].offset[0].next = -1;
449
450     if (prev != -1)
451     {
452         words[prev].offset[words[prev].support - 1].next = wc;
453     }
454     prev = wc;
455     wc++;
456
457 }
458 else
459 {
460
461     // If it is, increase the count for this word
462     ep->data++;
463
464     // add a new offset to the word array
465     l = words[ep->idx].support;
466     words[ep->idx].offset =
467     (sOffset_t *) realloc (words[ep->idx].offset,
468                           (l + 1) * sizeof (sOffset_t));
469     words[ep->idx].offset[l].seq = i;
470     words[ep->idx].offset[l].pos = j;
471     words[ep->idx].offset[l].prev = prev;
472     words[ep->idx].offset[l].next = -1;
473
474     // Update the next/prev
475     if (prev != -1)
476     {
477         words[prev].offset[words[prev].support - 1].next = ep->idx;
478     }
479     prev = ep->idx;
480
481     // Have to put this down here for cases when we create
482     // a word and it is immediately followed by itself!!
483     words[ep->idx].support += 1;
484 }
485 }
486 }
487
488
489 destroySHash (&wordHash);
490 *numWords = wc;
491 return words;
492 }
```

B.9.o.139 void getMatrixByName (char *name*[], int *mat*[][MATRIX_SIZE])

Referenced by main().

B.9.o.140 int main (int *argc*, char ** *argv*)

This is the main routine of the Gemoda source code. The routine performs basic operations such as parsing the input from the user and opening input files. Then, the function hashes words of length L. The unique words are aligned against each other to produce an adjacency

matrix that says whether the unique word i is sufficiently similar, based on the user supplied threshold, to the unique word j. This adjacency matrix is then dereferenced into an adjacency matrix in which each index of the matrix represents a unique position in the input sequences, rather than a unique word. This dereferencing is required for the convolution stage. Finally, this adjacency matrix is convolved and the final motifs are returned as a linked list. The routine then closes all input and output files and frees up dynamically allocated memory.

Definition at line 187 of file gemoda-s.c.

References alignWordsMat_bit(), bitGraphCheckBit(), bitGraphSetTrueSym(), calcStatAllCliqs(), convolve(), countWords2(), cumDMatrix(), deleteBitGraph(), FreeFSeqs(), getMatrixByName(), getStatMat(), cnode::length, mat, MATRIX_SIZE, matrixlist(), cSet_t::members, newBitGraph(), cnode::next, sPat_t::offset, popAllCll(), sOffset_t::pos, ReadFSeqs(), sOffset_t::seq, cnode::set, cSet_t::size, bitGraph_t::size, sortByStats(), cnode::stat, and usage().

```

188 {
189     int inputOption = 0;
190     char *sequenceFile = NULL;
191     char *outputFile = NULL;
192     char *matName = NULL;
193     FILE * SEQUENCE_FILE = NULL;
194     FILE * OUTPUT_FILE = NULL;
195     int L = 0;
196     int numberOfSequences = 0;
197     fSeq_t * mySequences = NULL;
198     fSeq_t * (*seqReadFunct) () = &ReadFSeqs;
199     sPat_t * words = NULL;
200     int wc;
201     int status = 0;
202     int g = 0;
203     int sup = 2;
204     int R = 1;
205     int P = 0;
206     int (*mat)[MATRIX_SIZE] = NULL;
207     int noConvolve = 0;
208     int j, k, i, l;
209     bitGraph_t * bg = NULL;
210     bitGraph_t * oam = NULL;
211
212     // new
213     int **offsetToIndex = NULL;
214     int *indexToSeq = NULL;
215     int *indexToPos = NULL;
216     int numberOfOffsets = 0;
217     int pos1, pos2;
218
219     // int *prevRowArray;
220     sOffset_t * offset1, *offset2;
221     cll_t * allCliques = NULL;
222     cll_t * curCliq = NULL;
223     int curSeq;
224     int curPos;
225     int clusterMethod = 0;
226
227     // patStats
228     int samp = 1;
229     unsigned int **d = NULL;
230     int supportDim = 0, lengthDim = 0;
231     int oamSize = 0;
232
233     /*
234         Get command-line options

```

```

235      */
236      while ((inputOption = getopt (argc, argv, "i:o:l:g:k:m:p:zc:ns:")) != EOF)
237      {
238          switch (inputOption)
239          {
240
241              // Input file
242              case 'i':
243                  sequenceFile = optarg;
244                  seqReadFunct = &ReadFSeqs;
245                  break;
246
247              // Output file
248              case 'o':
249                  outputFile =
250                      (char *) malloc ((strlen (optarg) + 1) * sizeof (char));
251                  if (outputFile == NULL)
252                  {
253                      fprintf (stderr, "Error allocating memory for options.\n");
254                      exit (EXIT_FAILURE);
255                  }
256                  else
257                  {
258                      strcpy (outputFile, optarg);
259                  }
260                  break;
261
262              // Minimum motif length
263              case 'l':
264                  L = atoi (optarg);
265                  break;
266
267              // Minimum motif similarity score
268              case 'g':
269                  g = atoi (optarg);
270                  status++;
271                  break;
272
273              // Minimum support (number of motif occurrences)
274              case 'k':
275                  sup = atoi (optarg);
276                  break;
277
278              // Similarity matrix used to find similarity score
279              case 'm':
280                  getMatrixByName (optarg, &mat);
281                  matName = (char *) malloc (strlen (optarg) * sizeof (char));
282                  if (matName == NULL)
283                  {
284                      fprintf (stderr, "Error allocating memory for options.\n");
285                      exit (EXIT_FAILURE);
286                  }
287                  else
288                  {
289                      strcpy (matName, optarg);
290                  }
291                  break;
292
293 /*****
294 * Recursive initial pruning: an option for clique finding.
295 * It takes all nodes with less than the minimum
296 * number of support and removes all of their nodes, and does this
297 * recursively so that nodes that are connected to many sparsely connected
298 * nodes will be removed and not left in the
299 * This option is deprecated as it is at worst no-gain and at best useful.
300 * It will be on by default for clique-finding, but can be turned
301 * back off with some
302 * minor tweaking. For almost all cases in which it does not speed

```

```

303 *      up computations, it will have a trivial time to perform. Thus, if
304 *      clique-finding is turned on, then R is set to 1 by default.
305     case 'r':
306         R = 1;
307         break;
308 ****
309     // Optional pruning parameter to require at motif occurrences
310     // in at least P distinct input sequences
311     case 'p':
312         P = atoi (optarg);
313         break;
314
315     // Clustering method.
316     case 'c':
317         clusterMethod = atoi (optarg);
318         break;
319     case 'n':
320         noConvolve = 1;
321         break;
322     case 's':
323         samp = atoi (optarg);
324         break;
325
326     // Catch-all.
327     case '?':
328         fprintf (stderr, "Unknown option '-%c'.\n", optopt);
329         usage (argv);
330         return EXIT_SUCCESS;
331     case 'z':
332         matrixlist ();
333         return EXIT_SUCCESS;
334     default:
335         usage (argv);
336         return EXIT_SUCCESS;
337     }
338     }
339
340 // Require a similarity matrix
341 if (mat == NULL)
342 {
343     usage (argv);
344     return EXIT_SUCCESS;
345 }
346
347 // Require an input file, a nonzero length, and a similarity threshold
348 // to be set.
349 if (sequenceFile == NULL || L == 0 || status < 1)
350 {
351     usage (argv);
352     return EXIT_SUCCESS;
353 }
354
355 // Open the sequence file
356 if ((SEQUENCE_FILE = fopen (sequenceFile, "r")) == NULL)
357 {
358     fprintf (stderr, "Couldn't open file %s; %s\n", sequenceFile,
359             strerror (errno));
360     exit (EXIT_FAILURE);
361 }
362
363 // Open the output file
364 if (outputFile != NULL)
365 {
366     if ((OUTPUT_FILE = fopen (outputFile, "w")) == NULL)
367     {
368         fprintf (stderr, "Couldn't open file %s; %s\n", outputFile,
369                 strerror (errno));
370         exit (EXIT_FAILURE);

```

```

371     }
372   }
373 else
374 {
375   OUTPUT_FILE = stdout;
376 }
377
378 // Allocate some sequences
379 mySequences = seqReadFunct (SEQUENCE_FILE, &numberOfSequences);
380 if (mySequences == NULL)
381 {
382   fprintf (stderr, "\nError reading your sequences/text.");
383   fprintf (stderr, "\nCheck the format/size of the file.");
384   fprintf (stderr, "\nERROR: %s\n", strerror (errno));
385   return EXIT_FAILURE;
386 }
387
388 // Close the input files
389 fclose (SEQUENCE_FILE);
390
391 // Verbosity in output helps to distinguish output files.
392 fprintf (OUTPUT_FILE, "\nMatrix used = %s\n", matName);
393 fprintf (OUTPUT_FILE, "Input file = %s\n", sequenceFile);
394 fprintf (OUTPUT_FILE, "l = %d, k = %d, g = %d\n", L, sup, g);
395 if (P > 1)
396 {
397   fprintf (OUTPUT_FILE, "Minimum # of sequences with motif = %d\n", P);
398 }
399 if (R > 0)
400 {
401   fprintf (OUTPUT_FILE, "Recursive pruning is ON.\n");
402 }
403
404 // Find the unique words in the input.
405 words = countWords2 (mySequences, numberOfSequences, L, &wc);
406
407 /*
408   fprintf(stderr, "Counted %d words\n", wc);
409 */
410 /*
411   fflush(stderr);
412 */
413
414 // Align the words that we just found by applying the similarity
415 // matrix to each pair of them. Note that
416 // bg is the adjacency matrix of words, but we
417 // need an adjacency matrix of offsets instead.
418 bg = alignWordsMat_bit (words, wc, mat, g);
419 fprintf (OUTPUT_FILE, "\nAligned! Creating offset matrix...\n");
420 fflush (NULL);
421
422 // Create an intermediate translation matrix
423 // to store the offset number of each sequence number/position.
424 //
425 // Note that this matrix is better called "Index to offset", and
426 // the other matrices are better called "offset to Seq" and
427 // "offset to Pos"
428 offsetToIndex = (int **) malloc (numberOfSequences * sizeof (int *));
429 if (offsetToIndex == NULL)
430 {
431   fprintf (stderr,
432     "Unable to allocate memory - offsetToIndex in gemoda.c\n%s\n",
433     strerror (errno));
434   fflush (stderr);
435   exit (0);
436 }
437 for (i = 0; i < numberOfSequences; i++)
438 {

```

```

439
440 // MPS 5/23/05: Added in "-L+2" to make there only be one
441 // blank between sequences.
442 offsetToIndex[i] =
443 malloc ((strlen (mySequences[i].seq) - L + 2) * sizeof (int));
444 if (offsetToIndex[i] == NULL)
445 {
446 fprintf (stderr,
447 "Unable to allocate memory - offsetToIndex[%d] in gemoda.c\n%s\n",
448 i, strerror (errno));
449 fflush (stderr);
450 exit (0);
451 }
452
453 // MPS 5/23/05: Added in "-L+2" to make there only be one
454 // blank between sequences.
455 for (j = 0; j < (strlen (mySequences[i].seq) - L + 2); j++)
456 {
457 offsetToIndex[i][j] = numberOffsets;
458 numberOffsets++;
459 }
460 }
461
462 // Now create translation matrices such that we can get the sequence
463 // or position number of a given offset.
464 indexToSeq = (int *) malloc (numberOffsets * sizeof (int));
465 if (indexToSeq == NULL)
466 {
467 fprintf (stderr,
468 "Unable to allocate memory - indexToSeq in gemoda.c\n%s\n",
469 strerror (errno));
470 fflush (stderr);
471 exit (0);
472 }
473 indexToPos = (int *) malloc (numberOffsets * sizeof (int));
474 if (indexToPos == NULL)
475 {
476 fprintf (stderr,
477 "Unable to allocate memory - indexToPos in gemoda.c\n%s\n",
478 strerror (errno));
479 fflush (stderr);
480 exit (0);
481 }
482 k = 0;
483 for (i = 0; i < numberOfSequences; i++)
484 {
485
486 // MPS 5/23/05: Added in "-L+2" to make there only be one
487 // blank between sequences.
488 for (j = 0; j < (strlen (mySequences[i].seq) - L + 2); j++)
489 {
490 indexToSeq[k] = i;
491 indexToPos[k] = j;
492 k++;
493 }
494 }
495
496 // Now make an offset adjacency matrix!
497 //
498 oam = newBitGraph (numberOffsets);
499
500 // Go through each unique word
501 for (i = 0; i < wc; i++)
502 {
503 offset1 = words[i].offset;
504
505 // Go through each occurrence
506 for (k = 0; k < words[i].support; k++)

```

```

507     {
508
509         // Use the offsetToIndex translation to get the offset
510         // of the first occurrence
511         pos1 = offsetToIndex[offset1[k].seq][offset1[k].pos];
512
513         // And go through each word in the first offset to
514         // find words that meet the similarity threshold
515         for (j = 0; j < wc; j++)
516         {
517             if (bitGraphCheckBit (bg, i, j))
518             {
519                 offset2 = words[j].offset;
520
521                 // And find all of their occurrences,
522                 // using offsetToIndex to get the
523                 // offsets, and then setting those
524                 // locations in the offset adjacency
525                 // matrix true.
526                 for (l = 0; l < words[j].support; l++)
527                 {
528                     pos2 = offsetToIndex[offset2[l].seq][offset2[l].pos];
529                     bitGraphSetTrueSym (oam, pos1, pos2);
530                 }
531             }
532         }
533     }
534 }
535 fprintf (OUTPUT_FILE, "Offset matrix created...");  

536 deleteBitGraph (bg);
537 if ((samp > 0) && (clusterMethod == 0))
538 {
539     fprintf (OUTPUT_FILE, " taking preliminary statistics.\n");
540     fflush (NULL);
541     d =
542     getStatMat (oam, sup, L, &supportDim, &lengthDim, numberSequences,
543                 samp, OUTPUT_FILE);
544     fprintf (OUTPUT_FILE, "Now filtering...\n");
545     fflush (NULL);
546 }
547 else
548 {
549     fprintf (OUTPUT_FILE, " now filtering.\n");
550     fflush (NULL);
551     d = NULL;
552     supportDim = 0;
553 }
554
555 // Now we're convolving on offsets
556 allCliques =
557 convolve (oam, sup, R, indexToSeq, P, clusterMethod, offsetToIndex,
558             numberSequences, noConvolve, OUTPUT_FILE);
559
560 // Do some early memory cleanup to limit usage
561 oamSize = oam->size;
562 deleteBitGraph (oam);
563 fprintf (OUTPUT_FILE, "Convolved! Now making output...\n");
564 fflush (NULL);
565 if ((samp > 0) && (clusterMethod == 0))
566 {
567     cumDMatrix (d, allCliques, supportDim, lengthDim, oamSize,
568                 numberSequences);
569     calcStatAllCliqs (d, allCliques, numberOffsets - numberSequences);
570     allCliques = sortByStats (allCliques);
571 }
572
573 // walk over the cliques and give some output in the format:
574 // pattern <pattern id num>: len=<motif length> sup=<motif instances>

```

```

575 // <sequence num> <position num> <motif instance>
576 // ...
577 curCliq = allCliques;
578
579 i = 0;
580 while (curCliq != NULL)
581 {
582     fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d", i,
583             curCliq->length + L, curCliq->set->size);
584     if (d != NULL)
585     {
586         fprintf (OUTPUT_FILE, "\tsignif=%le\n", curCliq->stat);
587     }
588     else
589     {
590         fprintf (OUTPUT_FILE, "\n");
591     }
592
593     for (j = 0; j < curCliq->set->size; j++)
594     {
595         pos1 = curCliq->set->members[j];
596         curSeq = indexToSeq[pos1];
597         curPos = indexToPos[pos1];
598         fprintf (OUTPUT_FILE, "%d\t%d\t", curSeq, curPos);
599         for (k = curPos; k < curPos + curCliq->length + L; k++)
600         {
601             fprintf (OUTPUT_FILE, "%c", mySequences[curSeq].seq[k]);
602         }
603         fprintf (OUTPUT_FILE, "\n");
604     }
605     fprintf (OUTPUT_FILE, "\n\n");
606     curCliq = curCliq->next;
607     i++;
608 }
609
610 // And do some memory cleanup
611 // And cleanup of probability stuff...
612 /*
613     free(letterfreqs); delete_augmented_matrix(augmat);
614 */
615 allCliques = popAllCll (allCliques);
616 free (indexToSeq);
617 indexToSeq = NULL;
618 free (indexToPos);
619 indexToPos = NULL;
620 for (i = 0; i < numberofSequences; i++)
621 {
622     free (offsetToIndex[i]);
623     offsetToIndex[i] = NULL;
624 }
625
626 // Free'ing added by MPS, 6/4
627 for (i = 0; i < wc; i++)
628 {
629     free (words[i].offset);
630 }
631 free (words);
632
633 // End free'ing added by MPS
634 free (offsetToIndex);
635 offsetToIndex = NULL;
636
637 // -----
638
639 // Free up fastaSequences
640 FreeFSeqs (mySequences, numberofSequences);
641 fclose (OUTPUT_FILE);
642 return 0;

```

```
643 }
```

B.9.0.141 void matrixlist (void)

This function prints a list of the matrices that Gemoda can use to do the alignment of words. Most of these matrices are appropriate for amino acid sequences. In addition, there are matrices for DNA sequences and an identity matrix that is appropriate for other sequences, such as the analysis of English text. The matrix is selected using the -m flag.

Definition at line 99 of file gemoda-s.c.

Referenced by main().

```
100 {
101   fprintf (stdout, "\nThe following similarity matrices are installed "
102           "with the default Gemoda installation.\n Most of these "
103           "were obtained from publically available BLAST distributions. \n\n"
104           "dna_idmat:\n\t"
105           "Identity matrix for DNA: returns 1 when A,C,G,T are "
106           "compared to \n\tthemselves, 0 otherwise.\n\n"
107           "identity_aa:\n\t"
108           "Identity matrix for amino acids: returns 1 when any \n\t"
109           "letter but J,O,U are compared to themselves, and 0 "
110           "otherwise.\n\n" "idmat:\n\t"
111           "Similar to identity_aa, but it returns 10 in place "
112           "of 1.\n\n" "est_idmat:\n\t"
113           "Similar to idmat, but it returns -10 in place of 0. " "\n\n"
114           "pam100:\n" "pam110:\n" "pam120:\n" "pam130:\n"
115           "pam140:\n" "pam150:\n" "pam160:\n" "pam190:\n"
116           "pam200:\n" "pam210:\n" "pam220:\n" "pam230:\n"
117           "pam240:\n" "pam250:\n" "pam260:\n" "pam280:\n"
118           "pam290:\n" "pam300:\n" "pam310:\n" "pam320:\n"
119           "pam330:\n" "pam340:\n" "pam360:\n" "pam370:\n"
120           "pam380:\n" "pam390:\n" "pam400:\n" "pam430:\n"
121           "pam440:\n" "pam450:\n" "pam460:\n" "pam490:\n"
122           "pam500:\n\t"
123           "PAM matrices for various evolutionary distances.\n\n"
124           "blosum30:\n" "blosum35:\n" "blosum40:\n" "blosum45:\n"
125           "blosum50:\n" "blosum55:\n" "blosum60:\n" "blosum62:\n"
126           "blosum65:\n" "blosum70:\n" "blosum75:\n" "blosum80:\n"
127           "blosum85:\n" "blosum90:\n" "blosum100:\n\t"
128           "BLOSUM matrices for various evolutionary distances.\n\n"
129           "blosumnn:\n\t" "BLOSUM matrix of unknown origin.\n\n"
130           "dayhoff:\n\t"
131           "'Vanilla-flavored' pam250, very similar to pam250.\n\n"
132           "phat_t75_b73:\n" "phat_t80_b78:\n" "phat_t85_b82:\n\t"
133           "BLOSUM-clustered scoring matrix with target frequency\n\t"
134           "PHDhtm clustering = {75,80,85}percent and background frequency\n\t"
135           "Persson-Argos clustering = {73,78,82}percent.\n\t"
136           "From Ng, Henikoff, & Henikoff, Bioinformatics 16: 760.\n\n"
137           "coil_mat:\n" "alpha_mat:\n" "beta_mat:\n\t"
138           "Three structure-specific matrices described by Luthy,\n\t"
139           "McLachlan, and Eisenberg in Proteins 10, 229-239, obtained from AAindex.\n\n");
140   fprintf (stdout, "\n");
141 }
```

B.9.0.142 `bitGraph_t* pruneBitGraph (bitGraph_t * bg, int * indexToSeq, int ** offsetToIndex, int numSeqs, int p)`

Simple function (non-recursive) to prune off the first level of motifs that will not meet the "minimum number of unique sequences" criterion. This could have been implemented as above, but it may have gotten a little expensive with less yield, so only the first run through is done here. Input: a bit graph to be pruned, a pointer to the structure that dereferences offset indices to sequence numbers, a pointer to the structure that dereferences seq/position to offsets, the number of unique sequences in the input set, and the minimum number of unique sequences that must contain the motif. Output: a pruned bitGraph.

Definition at line 402 of file newConv.c.

References emptySet(), bitGraph_t::graph, and nextBitBitSet().

```

404 {
405     int i = 0, j = 0, nextBit = 0;
406     int *seqNums = NULL;
407
408     // Since we don't immediately know which node is in which source
409     // sequence, we can't just count them up regularly. Instead, we'll
410     // need to keep track of which sequences they come from and
411     // increment _something_. What we chose to do here is just make
412     // an array of integers of length = <p>. Then, we try to put the
413     // source sequence number of each neighbor (including itself, since
414     // the main diagonal is still true at this time) into the next slot
415     // Since we will monotonically search the bitSet, we can just
416     // move on to the first bit in the next sequence using the
417     // offsetToIndex structure so that we know the next sequence number
418     // to be put in is always unique.
419     seqNums = (int *) malloc (p * sizeof (int));
420     if (seqNums == NULL)
421     {
422         fprintf (stderr, "Memory error - pruneBitGraph\n%s\n",
423                 strerror (errno));
424         fflush (stderr);
425         exit (0);
426     }
427
428     // So, for each row in the bitgraph...
429     for (i = 0; i < bg->size; i++)
430     {
431
432         // Make sure the whole array is -1 sentinels.
433         for (j = 0; j < p; j++)
434         {
435             seqNums[j] = -1;
436         }
437         j = 0;
438
439         // Find the first neighbor of this bit.
440         nextBit = nextBitBitSet (bg->graph[i], 0);
441         if (nextBit == -1)
442         {
443             continue;
444         }
445         else
446         {
447
448             // and put its sequence number in the array of ints.
449             seqNums[0] = indexToSeq[nextBit];
450         }

```

```

451
452     // If it's the last sequence, then bail out so that we don't
453     // segfault in the next step.
454     if (seqNums[0] >= numSeqs - 1)
455     {
456         emptySet (bg->graph[i]);
457         continue;
458     }
459
460     // Find the next neighbor of this bit, STARTING AT the first
461     // bit in the next sequence.
462     nextBit =
463     nextBitBitSet (bg->graph[i],
464                     offsetToIndex[indexToSeq[nextBit] + 1][0]);
465
466     // And iterate this until we run out of neighbors.
467     while (nextBit >= 0)
468     {
469         j++;
470         seqNums[j] = indexToSeq[nextBit];
471
472         // Or until this new neighbor will fill up the array
473         if (j == p - 1)
474         {
475             break;
476         }
477
478         // Or until this new neighbor is in the last sequence.
479         if (seqNums[j] >= numSeqs - 1)
480         {
481             break;
482         }
483
484         // Get the next neighbor!
485         nextBit =
486         nextBitBitSet (bg->graph[i],
487                         offsetToIndex[indexToSeq[nextBit] + 1][0]);
488     }
489
490     // If we didn't have enough unique sequences, and either a) we
491     // were in the nth-to-last sequence and there were no
492     // neighbors after it, or b) we were in the last sequence,
493     // then the last number will still be our sentinel, -1. If
494     // the last number is not a sentinel, then we have at least
495     // p distinct sequence occurrences, so we're OK.
496     if (seqNums[p - 1] == -1)
497     {
498         emptySet (bg->graph[i]);
499     }
500 }
501 free (seqNums);
502 return (bg);
503 }
```

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.

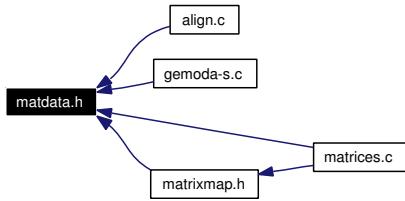
```

34     fprintf (stdout, "Usage: %s -i <Fasta sequence file> "
35         "-l <word size> \n\t-k <support> -g <threshold>"
36         "-m <matrix name> [-z] \n\t[-c <cluster method [0|1]>]"
37         "[ -p <unique support> ] \n\n"
38         "Required flags and input:\n\n"
39         "-i <Fasta sequence file>:\n\t"
40         "File containing all sequences to be searched, in Fasta format.\n\n"
41         "-l <word size>:\n\t"
42         "Minimum length of motifs; also the sliding window length\n\t"
43         "over which all motifs must meet the similarity criterion\n\n"
44         "-k <support>:\n\t" "Minimum number of motif occurrences.\n\n"
45         "-g <threshold>:\n\t"
46         "Similarity threshold. Two windows, when scored with the\n\t"
47         "similarity matrix defined by the -m flag, must have at least\n\t"
48
49         " this score in order to be deemed 'connected'. This criterion\n\t"
50         " must be met over all sliding windows of length l.\n\n"
51         "-m <matrix name>:\n\t"
52         "Name of the similarity matrix to be used to compare windows.\n\t"
53         "Use -z to see a list of matrices installed by default.\n\n"
54         "Optional flags and input:\n\n" "-z:\n\t"
55         "Lists all of the similarity matrices available with the\n\t"
56         "initial installation of Gemoda. Note that this overrides\n\t"
57         "all other options and will only give this output.\n\n"
58         "-c <cluster method [0|1]>:\n\t"
59         "The clustering method to be used after evaluating the "
60         "\n\tsimilarity of the unique words in the input. Note that the "
61
62         "\n\tclustering method will have a significant impact on both the "
63         "\n\tresults that one obtains and the computation time.\n\n\t"
64         "0: clique-finding\n\t"
65         "Uses established methods to find all maximal cliques in the "
66         "\n\t\data. This will give the most thorough results (that are "
67
68         "\n\t\tprobably exhaustive), but will also give less-significant "
69         "\n\t\tresults in addition to the most interesting and most\n\t"
70
71         "significant ones. The results are deterministic but may take some "
72
73         "\n\t\ttime on data sets with high similarity or if the similarity "
74         "\n\t\tthreshold is set extremely low.\n\t"
75         "1: single-linkage clustering\n\t"
76         "Uses a single-linkage-type clustering where all nodes that "
77         "\n\t\tare connected are put in the same cluster. This method is "
78
79         "\n\t\talso deterministic and will be faster than clique-finding, "
80
81         "\n\t\tbut it loses guarantees of exhaustiveness in searching the "
82         "\n\t\tdata set.\n\n" "-p <unique support>:\n\t"
83         "A pruning parameter that requires the motif to occur in "
84         "\n\tat least <unique support> different input sequences. Note "
85
86         "\n\tthat this parameter must be less than or equal to the total "
87         "\n\t\tsupport parameter set by the -k flag.\n\n", argv[0]);
88     fprintf (stdout, "\n");
89 }

```

B.10 matdata.h File Reference

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



Defines

- `#define MATRIX_SIZE 23`

Detailed Description

This file defines the size of the scoring matrices so that we don't have to pound-include the whole `matrices.h` file due to worries about incompatibilities with earlier extern variable declarations.

Definition in file [matdata.h](#).

Define Documentation

B.10.0.144 `#define MATRIX_SIZE 23`

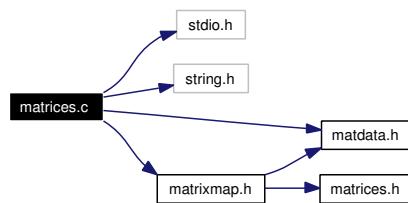
Definition at line 10 of file matdata.h.

Referenced by `main()`.

B.11 matrices.c File Reference

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

Include dependency graph for matrices.c:



Defines

- #define DEFAULT_MATRIX [blosum62](#)

Functions

- void [getMatrixByName](#) (char [name](#)[], const int(**[matp](#))[[MATRIX_SIZE](#)])

Detailed Description

This file contains functions for handling scoring matrices used for the sequence based Gemoda.
Definition in file [matrices.c](#).

Define Documentation

B.11.0.145 #define DEFAULT_MATRIX [blosum62](#)

Definition at line 7 of file [matrices.c](#).

Referenced by [getMatrixByName\(\)](#).

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

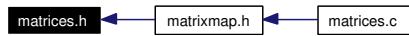
Definition at line 34 of file matrices.c.

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     {
41         break;
42     }
43     }
44     if (matrix_map[i].name != NULL)
45     {
46         *matp = (matrix_map[i].mat);
47     }
48     else
49     {
50         *matp = (DEFAULT_MATRIX);
51     }
52 }
```

B.12 matrices.h File Reference

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



Variables

- const int aaOrder []
- const int dna_idmat [MATRIX_SIZE][MATRIX_SIZE]
- const int identity_aa [MATRIX_SIZE][MATRIX_SIZE]
- const int idmat [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum100 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum30 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum35 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum40 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum45 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum50 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum55 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum60 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum62 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum65 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum70 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum75 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum80 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum85 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosum90 [MATRIX_SIZE][MATRIX_SIZE]
- const int blosumn [MATRIX_SIZE][MATRIX_SIZE]
- const int dayhoff [MATRIX_SIZE][MATRIX_SIZE]
- const int pam100 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam110 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam120 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam130 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam140 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam150 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam160 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam190 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam200 [MATRIX_SIZE][MATRIX_SIZE]
- const int pam210 [MATRIX_SIZE][MATRIX_SIZE]

- const int `pam220` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam230` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam240` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam250` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam260` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam280` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam290` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam300` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam310` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam320` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam330` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam340` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam360` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam370` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam380` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam390` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam400` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam430` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam440` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam450` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam460` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam490` [MATRIX_SIZE][MATRIX_SIZE]
- const int `pam500` [MATRIX_SIZE][MATRIX_SIZE]
- const int `phat_t75_b73` [MATRIX_SIZE][MATRIX_SIZE]
- const int `phat_t80_b78` [MATRIX_SIZE][MATRIX_SIZE]
- const int `phat_t85_b82` [MATRIX_SIZE][MATRIX_SIZE]
- const int `alpha_mat` [MATRIX_SIZE][MATRIX_SIZE]
- const int `beta_mat` [MATRIX_SIZE][MATRIX_SIZE]
- const int `coil_mat` [MATRIX_SIZE][MATRIX_SIZE]

Detailed Description

This file contains a number of scoring matrices, most of which are intended for comparing amino acid sequences; however a few are for DNA. In general, if a user wants to add their own matrix for use with Gemoda, they should add it to this file and recompile Gemoda.

Note that users are not restricted to 23x23 matrices. By changing aaOrder, you can easily make matrices for comparing ANSII strings with up to 256 different characters.

All of the matrices below were obtained directly from BLAST/WU-BLAST; they are all also part of the public domain, so there is nothing intrinsic to BLAST with respect to the matrices. It was just the easiest way to get all of the matrices into our software.

The most popular matrix for amino acid sequences is blosum62.

A good location for getting new scoring matrices, such as those based on structural data, is the AAIndex. URLs tend to change, so rather than us listing it here, Google it!

Definition in file [matrices.h](#).

Variable Documentation

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.12.0.175 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 const int phat_t80_b78[MATRIX_SIZE][MATRIX_SIZE]

Definition at line 1350 of file matrices.h.

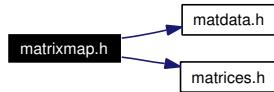
B.12.0.206 const int phat_t85_b82[MATRIX_SIZE][MATRIX_SIZE]

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:



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



Variables

- struct {

 char * name

 const int(* mat)[MATRIX_SIZE]

} matrix_map []

Detailed Description

This file contains structures and functions for handling scoring matrices.

Definition in file [matrixmap.h](#).

Variable Documentation

B.13.o.207 const int(* mat)[MATRIX_SIZE]

Definition at line 15 of file matrixmap.h.

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

B.13.o.208 struct { ... } matrix_map[]

This data structure maps the names of common matrices to the names of their variables

Referenced by getMatrixByName().

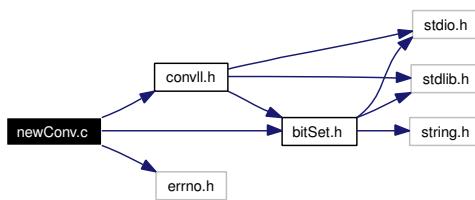
B.13.0.209 char* name

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:



Functions

- int [findCliques](#) (bitSet_t *Q, bitSet_t *cand, bitSet_t *mask, bitGraph_t *oG, int support, int qCount, cll_t **elemPats, int *indexToSeq, int p)
- int [singleLinkage](#) (bitSet_t *Q, bitSet_t *cand, bitSet_t *mask, bitGraph_t *oG, int support, int qCount, cll_t **elemPats, int *indexToSeq, int p)
- int [filterIter](#) (bitGraph_t *graph, int support, bitSet_t *changed, bitSet_t *work)
- int [filterGraph](#) (bitGraph_t *graph, int support, int R)
- bitGraph_t * [pruneBitGraph](#) (bitGraph_t *bg, int *indexToSeq, int **offsetToIndex, int numOfSeqs, int p)
- cll_t * [pruneCll](#) (cll_t *head, int *indexToSeq, int p)
- cll_t * [convolve](#) (bitGraph_t *bg, int support, int R, int *indexToSeq, int p, int clusterMethod, int **offsetToIndex, int numberOfSequences, int noConvolve, FILE *OUTPUT_FILE)

Detailed Description

This file contains the core functions that performed the convolution in the Gemoda algorithm. As well, there are two clustering functions defined in this file: one for single linkage clustering, and one for clique based clustering.

Definition in file [newConv.c](#).

Function Documentation

B.14.0.210 `dll_t* convolve (bitGraph_t * bg, int support, int R, int * indexToSeq, int p, int clusterMethod, int ** offsetToIndex, int numberOfSequences, int noConvolve, FILE * OUTPUT_FILE)`

Our outer convolution function. This function will call preliminary functions, cluster the data, and then call the main convolution function. This is the interface between the main gemoda-<x> code and the generic code that gets all of the work done. Input: the bitGraph to be clustered and convolved, the minimum support necessary for a motif to be returned, a flag indicating whether recursive filtering should be used, a pointer to the data structure that dereferences offset indices to sequence numbers, the number of unique source sequences that a motif must be present in, and a number indicating the clustering method that is to be used. Output: the final motif linked list with all motifs that are to be given as output to the user.

Definition at line 625 of file newConv.c.

References `bitGraphSetFalseDiagonal()`, `completeConv()`, `deleteBitSet()`, `fillSet()`, `filterGraph()`, `findCliques()`, `newBitSet()`, `pruneBitGraph()`, `pruneCll()`, `singleLinkage()`, `bitGraph_t::size`, and `yankCll()`.

```

629 {
630     bitSet_t * cand = NULL;
631     bitSet_t * mask = NULL;
632     bitSet_t * Q = NULL;
633     int size = bg->size;
634     cll_t * elemPats = NULL;
635     cll_t * allCliques = NULL;
636     cll_t * curr = NULL;
637
638     // contains indices (rows) containing the threshold value.
639     cand = newBitSet (size);
640     mask = newBitSet (size);
641     Q = newBitSet (size);
642     fillSet (cand);
643     fillSet (mask);
644
645     // Note that we prune based on p before setting the diagonal false.
646     if (p > 1)
647     {
648         bg =
649         pruneBitGraph (bg, indexToSeq, offsetToIndex, numberOfSequences, p);
650     }
651
652     // Now we set the main diagonal false for clustering and filtering.
653     bitGraphSetFalseDiagonal (bg);
654     filterGraph (bg, support, R);
655     fprintf (OUTPUT_FILE, "Graph filtered!  Now clustering...\n");
656     fflush (NULL);
657     if (clusterMethod == 0)
658     {
659         findCliques (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq, p);
660     }
661     else
662     {
663         singleLinkage (Q, cand, mask, bg, support, 0, &elemPats, indexToSeq,
664                         p);
665     }
666     fprintf (OUTPUT_FILE,
```

```

667     "Clusters found!  Now filtering clusters (if option set)...\\n");
668     fflush (NULL);
669     if (p > 1)
670     {
671         elemPats = pruneCll (elemPats, indexToSeq, p);
672     }
673     deleteBitSet (cand);
674     deleteBitSet (mask);
675     deleteBitSet (Q);
676
677     // Now let's convolve what we made.
678     if (noConvolve == 0)
679     {
680         fprintf (OUTPUT_FILE, "Now convolving...\\n");
681         fflush (NULL);
682         allCliques = completeConv (&elemPats, support, size, 0, indexToSeq, p);
683     }
684
685     else
686     {
687         curr = elemPats;
688         while (curr != NULL)
689         {
690             yankCll (&elemPats, NULL, &curr, &allCliques, 0);
691         }
692     }
693     return allCliques;
694 }
```

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 0 (and an altered bitGraph so that all nodes with connections have at least *<min support>="">* connections).

Definition at line 359 of file newConv.c.

References copySet(), countSet(), deleteBitSet(), emptySet(), filterIter(), newBitSet(), and bitGraph_t::size.

Referenced by convolve().

```

360 {
361     bitSet_t * changed = newBitSet (graph->size);
362     bitSet_t * work = newBitSet (graph->size);
363     emptySet (changed);
```

```

364     emptySet (work);
365
366     // Iteratively call the filtering by copying the previous "work" into
367     // "changed" after each iteration step.
368     if (R == 1)
369     {
370
371         do
372     {
373         filterIter (graph, support, changed, work);
374         copySet (work, changed);
375     }
376     while (countSet (changed) > 0);
377 }
378 else
379 {
380
381     // Otherwise, just do it once.
382     filterIter (graph, support, changed, work);
383 }
384 deleteBitSet (changed);
385 deleteBitSet (work);
386 return 0;
387 }
```

B.14.0.212 int filterIter ([bitGraph_t](#) * *graph*, int *support*, [bitSet_t](#) * *changed*, [bitSet_t](#) * *work*)

The iterator used to "filter" the graph. It takes information in the bitset telling which nodes' rows have changed and only checks them... this should make it pretty efficient time-wise at only a small memory cost. Note the convention that the first time this is called, the changed bitSet is empty... and that the master function is responsible for catching the signal that no changes were made in the last iteration. Input: the bitGraph to be filtered, the minimum support required for a motif to be returned, a bitSet with nodes changed from the previous iteration, and a bitSet to export the nodes changed in this iteration. Output: integer success value of 0 (and also a filtered bitGraph and a bitSet with the nodes changed in this iteration).

Definition at line 228 of file newConv.c.

References countSet(), emptySet(), [bitGraph_t](#)::graph, nextBitBitSet(), setFalse(), and setTrue().

Referenced by filterGraph().

```

230 {
231     int i = 0, j = 0;
232     int lastBit = 0, nextBit = 0, lastRow = 0, nextRow = 0;
233     int numNodes = 0;
234     int changedSize = countSet (changed);
235     emptySet (work);
236
237     // Note the convention that the first time the function is called,
238     // it is done with an empty "changed" bitSet as a sentinel. It is
239     // the responsibility of the master function calling the iterator
240     // to catch future empty changed sets to know that convergence has
241     // been achieved.
242     //
243     // So, if it's your first time through, go through each node and make
```

```

244 // sure that each is connected to at least <support> - 1 others.
245 if (changedSize == 0)
246 {
247     for (i = 0; i < graph->size; i++)
248     {
249         numNodes = countSet (graph->graph[i]);
250         if (numNodes >= support - 1)
251             {
252                 continue;
253             }
254         else
255             {
256
257                 // Otherwise, zero it out, but going one by
258                 // one so that you can also zero out the
259                 // symmetric bit.
260                 lastBit = 0;
261                 for (j = 0; j < numNodes; j++)
262                 {
263                     nextBit = nextBitBitSet (graph->graph[i], lastBit);
264                     if (nextBit == -1)
265                         {
266                             fprintf (stderr,
267                                 "\nEnd of bitSet reached! - initial\n");
268                             fflush (stderr);
269                             exit (0);
270                         }
271                     setFalse (graph->graph[i], nextBit);
272                     setFalse (graph->graph[nextBit], i);
273
274                     // And set that corresponding bit true
275                     // in the work bitSet so that we
276                     // know we changed it for the next
277                     // round.
278                     setTrue (work, nextBit);
279                     lastBit = nextBit + 1;
280                 }
281             }
282         }
283     }
284     else
285     {
286
287         // Otherwise, we've been here before, so just follow what
288         // the changed bitSet says to do... only those bitSets that
289         // were changed could possibly have gone under the minimum
290         // support requirement.
291         lastRow = 0;
292         for (i = 0; i < changedSize; i++)
293         {
294             nextRow = nextBitBitSet (changed, lastRow);
295             if (nextRow == -1)
296                 {
297                     fprintf (stderr, "\nEnd of bitSet reached! - iter,row\n");
298                     fflush (stderr);
299                     exit (0);
300                 }
301
302             // So now we've found the row that needs to be checked.
303             // We do the same thing we did above... either move
304             // on if it has enough possible support, or zero
305             // it out (with its symmetric locations) one by one.
306             numNodes = countSet (graph->graph[nextRow]);
307             if (numNodes >= support - 1)
308                 {
309                     lastRow = nextRow + 1;
310                     continue;
311                 }

```

```

312     else
313     {
314         lastBit = 0;
315         for (j = 0; j < numNodes; j++)
316     {
317         nextBit = nextBitBitSet (graph->graph[nextRow], lastBit);
318         if (nextBit == -1)
319         {
320             fprintf (stderr,
321                 "\nEnd of BitSet reached! = iter, Bit\n");
322             fflush (stderr);
323             exit (0);
324         }
325         setFalse (graph->graph[nextRow], nextBit);
326         setFalse (graph->graph[nextBit], nextRow);
327         setTrue (work, nextBit);
328         lastBit = nextBit + 1;
329     }
330     lastRow = nextRow + 1;
331 }
332 }
333 }
334 return 1;
335 }
```

B.14.0.213 int findCliques (*bitSet_t* * *Q*, *bitSet_t* * *cand*, *bitSet_t* * *mask*, *bitGraph_t* * *oG*, int *support*, int *qCount*, *cll_t* ** *elemPats*, int * *indexToSeq*, int *p*)

Recursive algorithm to exhaustively enumerate all of the maximal cliques that exist in the data. This is one of the main workhorses of Gemoda when used in its exhaustive form. This algorithm was originally published by Etsuji Tomita, Akira Tanaka, and Haruhisa Takahasi as a Technical Report of IPSJ (Information Processing Society of Japan): Tomita, E, A Tanaka, & H Takahasi (1989). "An optimal algorithm for finding all of the cliques". SIG Algorithms 12, pp 91-98. Input: a bitset with the nodes currently in the clique, a bitset with the candidates for expanding the clique, a bitset 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 0 (but more importantly, the elemPats clique linked list is expanded to contain all elementary (minimum-length) motif cliques.

Definition at line 37 of file newConv.c.

References bitSetIntersection(), checkBit(), countSet(), deleteBitSet(), bitGraph_t::graph, newBitSet(), nextBitBitSet(), pushClique(), setFalse(), setTrue(), and bitGraph_t::size.

Referenced by convolve().

```

40 {
41     bitSet_t ** gammaOG = NULL;
42     bitSet_t * candQ = newBitSet (oG->size);
43     bitSet_t * newMask = newBitSet (oG->size);
44     int i, q;
45     int graphSize;
46     int max = -1;
```



```

115     }
116     else if (qCount >= support)
117     {
118
119         // This should be done when:
120         // 1. countSet(newMask) == 0 [connected subgraph is maximal]
121         // 2. Qcount >= minCount [connected subgraph has enough nodes]
122         *elemPats = pushClique (Q, *elemPats, indexToSeq, p);
123     }
124
125     // Remove q from Q, and remove q from cand
126     setFalse (Q, q);
127     setFalse (cand, q);
128     q = nextBitBitSet (cand, q + 1);
129 }
130 qCount--;
131 deleteBitSet (candQ);
132 deleteBitSet (newMask);
133 return 0;
134 }
```

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%s\n",
423                 strerror (errno));
424         fflush (stderr);
425         exit (0);
426     }
```

```

427
428     // So, for each row in the bitgraph...
429     for (i = 0; i < bg->size; i++)
430     {
431
432         // Make sure the whole array is -1 sentinels.
433         for (j = 0; j < p; j++)
434         {
435             seqNums[j] = -1;
436         }
437         j = 0;
438
439         // Find the first neighbor of this bit.
440         nextBit = nextBitBitSet (bg->graph[i], 0);
441         if (nextBit == -1)
442         {
443             continue;
444         }
445         else
446         {
447
448             // and put its sequence number in the array of ints.
449             seqNums[0] = indexToSeq[nextBit];
450         }
451
452         // If it's the last sequence, then bail out so that we don't
453         // segfault in the next step.
454         if (seqNums[0] >= numSeqs - 1)
455         {
456             emptySet (bg->graph[i]);
457             continue;
458         }
459
460         // Find the next neighbor of this bit, STARTING AT the first
461         // bit in the next sequence.
462         nextBit =
463         nextBitBitSet (bg->graph[i],
464                         offsetToIndex[indexToSeq[nextBit] + 1][0]);
465
466         // And iterate this until we run out of neighbors.
467         while (nextBit >= 0)
468         {
469             j++;
470             seqNums[j] = indexToSeq[nextBit];
471
472             // Or until this new neighbor will fill up the array
473             if (j == p - 1)
474             {
475                 break;
476             }
477
478             // Or until this new neighbor is in the last sequence.
479             if (seqNums[j] >= numSeqs - 1)
480             {
481                 break;
482             }
483
484             // Get the next neighbor!
485             nextBit =
486             nextBitBitSet (bg->graph[i],
487                           offsetToIndex[indexToSeq[nextBit] + 1][0]);
488         }
489
490         // If we didn't have enough unique sequences, and either a) we
491         // were in the nth-to-last sequence and there were no
492         // neighbors after it, or b) we were in the last sequence,
493         // then the last number will still be our sentinel, -1. If
494         // the last number is not a sentinel, then we have at least

```

```

495     // p distinct sequence occurrences, so we're OK.
496     if (seqNums[p - 1] == -1)
497     {
498         emptySet (bg->graph[i]);
499     }
500 }
501 free (seqNums);
502 return (bg);
503 }
```

B.14.0.215 `cll_t* pruneCll (cll_t * head, int * indexToSeq, int p)`

Prunes a motif linked list of all motifs without support in at least

unique source sequences. Input: head of a motif linked list, pointer to a structure that dereferences offset indices to sequence numbers, minimum number of unique source sequences in which a motif must occur. Output: head of a (potentially altered) motif linked list.

Definition at line 514 of file newConv.c.

References cSet_t::members, cnode::next, cnode::set, and cSet_t::size.

Referenced by completeConv(), and convolve().

```

515 {
516     int i = 0, j = 0, thisSeq = 0;
517     int *seqNums = NULL;
518     cll_t * curr = head;
519     cll_t * prev = NULL;
520     cll_t * storage = NULL;
521
522     // We'll do this similar to the pruneBitGraph function... we will
523     // keep track of which source sequence each motif occurrence was in.
524     // Again, since the occurrences are listed monotonically, we only
525     // need to compare the last non-sentinel index to the current
526     // sequence number.
527     seqNums = (int *) malloc (p * sizeof (int));
528     if (seqNums == NULL)
529     {
530         fprintf (stderr, "Memory error - pruneCll\n%s\n", strerror (errno));
531         fflush (stderr);
532         exit (0);
533     }
534     while (curr != NULL)
535     {
536
537         // First make sure the set size is at least p.
538         // This is redundant, but extremely simple and not expensive,
539         // so we'll leave it in just as a check.
540         if (curr->set->size < p)
541         {
542             if (prev != NULL)
543             {
544                 prev->next = curr->next;
545             }
546             else
547             {
548                 head = curr->next;
549             }
550             storage = curr->next;
551             free (curr->set->members);
552             free (curr->set);
```

```

553     free (curr);
554     curr = storage;
555     continue;
556   }
557   for (i = 0; i < p; i++)
558   {
559     seqNums[i] = -1;
560   }
561   j = 0;
562   seqNums[0] = indexToSeq[curr->set->members[0]];
563
564   // Note, we've checked to make sure size > p, and we know
565   // p must be 2 or greater, so we can start at 1 without
566   // worrying about segfaulting
567   for (i = 1; i < curr->set->size; i++)
568   {
569     thisSeq = indexToSeq[curr->set->members[i]];
570     if (thisSeq != seqNums[j])
571     {
572       j++;
573       seqNums[j] = thisSeq;
574       if (j == p - 1)
575       {
576         break;
577       }
578     }
579   }
580
581   // Same story as before... if the last number is -1,
582   // then we didn't have enough to fill up the <p> different
583   // slots, so this doesn't meet our criterion.
584   if (seqNums[p - 1] == -1)
585   {
586     if (prev != NULL)
587     {
588       prev->next = curr->next;
589     }
590     else
591     {
592       head = curr->next;
593     }
594     storage = curr->next;
595     free (curr->set->members);
596     free (curr->set);
597     free (curr);
598     curr = storage;
599   }
600   else
601   {
602     prev = curr;
603     curr = curr->next;
604   }
605 }
606 free (seqNums);
607 return (head);
608 }

```

B.14.0.216 int singleLinkage (**bitSet_t** * *Q*, **bitSet_t** * *cand*, **bitSet_t** * *mask*, **bitGraph_t** * *oG*, int *support*, int *qCount*, **cll_t** ** *elemPats*, int * *indexToSeq*, int *p*)

A recursive routine for single linkage clustering. This clustering is much faster than exhaustively enumerating all cliques, but it puts each node in only one cluster and is not guaranteed to give all possible motifs. Input: a bitSet containing the current motif, a bitSet containing candidates

to be added to the current motif, a bitSet containing the current subgraph to be clustered, the original bitGraph to be clustered, the minimum support necessary for a motif to be returned, the current number of nodes in the motif, a linked list of elementary motifs (length is the same as the window size), pointer to a structure to derference index values to sequence numbers, and the minimum number of unique sequences that a motif must be in to be returned. Output: integer success value of o (but more importantly, the linked list elemPats is updated to contain all of the motifs of length = window size).

Definition at line 154 of file newConv.c.

References bitSetUnion(), checkBit(), copySet(), countSet(), bitGraph_t::graph, nextBitBitSet(), pushClique(), and setFalse().

Referenced by convolve().

```

157 {
158     int i = 0;
159     int j = 0;
160
161     // go to the first vertex that has not been clustered yet
162     i = nextBitBitSet (cand, 0);
163     if (i != -1)
164     {
165
166         // this vertex has been clustered
167         setFalse (cand, i);
168
169         // start a new cluster, Q
170         copySet (oG->graph[i], Q);
171
172         // go over each vertex in the cluster
173         j = nextBitBitSet (Q, 0);
174         while (j != -1)
175         {
176
177             // if this vertex has been clustered already, skip it and go
178             // to the next one
179             if (!checkBit (cand, j))
180             {
181                 j = nextBitBitSet (Q, j + 1);
182                 continue;
183             }
184
185             // Add this vertex's neighbors to the current cluster
186             bitSetUnion (Q, oG->graph[j], Q);
187
188             // This vertex has now been clustered
189             setFalse (cand, j);
190
191             // go over each vertex in the cluster
192             j = nextBitBitSet (Q, 0);
193         }
194
195         // Did we make a cluster that was large enough?
196         if (countSet (Q) >= support)
197         {
198             *elemPats = pushClique (Q, *elemPats, indexToSeq, p);
199         }
200
201         // recurse
202         singleLinkage (Q, cand, mask, oG, support, 0, elemPats, indexToSeq,
203                         p);
204     }

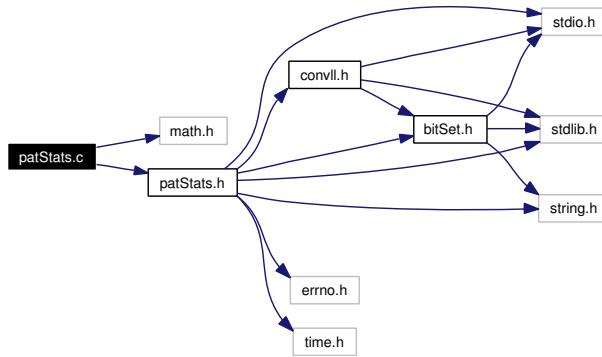
```

```
205     else
206     {
207         return 0;
208     }
209     return 0;
210 }
```

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 an 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 cumDMatix (unsigned int ** *d*, *cll_t* * *cliqs*, int *currSupport*, int *currLength*, int *bgSize*, int *numSeqs*)

Definition at line 522 of file patStats.c.

References getLargestLength(), and getLargestSupport().

Referenced by main().

```

524 {
525     int maxSup = 0;
526     int maxLen = 0;
527     int i, j;
528     int numWins = 0;
529
530     maxSup = getLargestSupport (cliqs);
531     maxLen = getLargestLength (cliqs);
532
533 /***** COMMENTED OUT
534     // First we note that the number of unique streaks of a given
535     // support is defined by d[support][1], where as 1 increases,
536     // the value of d decreases because only unique streaks are
537     // counted.
538     // We also note that the number of disjoint node-pairs with a given
539     // number of other nodes in common is defined by d[support][0].
540     // So, in order to properly account for all "unique" comparisons
541     // (which is equal to (# streaks + # disjoint node-pairs), we must
542     // add d[support][1] to d[support][0].
543
544     for (i = 0; i < currSupport + 1; i++) {

```

```

545     d[i][0] += d[i][1];
546 }
***** */
548
549 // We no longer need to do that, since now we sum across both
550 // the support and the length dimensions. Now, d[support][0] will
551 // necessarily include d[support][1] being added to it. We don't
552 // want to add this anymore, otherwise we would be underestimating
553 // the probability of making that first connection. For instance,
554 // if there were no nodes with 20 in common that weren't also
555 // connected, and no nodes whatsoever with more than 20 in common,
556 // we'd want the p[20][0] to be 1, which would be
557 // d[20][1]/d[20][0]. When summing across length directions,
558 // this happens naturally, whereas before we needed to do it
559 // artificially as per above. If we did above, we'd have the
560 // probability of each node being 1/2 instead of 1.
561
562 // Rather than storing doubles and doing lots of multiplications,
563 // we're going to limit the number of operations done in the actual
564 // probability calculation by only storing cumulative sums in d.
565 // Now remember, what we're storing at each location is the
566 // number of nodes with [i] or more nodes in common (including
567 // each other and selves) that can be extended [j] times (with
568 // their initial similarity counting as 1).
569 //
570 // We go up to the last possible index in the length direction, which
571 // means going up to [maxLen]. We know that this is legitimate
572 // because maxLen is less than or equal to the longest possible
573 // diagonal, and the longest possible diagonal will be less
574 // than or equal to currLength. Since we have allotted
575 // (currLength + 1) integers, we know we're OK to access [currLength].
576 for (j = 0; j < currLength + 1; j++)
577 {
578
579 // We start at currSupport - 1, because currSupport will
580 // clearly not be changed, and this makes it a much easier
581 // loop to read.
582 for (i = currSupport - 1; i >= 0; i--)
583 {
584     d[i][j] += d[i + 1][j];
585 }
586 }
587 for (i = 0; i < currSupport + 1; i++)
588 {
589     for (j = currLength - 1; j >= 0; j--)
590     {
591         d[i][j] += d[i][j + 1];
592     }
593 }
594
595 // Now we need to forcibly set d[0][0] to its correct value... it's
596 // just the total number of comparisons, not including comparisons
597 // to delimiter 0's meant to separate sequences. The number of
598 // windows is equal to the number of offsets minus the number
599 // of sequences (assuming one delimiter per sequence). We don't count
600 // the main diagonal, so the first row has one less, and we want to
601 // sum over all the subsequent rows in the upper half of the matrix.
602 // So it's (numWins - 1)*(numWins - 1 + 1)/2 to sum that up.
603 numWins = bgSize - numSeqs;
604 d[0][0] = numWins * (numWins - 1) / 2;
605
606 /*
607     for (i = 0; i <= maxSup; i++) { printf("support = %d:\t",i); for (j = 0; j <
608     maxLen; j++) { printf("%d\t",d[i][j]); } printf("\n"); }
609 */
610 return 1;
611 }

```

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     {
693         return 0;
694     }
695     else
696     {
697
698         // Still, it's supportDim + 1, because we have an extra
699         // one for the "0" support.
700         for (i = 0; i < supportDim + 1; i++)
701         {
702             free (d[i]);
703         }
704         free (d);
705         return 0;
706     }
707 }
```

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     {
53         len = curCliq->length;
54     }
55     curCliq = curCliq->next;
56 }
57
58     // We return (len + 1) because the length of the shortest streak
59     // is one, but is stored in the cluster data structure as being
60     // zero (number of extensions that have been made).
61     return (len + 1);
62 }
```

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().

```

23 {
24     int size = 0;
25     cll_t * curCliq = NULL;
26     curCliq = cliqs;
27     while (curCliq != NULL)
28     {
29         if (curCliq->set->size > size)
30         {
31             size = curCliq->set->size;
32         }
33         curCliq = curCliq->next;
34     }
35     return size;
36 }
```

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().

```

331 {
332     int *Q = NULL;
333     unsigned int **d = NULL;
334     int i, j, k;
335     int x, y;
336     bitSet_t * X = NULL;
337     int currSupport;
338     int currLength;
339     int multiplier = 50;
340     int diagonal = 0;
341     time_t probStart, probEnd;
342     int timeNeeded = 0;
343     int sampleCounter = 1;
344
345     // int visitCounter = 0, uniqCounter = 0;
346     currSupport = support * multiplier;
347     currLength = length * multiplier;
348     X = newBitSet (bg->size);
349
350     // printf("Made bitSet of size %d\n", bg->size);
```

```

351     Q = (int *) malloc (bg->size * sizeof (int));
352     if (Q == NULL)
353     {
354         fprintf (stderr,
355             "\nMemory error --- couldn't allocate array!"    "%s\n",
356             strerror (errno));
357         fflush (stderr);
358         exit (0);
359     }
360     for (i = 0; i < bg->size; i++)
361     {
362         Q[i] = 0;
363     }
364     d =
365     (unsigned int **) malloc ((currSupport + 1) * sizeof (unsigned int *));
366     if (d == NULL)
367     {
368         fprintf (stderr,
369             "\nMemory error --- couldn't allocate array!"    "%s\n",
370             strerror (errno));
371         fflush (stderr);
372         exit (0);
373     }
374     for (i = 0; i < currSupport + 1; i++)
375     {
376         d[i] =
377         (unsigned int *) malloc ((currLength + 1) * sizeof (unsigned int));
378         if (d[i] == NULL)
379         {
380             fprintf (stderr, "\nMemory error --- couldn't allocate array!"
381                 "%s\n", strerror (errno));
382             fflush (stderr);
383             exit (0);
384         }
385         for (j = 0; j < currLength + 1; j++)
386         {
387             d[i][j] = 0;
388         }
389     }
390
391 // printf("size=%d\n",bg->size);
392 time (&probStart);
393 for (i = 0; i < bg->size; i++)
394 {
395     if (i == 200)
396     {
397         time (&probEnd);
398         timeNeeded = ((double) (probEnd - probStart)) /
399             ((double) 60) * ((double) bg->size) / ((double) 200);
400         if (timeNeeded > 2)
401         {
402             fprintf (OUTPUT_FILE,
403                 "Max total time to calculate probability:\n");
404             fprintf (OUTPUT_FILE, "\t%d minutes\n", timeNeeded);
405             fprintf (OUTPUT_FILE, "Actual time will be less than this, "
406                 "but at least half of it.\n");
407             fprintf (OUTPUT_FILE,
408                 "To bypass excessive probability calculations,"
409                 " cancel and use a different value\n"
410                 " for the '-s' flag (samples every "
411                 "'s' points).\n");
412             fflush (NULL);
413         }
414     }
415     j = nextBitBitSet (bg->graph[i], 0);
416     while (j >= 0)
417     {
418         k = nextBitBitSet (bg->graph[i], j + 1);

```

```

419     while (k >= 0)
420     {
421         if (checkBit (bg->graph[j], k) == 0)
422         {
423             if (sampleCounter == s)
424             {
425                 bitGraphRowIntersection (bg, j, k, X);
426
427                 // visitCounter++;
428                 if (nextBitBitSet (X, 0) >= i)
429                 {
430
431                     // uniqCounter++;
432                     x = countSet (X);
433                     while (x > currSupport)
434                     {
435                         d =
436                         increaseMem (d, 1, currSupport, currLength,
437                                     currSupport +
438                                     support * multiplier);
439                         currSupport += support * multiplier;
440                     }
441                     d[x][0] += 1;
442                 }
443                 sampleCounter = 0;
444             }
445             sampleCounter++;
446         }
447         k = nextBitBitSet (bg->graph[i], k + 1);
448     }
449     if (j <= i)
450     {
451         j = nextBitBitSet (bg->graph[i], j + 1);
452         continue;
453     }
454     bitGraphRowIntersection (bg, i, j, X);
455     x = countSet (X);
456
457     // Note, now we're using "diagonals" rather than
458     // location in a horizontal array. So you always
459     // start from the main diagonal at 0 and move out.
460     diagonal = j - i;
461
462     // We change this to greater-than-one because
463     // after Q[diagonal] is reduced to one, it isn't
464     // visited again until we reach a new streak, (because
465     // the next bit in the diagonal is a zero), and at
466     // that point we want to start with a new diagonal
467     // measure.
468     if (Q[diagonal] > 1)
469     {
470         y = Q[diagonal] - 1;
471         Q[diagonal]--;
472     }
473     else
474     {
475         y = measureDiagonal (bg, i, j);
476         Q[diagonal] = y;
477     }
478     while (x > currSupport)
479     {
480         d = increaseMem (d, 1, currSupport, currLength,
481                         currSupport + support * multiplier);
482         currSupport += support * multiplier;
483     }
484     while (y > currLength)
485     {
486         d =

```

```

487     increaseMem (d, 2, currSupport, currLength,
488                 currLength + length * multiplier);
489     currLength += length * multiplier;
490   }
491   d[x][y]++;
492   j = nextBitBitSet (bg->graph[i], j + 1);
493
494   /*
495    if(x != 0){ printf("%d:\t%d %d\n", j, x, y); fflush(stdout); }
496   */
497 }
498
499 /*
500  printf("done\n"); fflush(stdout);
501 */
502 }
503
504 // We need to rescale by the sampling factor for all i>0 in d[i][0].
505 //
506 for (i = 1; i < currSupport; i++)
507 {
508   d[i][0] *= s;
509 }
510
511 // Now we only need to assign the correct value for d[0][0]...
512 // but rather than figuring that out, we will just assign it in the
513 // cumulative function, since there it is merely the number of unique
514 // non-self comparisons and is easy to calculate.
515 deleteBitSet (X);
516 free (Q);
517 *supportDim = currSupport;
518 *lengthDim = currLength;
519 return (d);
520 }
```

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().

```

93 {
94   int i = 0, j = 0;
95   if (dimToChange == 1)
96   {
97     d =
98     (unsigned int **) realloc (d, (newVal + 1) * sizeof (unsigned int *));
99     if (d == NULL)
100    {
101      fprintf (stderr, "\nMemory error --- couldn't allocate array!"
102              "\n%s\n", strerror (errno));
103      fflush (stderr);
104      exit (0);
105    }
106    for (i = currSupport + 1; i < newVal + 1; i++)
```

```

107     {
108         d[i] =
109             (unsigned int *) malloc ((currLength + 1) *
110                             sizeof (unsigned int));
111         if (d[i] == NULL)
112         {
113             fprintf (stderr,
114                 "\nMemory error --- couldn't allocate array!"
115                 "\n%s\n", strerror (errno));
116             fflush (stderr);
117             exit (0);
118         }
119         for (j = 0; j < currLength + 1; j++)
120         {
121             d[i][j] = 0;
122         }
123     }
124     return d;
125 }
126 else if (dimToChange == 2)
127 {
128     for (i = 0; i < currSupport + 1; i++)
129     {
130         d[i] =
131             (unsigned int *) realloc (d[i],
132                             (newVal + 1) * sizeof (unsigned int));
133         if (d[i] == NULL)
134         {
135             fprintf (stderr,
136                 "\nMemory error --- couldn't allocate array!"
137                 "\n%s\n", strerror (errno));
138             fflush (stderr);
139             exit (0);
140         }
141         for (j = currLength + 1; j < newVal + 1; j++)
142         {
143             d[i][j] = 0;
144         }
145     }
146     return d;
147 }
148 else
149 {
150     fprintf (stderr, "Invalid arguments to increaseMem!\n\n");
151     fflush (stderr);
152     exit (0);
153 }
154 }
```

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().

```

73 {
74     int len = 0;
75     while (bitGraphCheckBit (bg, i + len, j + len) != 0)
76     {
77         len++;
78     }
79     return len;
80 }

```

B.15.0.226 `unsigned int oldGetStatMat (bitGraph_t * bg, int support, int length, int * supportDim, int * lengthDim, int numBlanks)`**

OK, here is something that is a little bit "hackish" but that we have to do. Since our initial matrix is being pruned and filtered before being clustered, but we need to calculate stats based on the original matrix, we need to get information from the matrix before pruning, so we're using this function. We could just make a copy of that matrix, but it's far too big, and that would cause an unnecessary 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`.

```

198 {
199     int *Q = NULL;
200     unsigned int **d = NULL;
201     int i, j;
202     int x, y;
203     bitSet_t * X = NULL;
204     int currSupport;
205     int currLength;
206     int multiplier = 50;

```

```

207     time_t probStart, probEnd;
208     int timeNeeded = 0;
209     currSupport = support * multiplier;
210     currLength = length * multiplier;
211     X = newBitSet (bg->size);
212
213     // printf("Made bitSet of size %d\n", bg->size);
214     Q = (int *) malloc (bg->size * sizeof (int));
215     if (Q == NULL)
216     {
217         fprintf (stderr,
218             "\nMemory error --- couldn't allocate array!"  "\n%s\n",
219             strerror (errno));
220         fflush (stderr);
221         exit (0);
222     }
223     for (i = 0; i < bg->size; i++)
224     {
225         Q[i] = 0;
226     }
227     d =
228     (unsigned int **) malloc ((currSupport + 1) * sizeof (unsigned int *));
229     if (d == NULL)
230     {
231         fprintf (stderr,
232             "\nMemory error --- couldn't allocate array!"  "\n%s\n",
233             strerror (errno));
234         fflush (stderr);
235         exit (0);
236     }
237     for (i = 0; i < currSupport + 1; i++)
238     {
239         d[i] =
240         (unsigned int *) malloc ((currLength + 1) * sizeof (unsigned int));
241         if (d[i] == NULL)
242         {
243             fprintf (stderr, "\nMemory error --- couldn't allocate array!"
244                 "\n%s\n", strerror (errno));
245             fflush (stderr);
246             exit (0);
247         }
248         for (j = 0; j < currLength + 1; j++)
249         {
250             d[i][j] = 0;
251         }
252     }
253     time (&probStart);
254     for (i = 0; i < bg->size; i++)
255     {
256         if (i == 200)
257         {
258             time (&probEnd);
259             timeNeeded = ((double) (probEnd - probStart)) /
260             ((double) 60) * ((double) bg->size) / ((double) 200);
261             if (timeNeeded > 2)
262             {
263                 printf ("Max total time to calculate probability:\n");
264                 printf ("\t%d minutes\n", timeNeeded);
265                 printf ("Actual time will be less than this, but at",
266                     "least half of it.\n");
267                 printf ("To bypass excessive probability calculations,",
268                     "cancel and use the '-d' flag.\n");
269                 fflush (NULL);
270             }
271         }
272         for (j = bg->size - 1; j > i; j--)
273     {
274         bitGraphRowIntersection (bg, i, j, X);

```

```

275     x = countSet (X);
276     if (Q[j - 1] != 0)
277     {
278         y = Q[j - 1] - 1;
279         Q[j] = Q[j - 1] - 1;
280     }
281     else
282     {
283         y = measureDiagonal (bg, i, j);
284         Q[j] = y;
285     }
286     while (x > currSupport)
287     {
288         d = increaseMem (d, 1, currSupport, currLength,
289                         currSupport + support * multiplier);
290         currSupport += support * multiplier;
291     }
292     while (y > currLength)
293     {
294         d =
295         increaseMem (d, 2, currSupport, currLength,
296                         currLength + length * multiplier);
297         currLength += length * multiplier;
298     }
299     d[x][y]++;
300
301     /*
302      if(x != 0){ printf("%d:\t%d %d\n", j, x, y); fflush(stdout); }
303     */
304 }
305
306 /*
307  printf("done\n"); fflush(stdout);
308 */
309 }
310
311 // We know that the "blanks", inserted to delimit unique sequences
312 // and prevent convolution through them, will skew our statistics,
313 // so we subtract them. We know that they will never be similar to
314 // any others, so will only add to the d[0][0] number. Furthermore,
315 // we know how many they add. Since d never hits the main diagonal
316 // and only does the upper half of the matrix, the first one
317 // contributes bssize - 1 to d[0][0], the next bssize - 2, etc.
318 for (i = 0; i < numBlanks; i++)
319 {
320     d[0][0] -= bg->size - 1 - i;
321 }
322 deleteBitSet (X);
323 free (Q);
324 *supportDim = currSupport;
325 *lengthDim = currLength;
326 return (d);
327 }
```

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().

```

733 {
734     cll_t * curCliq = NULL;
735     cll_t ** arrayOfCliqs = NULL;
736     int numOfCliqs = 0;
737     int i = 0;
738     curCliq = allCliqs;
739     if (curCliq != NULL)
740     {
741         numOfCliqs = curCliq->id + 1;
742     }
743     else
744     {
745         return (NULL);
746     }
747     arrayOfCliqs = (cll_t **) malloc (numOfCliqs * sizeof (cll_t *));
748     for (i = 0; i < numOfCliqs; i++)
749     {
750         arrayOfCliqs[i] = curCliq;
751         curCliq = curCliq->next;
752     }
753     qsort (arrayOfCliqs, numOfCliqs, sizeof (cll_t *), statCompare);
754     for (i = 0; i < numOfCliqs - 1; i++)
755     {
756         arrayOfCliqs[i]->next = arrayOfCliqs[i + 1];
757     }
758     arrayOfCliqs[numOfCliqs - 1]->next = NULL;
759     return (arrayOfCliqs[0]);
760 }
```

B.15.0.228 **int statCompare (const cll_t ***first*, const cll_t ***second*)**

Definition at line 709 of file patStats.c.

Referenced by sortByStats().

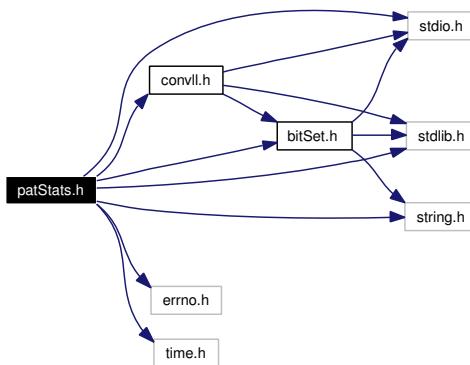
```

710 {
711     double difference = (*first)->stat - (*second)->stat;
712     if (difference < 0)
713     {
714         return (-1);
715     }
716     else if (difference > 0)
717     {
718         return (1);
719     }
720     else
721     {
722         return (0);
723     }
724 }
```

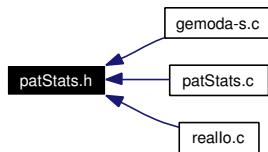
B.16 patStats.h File Reference

```
#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 calcStatCliq(), cnode::next, and cnode::stat.

Referenced by main().

B.16.0.230 int cumDMatrix (*unsigned int ** d, cll_t * cliqs, int currSupport, int currLength, int bgSize, int numSeqs*)

Definition at line 460 of file patStats.c.

References getLargestLength(), and getLargestSupport().

Referenced by main().

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.

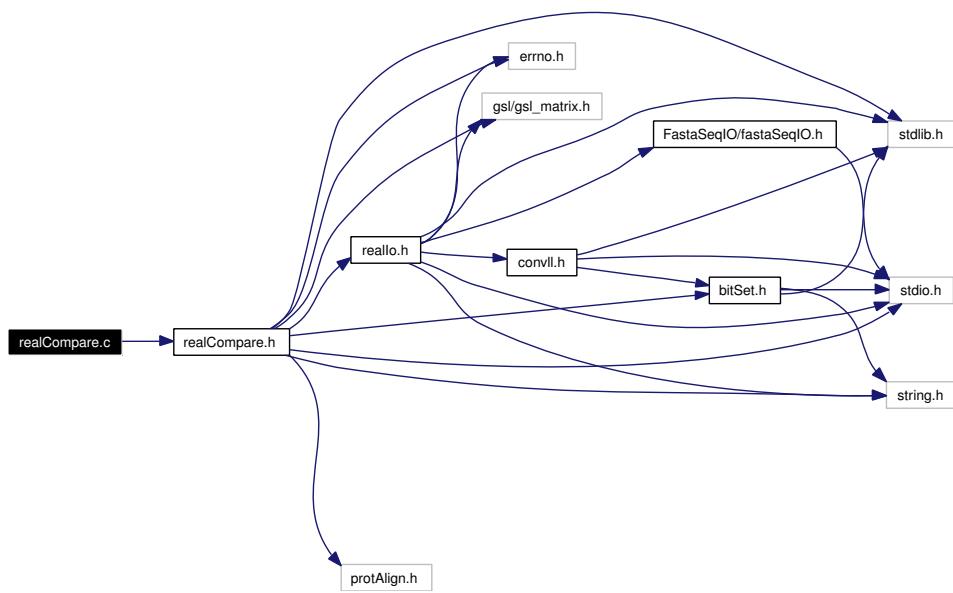
References cnode::id.

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 *extraParams)
- double(*)(*rdh_t* *, int, int, int, double *) [getCompFunc](#) (int compFunc)
- [bitGraph_t](#) * [realComparison](#) (*rdh_t* *data, int L, double g, int compFunc, double *extraParams)

Detailed Description

This file defines a series of functions that are used during the comparison phase of the Gemoda algorithm in the real valued implementation. We define a handful of comparison functions — some that are well suited to protein structure comparison and others that are more suited to the comparison of mass spectrometry spectra.

Definition in file [realCompare.c](#).

Function Documentation

B.17.0.234 double generalMatchFactor (*rdh_t* * *data*, int *win1*, int *win2*, int *L*, double * *extraParams*)

This function is used to compute a generalized match factor, which is useful for computing the degree of similarity between mass spectrometry spectra.

Definition at line 111 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc().

```

113 {
114     int i, j;
115     double numerator = 0.0;
116
117     /*
118      double denominator=0.0;
119     */
120     double xsum;
121     double ysum;
122     double ldenom = 0.0;
123     double rdenom = 0.0;
124     int dim;
125     int seq1, pos1;
126     int seq2, pos2;
127     gsl_matrix_view view1;
128     gsl_matrix_view view2;
129     gsl_matrix * mat1;
130     gsl_matrix * mat2;
131     dim = getRdhDim (data);
132
133     // Find out which seq,pos pairs these two
134     // windows correspond to
135     getRdhIndexSeqPos (data, win1, &seq1, &pos1);
136     getRdhIndexSeqPos (data, win2, &seq2, &pos2);
137
138     // Get a reference to a submatrix. That is,
139     // 'chop out' the window.
140     view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
141     view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);
142
143     // Some error checking here would be nice!
144     // Did we get the matrices we wanted?
145
146     // This just makes it easier to handle the views
147     mat1 = &view1.matrix;
148     mat2 = &view2.matrix;
149
150     // Loop over each position
151     for (i = 0; i < mat1->sizel; i++)
152     {
153         xsum = 0.0;
154         ysum = 0.0;
155
156         // Loop over each dimension at each position
157         for (j = 0; j < dim; j++)
158         {
159             xsum += gsl_matrix_get (mat1, i, j);
160             ysum += gsl_matrix_get (mat2, i, j);
161         }
162         numerator += (i + 1) * sqrt (xsum * ysum);

```

```

163     ldenom += (i + 1) * xsum;
164     rdenom += (i + 1) * ysum;
165 }
166 return pow (numerator, 2.0) / (ldenom * rdenom);
167 }
```

B.17.0.235 double(*)(**rdh_t** *, int, int, int, double *) **getCompFunc()**

Definition at line 264 of file realCompare.c.

References generalMatchFactor(), massSpecCompareWEElut(), 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 = &massSpecCompareWEElut;
277             break;
278         default:
279             comparisonFunc = &rmsdCompare;
280             break;
281     }
282     return (comparisonFunc);
283 }
```

B.17.0.236 double **massSpecCompareWEElut** (**rdh_t** * *data*, int *win1*, int *win2*, int *L*, double * *extraParams*)

This function is used to compute the match factor between two mass spectrometry spectra in a similar manner to the previous function; however, this function imposes a penalty for spectra that are separated by large distances in elution time. This function is commonly used by SpecConnect.

Definition at line 178 of file realCompare.c.

References getRdhDim(), getRdhIndexSeqPos(), and rdh_t::seq.

Referenced by getCompFunc().

```

180 {
181     int i, j;
182     double numerator = 0.0;
183
184     /*
185      double denominator=0.0;
186      */
187     double xsum;
188     double ysum;
189     double cum;
```

```

190  double ldenom = 0.0;
191  double rdenom = 0.0;
192  int dim;
193  int seq1, pos1;
194  int seq2, pos2;
195  double weight = 2.0;
196  gsl_matrix_view view1;
197  gsl_matrix_view view2;
198  gsl_matrix * mat1;
199  gsl_matrix * mat2;
200  double maxElut = -1;
201  if (extraParams != NULL)
202  {
203      maxElut = extraParams[0];
204  }
205  dim = getRdhDim (data);
206
207  // Find out which seq, pos pairs these two
208  // windows correspond to
209  getRdhIndexSeqPos (data, win1, &seq1, &pos1);
210  getRdhIndexSeqPos (data, win2, &seq2, &pos2);
211
212  // Get a reference to a submatrix. That is,
213  // 'chop out' the window.
214  view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
215  view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);
216
217  // Some error checking here would be nice!
218  // Did we get the matrices we wanted?
219
220  // This just makes it easier to handle the views
221  mat1 = &view1.matrix;
222  mat2 = &view2.matrix;
223  cum = 1.0;
224
225  // Loop over each position
226  for (i = 0; i < mat1->sizel; i++)
227  {
228      xsum = 0.0;
229      ysum = 0.0;
230
231      // First take the first dimension for elution time
232      if (maxElut >= 0)
233      {
234          if (fabs
235              (gsl_matrix_get (mat1, i, 0) - gsl_matrix_get (mat2, i, 0)) >
236              maxElut)
237          {
238              cum = 0;
239              break;
240          }
241      }
242
243      // printf("\n");
244      //
245      // Loop over each subsequent dimension at each position
246      for (j = 1; j < dim; j++)
247      {
248
249          // printf("mat1val=%lf,mat2val=%lf\n",gsl_matrix_get(mat1,i,j),
250          // gsl_matrix_get(mat2,i,j));
251          numerator += pow (j, weight) * sqrt (gsl_matrix_get (mat1, i, j)
252                                         *gsl_matrix_get (mat2, i,
253                                         j));
254          ldenom += pow (j, weight) * gsl_matrix_get (mat1, i, j);
255          rdenom += pow (j, weight) * gsl_matrix_get (mat2, i, j);
256
257          // printf("numer=%lf,ldenom=%lf,rdenom=%lf\n",numerator,

```

```

258         // ldenom,rdenom);
259     }
260     cum *= pow (numerator, 2.0) / (ldenom * rdenom);
261   }
262   return pow (cum, 1.0 / L);
263 }
```

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().

```

287 {
288   int i, j;
289   int seq1, pos1;
290   int seq2, pos2;
291   bitGraph_t * bg = NULL;
292   double score;
293   double (*comparisonFunc) (rdh_t *, int, int, int, double *) = &rmsdCompare;
294
295   // Initialize the rdh's index
296   initRdhIndex (data, L, 1);
297
298   // Allocate a new bit graph
299   bg = newBitGraph (data->indexSize);
300
301   // Choose the comparison function, pass a reference to it
302   comparisonFunc = getCompFunc (compFunc);
303   for (i = 0; i < data->indexSize; i++)
304   {
305
306     // Skip separators
307     getRdhIndexSeqPos (data, i, &seq1, &pos1);
308     if (seq1 == -1 || pos1 == -1)
309     {
310       continue;
311     }
312     for (j = i; j < data->indexSize; j++)
313     {
314       getRdhIndexSeqPos (data, j, &seq2, &pos2);
315       if (seq2 == -1 || pos2 == -1)
316       {
317         continue;
318       }
319
320       // This is the comparison function
321       score = comparisonFunc (data, i, j, L, extraParams);
322
323       // printf("score (%2d,%2d) vs. (%2d, %2d) =\t%lf\n",seq1, pos1, seq2, pos2,
324       // score);
325       if (compFunc == 0)
326       {
327         if (score <= g)
328         {
329           bitGraphSetTrueSym (bg, i, j);
330         }
331       }

```

```

332     else if ((compFunc == 1) || (compFunc == 2))
333     {
334         if (score >= g)
335         {
336             bitGraphSetTrueSym (bg, i, j);
337         }
338     }
339     else
340     {
341         fprintf (stderr, "Comparison function undefined in "
342                 "realComparison function,\n located in "
343                 "realCompare.c. Exiting.\n\n");
344         fflush (stderr);
345         exit (0);
346     }
347 }
348 }
349 return bg;
350 }
```

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\(\)](#).

```

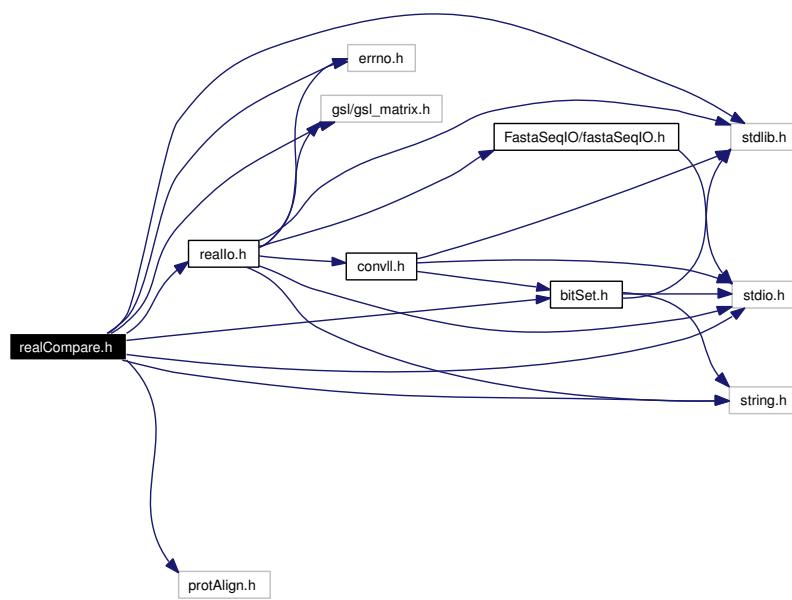
32 {
33     int trans = 1;
34     int rot = 1;
35     int dim;
36     double result = 0;
37     int seq1, pos1;
38     int seq2, pos2;
39     gsl_matrix_view view1;
40     gsl_matrix_view view2;
41     gsl_matrix * mat1;
42     gsl_matrix * mat2;
43     gsl_matrix * mat1copy;
44     gsl_matrix * mat2copy;
45
46     // The "rint" function is in math.h and rounds a number to the
47     // nearest integer. It raises an "inexact exception" if the
48     // number initially wasn't an integer.
```

```
49     if (extraParams != NULL)
50     {
51         trans = rint (extraParams[0]);
52         rot = rint (extraParams[1]);
53     }
54 dim = getRdhDim (data);
55
56 // Find out which seq,pos pairs these two
57 // windows correspond to
58 getRdhIndexSeqPos (data, win1, &seq1, &pos1);
59 getRdhIndexSeqPos (data, win2, &seq2, &pos2);
60
61 // Get a reference to a submatrix. That is,
62 // 'chop out' the window.
63 view1 = gsl_matrix_submatrix (data->seq[seq1], pos1, 0, L, dim);
64 view2 = gsl_matrix_submatrix (data->seq[seq2], pos2, 0, L, dim);
65
66 // This just makes it easier to handle the views
67 mat1 = &view1.matrix;
68 mat2 = &view2.matrix;
69
70 // Create copies of the windows, because our comparison
71 // will require altering the matrices
72 mat1copy = gsl_matrix_alloc (mat1->size1, mat1->size2);
73 mat2copy = gsl_matrix_alloc (mat2->size1, mat2->size2);
74 gsl_matrix_memcpy (mat1copy, mat1);
75 gsl_matrix_memcpy (mat2copy, mat2);
76
77 /*
78     printf("matrix1:\n"); gsl_matrix_pretty_fprintf(stdout, mat1copy, "%f ");
79     printf("\nmatrix2:\n"); gsl_matrix_pretty_fprintf(stdout, mat2copy, "%f ");
80 */
81
82 // Are we going to do a translation?
83 if (trans == 1)
84 {
85     moveToCentroid (mat1copy);
86     moveToCentroid (mat2copy);
87 }
88
89 // Are we going to do a rotation?
90 if (rot == 1)
91 {
92
93     // Rotate mat2copy to have a minimal
94     // rmsd with mat1copy
95     rotateMats (mat1copy, mat2copy);
96 }
97
98 // Compute the rmsd between mat2copy and mat2copy
99 result = gsl_matrix_rmsd (mat1copy, mat2copy);
100 gsl_matrix_free (mat1copy);
101 gsl_matrix_free (mat2copy);
102 return result;
103 }
```

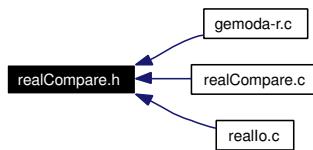
B.18 realCompare.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <gsl/gsl_matrix.h>
#include "realIo.h"
#include "bitSet.h"
#include "protAlign.h"
```

Include dependency graph for realCompare.h:



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



Functions

- double [rmsdCompare](#) ([rdh_t](#) *data, int win1, int win2, int L, double *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 **extraParams*)
- [bitGraph_t](#) * [realComparison](#) (*rdh_t* **data*, int *l*, double *g*, int *compFunc*, double **extraParams*)

Variables

- double(*)(*rdh_t* *, int, int, int, double *) [getCompFunc](#) (int *compFunc*)

Detailed Description

This file contains declarations and definitions used for the comparison of real valued data during the comparison phase of Gemoda. The functions declared here are defined in [realCompare.c](#).

Definition in file [realCompare.h](#).

Function Documentation

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 two 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.18.0.241 `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)`

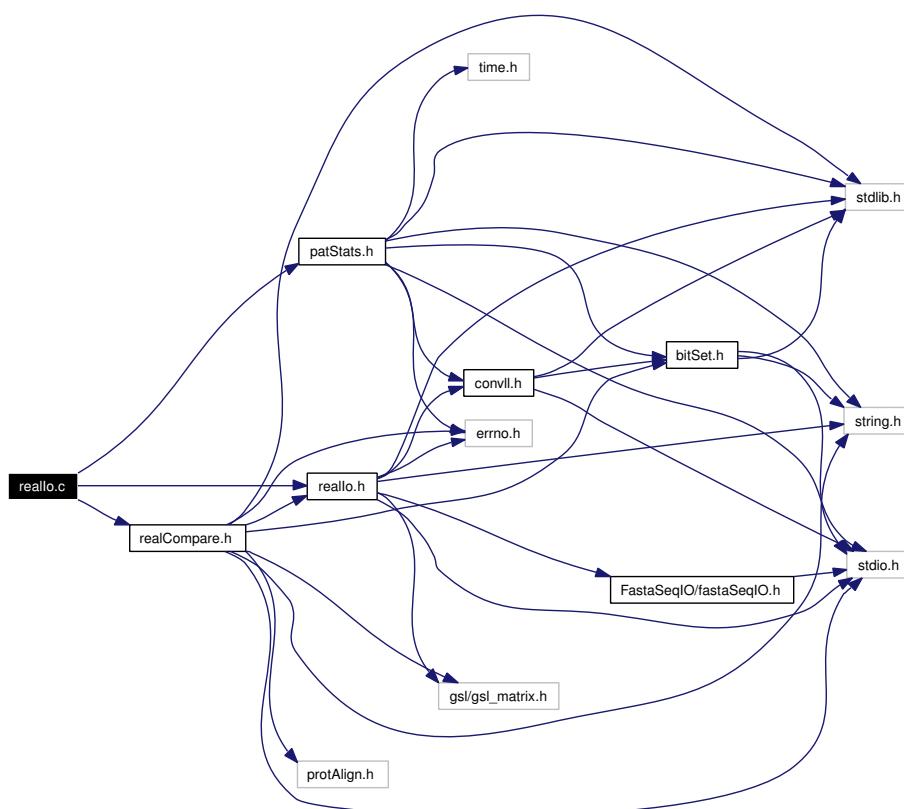
Definition at line 36 of file realCompare.h.

Referenced by findCliqueCentroid(), outputRealPatsWCentroid(), and realComparison().

B.19 realIo.c File Reference

```
#include "realIo.h"
#include "realCompare.h"
#include "patStats.h"
```

Include dependency graph for realIo.c:



Functions

- [wordToDouble](#) (char *s, int begin, int end)
- int [countFields](#) (char *s, char sep)
- int [checkRealDataFormat](#) (char **buf, int nl, char sep, int *numSeq_p, int *dim_p)
- int [countTotalFields](#) (char **buf, int nl, char sep)
- [rdh_t * initRdh](#) (int x)
- int [getRdhSeqLength](#) ([rdh_t](#) *data, int seqNo)
- int [initRdhIndex](#) ([rdh_t](#) *data, int wordSize, int seqGap)
- [rdh_t * freeRdh](#) ([rdh_t](#) *data)

- int `getRdhDim (rdh_t *data)`
- int `setRdhLabel (rdh_t *data, int seqNo, char *s)`
- int `setRdhValue (rdh_t *data, int seqNo, int posNo, int dimNo, double val)`
- int `setRdhIndex (rdh_t *data, int seqNo, int posNo, int index)`
- int `getRdhIndexSeqPos (rdh_t *data, int index, int *seq, int *pos)`
- double `getRdhValue (rdh_t *data, int seqNo, int posNo, int dimNo)`
- char * `getRdhLabel (rdh_t *data, int seqNo)`
- int `printRdhSeq (rdh_t *data, int seqNo, FILE *FH)`
- int `setRdhColFromString (rdh_t *data, int seqNo, int colNo, char *s, char sep)`
- int `initRdhGslMat (rdh_t *data, int seqNo, int x, int y)`
- int `pushOnRdhSeq (rdh_t *data, char **buf, int startLine, int dim, char sep)`
- `rdh_t * parseRealData (char **buf, int nl, char sep, int numSeq, int dim)`
- `rdh_t * readRealData (FILE *INPUT)`
- int `outputRealPats (rdh_t *data, cll_t *allPats, int L, FILE *OUTPUT_FILE, int **d)`
- int `findCliqueCentroid (rdh_t *data, cll_t *curCliq, int L, int compFunc, double *extraParams, int *candidates)`
- int `makeAlternateCentroid (rdh_t *data, cll_t *curCliq, int *candidates)`
- int `outputRealPatsWCCentroid (rdh_t *data, cll_t *allPats, int L, FILE *OUTPUT_FILE, double *extraParams, int compFunc)`

Detailed Description

This file defines functions that are used for the parsing of user supplied data in the real valued implementation of Gemoda.

Definition in file `realIo.c`.

Function Documentation

B.19.o.244 int `checkRealDataFormat (char ** buf, int nl, char sep, int * numSeq_p, int * dim_p)`

Check that each sequence has the same dimensionality and that, within a sequence, each dimension has the same number of entries. Note: this routine alters `*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()`.

```
164 {
165     int i;
166     int thisDim = 0;
167     int status = 1;
168     int width;
169     int fieldCount = 0;           // number of positions in a single sequence
170     int numSeq = 0;             // number of sequences
171     int dim = 0;                // The dimensionality of the sequences
172
173     // NOTE this is not checking the dimensionality of the last sequence...
174     // that's bad. We can fix that though.
175     // Check the dimensionality of each sequence
176     for (i = 0; i < nl; i++)
177     {
178         if (buf[i][0] == '>')
179     {
180
181         // If this is only the second sequence we've seen,
182         // record the dimensionality of the first sequence
183         // as the dim to insist upon from here on out
184         if (numSeq == 1)
185         {
186             dim = thisDim;
187
188             // For other sequences, we need to check to make sure
189             // that they've got the same dimensions as previous
190             // sequences
191         }
192         else if (numSeq > 1)
193         {
194
195             // If the dimensions are wrong, quit with status=0
196             if (thisDim != dim)
197             {
198                 status = 0;
199                 break;
200             }
201         }
202         numSeq++;
203         width = 0;
204         thisDim = 0;
205     }
206     else
207     {
208
209         // Field count can be different for each sequence but
210         // must be the same for each dimension in a single sequence
211         fieldCount = countFields (buf[i], sep);
212
213         // If this is the first row of this sequence,
214         // then store the number of fields
215         if (thisDim == 0)
216         {
217             width = fieldCount;
218
219             // If it's not the first row, make sure it has the
220             // same number of fields as previous rows in this
221             // sequence
222         }
223     else
224     {
225         if (fieldCount != width)
226         {
227             status = 0;
228             break;
229         }
230     }
231     thisDim++;
232 }
```

```

232     }
233     }
234
235     // Pass back the numSeq and dim
236     *numSeq_p = numSeq;
237     *dim_p = thisDim;
238     return status;
239 }
```

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().

```

91 {
92     int i;
93     int begin = 0;
94     int end = 0;
95     int status = 0;          // 0 = in sep, 1 = in word
96     int fieldCount = 0;
97     double val;
98     if (s == NULL)
99     {
100         fprintf (stderr, "Passed NULL string to countFields -- error!");
101         fflush (stderr);
102         exit (0);
103     }
104
105    // Loop over the length of the string
106    for (i = 0; i < strlen (s); i++)
107    {
108
109        // The previous state was space
110        if (status == 0)
111        {
112
113            // We hit a word
114            if (s[i] != sep)
115            {
116                begin = i;
117                status = 1;
118            }
119            else
120            {
121                // We hit more space
122                continue;
123            }
124        }
125        // The previous state was word
126        if (s[i] != sep)
127        {
128            continue;
129        }
130        else
131        {
```

```

132         end = i - 1;
133         status = 0;
134
135         // being and end now delimit a word,
136         // turn that word into a double
137         val = wordToDouble (s, begin, end);
138         fieldCount++;
139     }
140 }
141 }
142
143 // At the end, if we were in a word, we have
144 // one more field
145 if (status == 1)
146 {
    // We're in a word
    val = wordToDouble (s, begin, strlen (s));
    fieldCount++;
}
150 return fieldCount;
151 }
```

B.19.o.246 int countTotalFields (char ** *buf*, int *nl*, char *sep*)

Count the number of fields in each sequence and return the sum of these.

Definition at line 246 of file realIo.c.

References countFields().

Referenced by parseRealData().

```

247 {
248     int i = 0;
249     int totalFields = 0;
250     int seqNo = 0;
251     while (i < nl)
252     {
253
254         // Hit a new sequence
255         if (buf[i][0] == '>')
256     {
257         seqNo++;
258
259         // Assume that the sequence has at least
260         // one row (should have called checkRealDataFormat!
261         // and that each row has the same number of fields
262         totalFields += countFields (buf[i + 1], sep);
263     }
264     i++;
265 }
266 return totalFields;
267 }
```

B.19.o.247 int findCliqueCentroid (*rdh_t* * *data*, *cll_t* * *curCliq*, int *L*, int *compFunc*, double * *extraParams*, int * *candidates*)

This function is used to find the centroid of a clique. That is, to find the center of mass.

Definition at line 1096 of file realIo.c.

References getCompFunc, cSet_t::members, cnode::set, and cSet_t::size.

Referenced by outputRealPatsWCentroid().

```

1098 {
1099     double (*comparisonFunc) (rdh_t *, int, int, int, double *) = NULL;
1100     int i = 0, j = 0, indmin = -1, counter = 0;
1101     double sim = 0, min = 0, flagmin = 0;
1102     double *cliqueAdjMat = NULL;
1103     cliqueAdjMat = (double *) malloc (curCliq->set->size * sizeof (double));
1104     if (cliqueAdjMat == NULL)
1105     {
1106         fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
1107         fflush (stderr);
1108         exit (0);
1109     }
1110     for (i = 0; i < curCliq->set->size; i++)
1111     {
1112         cliqueAdjMat[i] = 0;
1113     }
1114
1115 // We'll accumulate our comparison function values... except here
1116 // we're really assuming that we're using a match factor, with
1117 // value less than one, so that we can subtract it from one to
1118 // get a distance, and then find the centroid by identifying the
1119 // node with the smallest cumulative Euclidean distance to all
1120 // nodes.
1121 // Note that we only need to compare each unique pair, and can apply
1122 // the results from each comparison to each member of the pair,
1123 // hence the somewhat odd indices of initiation for the for loops.
1124 comparisonFunc = getCompFunc (compFunc);
1125 for (i = 0; i < curCliq->set->size; i++)
1126 {
1127     for (j = i + 1; j < curCliq->set->size; j++)
1128     {
1129         sim =
1130         comparisonFunc (data, curCliq->set->members[i],
1131                         curCliq->set->members[j], L, extraParams);
1132
1133         // printf("i = %d, j = %d, L = %d, extra = %lf, sim =
1134         // %lf\n",i,j,L,extraParams[0],sim);
1135         cliqueAdjMat[i] += pow (1 - sim, 2);
1136         cliqueAdjMat[j] += pow (1 - sim, 2);
1137     }
1138 }
1139
1140 // Now we find the minimum Euclidean distance.
1141 min = cliqueAdjMat[0];
1142 indmin = 0;
1143 for (i = 1; i < curCliq->set->size; i++)
1144 {
1145     // printf("index %d product = %lf\n",i,cliqueAdjMat[i]);
1146     if (cliqueAdjMat[i] < min)
1147     {
1148         indmin = i;
1149         min = cliqueAdjMat[i];
1150         flagmin = 0;
1151     }
1152     else if (cliqueAdjMat[i] == min)
1153     {
1154         flagmin = 1;
1155     }
1156 }
1157
1158 // If we had a duplicate on the minimum, we locate all duplicates.
1159 if (flagmin == 1)

```

```

1161     {
1162         counter = 0;
1163         for (i = 0; i < curCliq->set->size; i++)
1164         {
1165             if (cliqueAdjMat[i] == min)
1166             {
1167                 counter++;
1168                 candidates[counter] = i;
1169             }
1170         }
1171         // Store the number of candidates at the array's beginning
1172         candidates[0] = counter;
1173         free (cliqueAdjMat);
1174         return (-1);
1175     }
1176     else
1177     {
1178         free (cliqueAdjMat);
1179         return (indmin);
1180     }
1181 }
1182 }
```

B.19.0.248 **rdh_t*** freeRdh (**rdh_t ****data*)

This function returns a null pointer after freeing the memory associated with a real data holder object. The function takes one parameter: a pointer to the real data holder, *data*.

Definition at line 462 of file realIo.c.

References `rdh_t::indexToPos`, `rdh_t::indexToSeq`, `rdh_t::label`, `rdh_t::offsetToIndex`, and `rdh_t::seq`.

Referenced by `main()`.

```

463 {
464     int i;
465     if (data != NULL)
466     {
467         if (data->indexToPos != NULL)
468         {
469             free (data->indexToPos);
470             data->indexToPos = NULL;
471         }
472         if (data->indexToSeq != NULL)
473         {
474             free (data->indexToSeq);
475             data->indexToSeq = NULL;
476         }
477         if (data->offsetToIndex != NULL)
478         {
479             for (i = 0; i < data->size; i++)
480             {
481                 free (data->offsetToIndex[i]);
482                 data->offsetToIndex[i] = NULL;
483             }
484             free (data->offsetToIndex);
485             data->offsetToIndex = NULL;
486         }
487         for (i = 0; i < data->size; i++)
488         {
489             if (data->seq[i] != NULL)
```

```

490      {
491          gsl_matrix_free (data->seq[i]);
492          data->seq[i] = NULL;
493      }
494      if (data->label[i] != NULL)
495      {
496          free (data->label[i]);
497          data->label[i] = NULL;
498      }
499  }
500  if (data->seq != NULL)
501  {
502      free (data->seq);
503      data->seq = NULL;
504  }
505  if (data->label != NULL)
506  {
507      free (data->label);
508      data->label = NULL;
509  }
510  free (data);
511  data = NULL;
512 }
513 return data;
514 }
```

B.19.o.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.o.250 int getRdhIndexSeqPos ([rdh_t](#) * *data*, int *index*, int * *seq*, int * *pos*)

This function is used to access and change the sequence and position values, given an index. The function takes four parameters: a pointer to the real data holder, *data*, an integer *index*, a pointer integer *seq*, and a pointer integer *pos*.

Definition at line 633 of file realIo.c.

References [rdh_t::indexSize](#), [rdh_t::indexToPos](#), and [rdh_t::indexToSeq](#).

Referenced by generalMatchFactor(), makeAlternateCentroid(), massSpecCompareWElut(), outputRealPats(), outputRealPatsWCentroid(), realComparison(), and rmsdCompare().

```

634 {
635   if (data == NULL || data->indexToSeq == NULL || data->indexToPos == NULL
636   || index > data->indexSize)
637   {
638     fprintf (stderr, "Passed bad data to getRdhIndexSeqPos -- error!");
639     fflush (stderr);
640     exit (0);
641   }
642
643 /*
644   printf("Setting index %d -> %d, %d\n", index, seqNo, posNo);
645 */
646 /*
647   fflush(stdout);
648 */
649 *seq = data->indexToSeq[index];
650 *pos = data->indexToPos[index];
651 return 0;
652 }
```

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 reallo.c.

References *rdh_t::label*.

Referenced by *printRdhSeq()*.

```

690 {
691   if (data == NULL || data->label == NULL || data->label[seqNo] == NULL)
692   {
693     fprintf (stderr, "Passed bad data to getRdhLabel -- error!");
694     fflush (stderr);
695     exit (0);
696   }
697   return data->label[seqNo];
698 }
```

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 reallo.c.

References *rdh_t::seq*.

Referenced by *getRdhValue()*, *initRdhIndex()*, *printRdhSeq()*, and *setRdhValue()*.

```

332 {
333     if (data == NULL || data->seq == NULL || data->seq[seqNo] == NULL)
334     {
335         fprintf (stderr, "Passed bad data to getRdhSeqLength -- error!");
336         fflush (stderr);
337         exit (0);
338     }
339     return data->seq[seqNo]->sizel;
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
282     // Allocate space for our structure
283     data = (rdh_t *) malloc (sizeof (rdh_t));
284     if (data == NULL)
285     {

```

```

286     fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
287     fflush (stderr);
288     exit (0);
289 }
290 data->size = x;
291
292 // Index has to be initialized later, once
293 // we know the word size.
294 data->indexSize = 0;
295 data->indexToSeq = NULL;
296 data->indexToPos = NULL;
297
298 /*
299  data->indexSize = y;
300 */
301 data->label = (char **) malloc (data->size * sizeof (char *));
302 if (data->label == NULL)
303 {
304     fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
305     fflush (stderr);
306     exit (0);
307 }
308 data->seq = (gsl_matrix **) malloc (data->size * sizeof (gsl_matrix *));
309 if (data->seq == NULL)
310 {
311     fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
312     fflush (stderr);
313     exit (0);
314 }
315 for (i = 0; i < data->size; i++)
316 {
317     data->label[i] = NULL;
318     data->seq[i] = NULL;
319 }
320 return data;
321 }
```

B.19.o.255 int initRdhGslMat ([rdh_t](#) * *data*, int *seqNo*, int *x*, int *y*)

This function is used to initialize the memory for the matrix in which the real value to data are stored. To store these data, we use the GNU scientific library. The function takes four parameters: a pointer to the real data holder *data*; an integer, which is the sequence number to be set *seqNo*; an integer, which is the first dimension of the matrix size *x*; and an integer, which is the second dimension of the matrix size *y*;

Definition at line 829 of file realIo.c.

References [rdh_t::seq](#).

Referenced by [pushOnRdhSeq\(\)](#).

```

830 {
831     data->seq[seqNo] = gsl_matrix_alloc (x, y);
832     if (data->seq[seqNo] == NULL)
833     {
834         return 0;
835     }
836     else
837     {
838         return 1;
839     }
840 }
```

B.19.0.256 int initRdhIndex (*rdh_t* * *data*, int *wordSize*, int *seqGap*)

This function is used to initialize the two indices inside a real data holder. The function takes as its input three parameters a pointer to the real data holder, *data*, the size of the words to be compared during the comparison stage *wordSize*, and an integer *seqGap*, which is used to place empty data between unique sequences, such that we do not convolve from one sequence into another during the convolution stage.

Definition at line 358 of file reallo.c.

References `getRdhSeqLength()`, `rdh_t::indexSize`, `rdh_t::indexToPos`, `rdh_t::indexToSeq`, `rdh_t::offsetToIndex`, and `rdh_t::size`.

Referenced by `realComparison()`.

```

359 {
360     int i, j, k;
361     int numWindows = 0;
362     int thisNumWindows;
363     int numSeq;
364     int seqLen = 0;
365
366     // The number of sequences
367     numSeq = data->size;
368
369     // Allocate offsetToIndex's outer structure
370     data->offsetToIndex = (int **) malloc (numSeq * sizeof (int *));
371     if (data->offsetToIndex == NULL)
372     {
373         fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
374         fflush (stderr);
375         exit (0);
376     }
377
378     // For each sequence
379     for (i = 0; i < numSeq; i++)
380     {
381
382         // How many windows are in this sequence
383         seqLen = getRdhSeqLength (data, i);
384         numWindows += seqLen - wordSize + 1;
385
386         // And also use this to further allocate offsetToIndex
387         data->offsetToIndex[i] =
388             (int *) malloc ((seqLen - wordSize + 1) * sizeof (int));
389         if (data->offsetToIndex[i] == NULL)
390         {
391             fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
392             fflush (stderr);
393             exit (0);
394         }
395     }
396
397     // One index for each word plus seqGap between each sequence
398     // and a gap at the end
399     data->indexSize = numWindows + numSeq * seqGap;
400
401     // Allocate indexToSeq
402     // NOTE that it should be size of int, not int *... I think we got
403     // fortunate in the previous revision because they are the same
404     // size
405     data->indexToSeq = (int *) malloc (data->indexSize * sizeof (int));
406     if (data->indexToSeq == NULL)

```

```

407      {
408        fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
409        fflush (stderr);
410        exit (0);
411      }
412
413 // Allocate indexToPos
414 // See above for int vs. int* argument.
415 data->indexToPos = (int *) malloc (data->indexSize * sizeof (int));
416 if (data->indexToPos == NULL)
417 {
418   fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
419   fflush (stderr);
420   exit (0);
421 }
422
423 // Fill in the values
424 k = 0;
425 for (i = 0; i < numSeq; i++)
426 {
427
428   // How many windows are in this sequence?
429   thisNumWindows = getRdhSeqLength (data, i) - wordSize + 1;
430
431   // For each window, make an entry in the indexToSeq
432   // and indexToPos and offsetToIndex
433   for (j = 0; j < thisNumWindows; j++)
434   {
435     data->indexToSeq[k] = i;
436     data->indexToPos[k] = j;
437     data->offsetToIndex[i][j] = k;
438     k++;
439   }
440
441   // Add gaps between sequences in the index.
442   // Usually seqGap is just 1;
443   for (j = 0; j < seqGap; j++)
444   {
445
446     // -1 means no sequence and no position
447     data->indexToSeq[k] = -1;
448     data->indexToPos[k] = -1;
449     k++;
450   }
451 }
452 return 0;
453 }
```

B.19.o.257 **int makeAlternateCentroid (*rdh_t* * *data*, *cll_t* * *curCliq*, *int* * *candidates*)**

This function is used to choose an alternate centroid for a given clique. In order to make the centroid decision slightly less dependent on input order, we decide to choose from the tied candidates the one whose relative position in the sequence is highest. There is no basis in theory for this, it is done so that a consistent choice is made. Only rarely will two spectra be tied for being a centroid and have the same sequence number. In that case, we pretty much have to default to the sequence number, which is what would be done without this function. Note that now though we are less sensitive to the order of input of the sequences, we are now more sensitive to the context surrounding a given spectrum. That is, if it is put in the beginning of the sequence, it is more likely to be chosen. This choice can only be justified insofar as if multiple choices are tied, then they are the same cumulative distance to the clique, and so **any** should

be allowed to be chosen equally. There should be little difference in terms of tangible results. This just makes the semantics consistent.

Definition at line 1202 of file realIo.c.

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

Referenced by outputRealPatsWCentroid().

```

1203 {
1204     int indmin, min, i;
1205     int curSeq, curPos;
1206     int numCandidates = candidates[0];
1207     indmin = candidates[1];
1208     getRdhIndexSeqPos (data, curCliq->set->members[indmin], &curSeq, &curPos);
1209     min = curPos;
1210
1211     // We use less-than-or-equal here because we're starting at 1,
1212     // so we want 1 to end. The length of candidates is one more than
1213     // the maxSup, so we know we can reach candidates[maxSup] without
1214     // a segfault.
1215     for (i = 2; i <= numCandidates; i++)
1216     {
1217         getRdhIndexSeqPos (data, curCliq->set->members[candidates[i]], &curSeq,
1218                             &curPos);
1219         if (curPos < min)
1220         {
1221             indmin = candidates[i];
1222             min = curPos;
1223         }
1224     }
1225     return (indmin);
1226 }
```

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().

```

1048 {
1049     int i, j, pos1;
1050     int curSeq, curPos;
1051     cll_t *curCliq = NULL;
1052     curCliq = allPats;
1053     i = 0;
1054     while (curCliq != NULL)
1055     {
1056         fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d\t", i,
```

```

1057         curCliq->length + L, curCliq->set->size);
1058     if (d != NULL)
1059     {
1060         fprintf (OUTPUT_FILE, "\tsignif=%le\n", curCliq->stat);
1061     }
1062     else
1063     {
1064         fprintf (OUTPUT_FILE, "\n");
1065     }
1066     for (j = 0; j < curCliq->set->size; j++)
1067     {
1068         pos1 = curCliq->set->members[j];
1069         getRdhIndexSeqPos (data, pos1, &curSeq, &curPos);
1070         fprintf (OUTPUT_FILE, " %d\t%d\t", curSeq, curPos);
1071         fprintf (OUTPUT_FILE, "%lf\t",
1072                 gsl_matrix_get (data->seq[curSeq], curPos, 0));
1073
1074         /*
1075             for(k=curPos ; k<curPos+curCliq->length+L ; k++){ fprintf(OUTPUT_FILE, "%c",
1076             mySequences[curSeq].seq[k]); }
1077         */
1078         fprintf (OUTPUT_FILE, "\n");
1079     }
1080     fprintf (OUTPUT_FILE, "\n\n");
1081     curCliq = curCliq->next;
1082     i++;
1083 }
1084 return 0;
1085 }
```

B.19.o.259 int outputRealPatsWCentroid ([rdh_t](#) * *data*, [cll_t](#) * *allPats*, int *L*, [FILE](#) * *OUTPUT_FILE*, double * *extraParams*, int *compFunc*)

This function is used to output real valued patterns in a format such that they are centered on a particular centroid.

Definition at line 1233 of file realIo.c.

References [findCliqueCentroid\(\)](#), [getCompFunc](#), [getRdhIndexSeqPos\(\)](#), [cnode::length](#), [makeAlternateCentroid\(\)](#), [cSet_t::members](#), [cnode::next](#), [cnode::set](#), and [cSet_t::size](#).

Referenced by [main\(\)](#).

```

1236 {
1237     int i, j, k, pos1, centroid;
1238     int curSeq, curPos;
1239     int maxSup = 0;
1240     cll_t *curCliq = NULL;
1241     double mftoCentroid = 0;
1242     double (*comparisonFunc) (rdh_t *, int, int, int, double *) = NULL;
1243     int *candidates = NULL;
1244     curCliq = allPats;
1245     while (curCliq != NULL)
1246     {
1247         if (curCliq->set->size > maxSup)
1248         {
1249             maxSup = curCliq->set->size;
1250         }
1251         curCliq = curCliq->next;
1252     }
1253     candidates = (int *) malloc ((maxSup + 1) * sizeof (int));
```

```

1254     if (candidates == NULL)
1255     {
1256         fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
1257         fflush (stderr);
1258         exit (0);
1259     }
1260     for (i = 0; i <= maxSup; i++)
1261     {
1262         candidates[i] = 0;
1263     }
1264     comparisonFunc = getCompFunc (compFunc);
1265     curCliq = allPats;
1266     i = 0;
1267     while (curCliq != NULL)
1268     {
1269         fprintf (OUTPUT_FILE, "pattern %d:\tlen=%d\tsup=%d\n", i,
1270                 curCliq->length + L, curCliq->set->size);
1271         centroid =
1272         findCliqueCentroid (data, curCliq, L, compFunc, extraParams,
1273                             candidates);
1274         if (centroid < 0)
1275         {
1276             centroid = makeAlternateCentroid (data, curCliq, candidates);
1277
1278             // fprintf(OUTPUT_FILE, "WARNING: No single node in"
1279             // " cluster has non-zero similarity to all other\n nodes"
1280             // " in cluster; centroid set to first node.\n");
1281             // centroid = 0;
1282         }
1283         for (j = 0; j < curCliq->set->size; j++)
1284         {
1285             pos1 = curCliq->set->members[j];
1286             getRdhIndexSeqPos (data, pos1, &curSeq, &curPos);
1287             fprintf (OUTPUT_FILE, "%d\t%d\t", curSeq, curPos);
1288
1289             // fprintf(OUTPUT_FILE, "%lf\t",
1290             // gsl_matrix_get(data->seq[curSeq],curPos,0));
1291             mfToCentroid =
1292             comparisonFunc (data, curCliq->set->members[j],
1293                             curCliq->set->members[centroid], L, extraParams);
1294             fprintf (OUTPUT_FILE, "%lf\t", mfToCentroid);
1295
1296             /*
1297                 for(k=curPos ; k<curPos+curCliq->length+L ; k++){ fprintf(OUTPUT_FILE, "%c",
1298                     mySequences[curSeq].seq[k]); }
1299             */
1300             fprintf (OUTPUT_FILE, "\n");
1301         }
1302         fprintf (OUTPUT_FILE, "\n\n");
1303         curCliq = curCliq->next;
1304         i++;
1305         for (k = 0; k <= maxSup; k++)
1306         {
1307             candidates[k] = 0;
1308         }
1309     }
1310     free (candidates);
1311     return 0;
1312 }
```

B.19.0.260 ***rdh_t**** *parseRealData* (*char ** buf, int nl, char sep, int numSeq, int dim*)

This function is used to parse a single line of a fastA formatted input buffer containing real valued data. The function takes

parameters: a pointer to an array of pointers to characters, which stores the sequences that we will read from *buf*; an integer, which is the line in the buffer on which we should start *nl*; a single character, which is used to delimit the input data *sep*; an integer which is the number of the sequence that we are currently reading in *numSeq*; an integer that is the dimensionality of the input data *dim*;

Definition at line 933 of file realIo.c.

References countTotalFields(), initRdh(), and pushOnRdhSeq().

Referenced by readRealData().

```

934 {
935     int i;
936     int seqNo = -1;
937     int totalNumFields;
938     rdh_t *data = NULL;
939     totalNumFields = countTotalFields (buf, nl, sep);
940
941     /*
942      data = initRdh(numSeq, totalNumFields + numSeq - 1);
943      */
944     data = initRdh (numSeq);
945
946     // We're going to add an empty index between
947     // windows that correspond to different
948     // sequences
949
950     // Fast forward to the first sequence
951     i = 0;
952     while (i < nl)
953     {
954
955         // Hit a new sequence
956         if (buf[i][0] == '>')
957     {
958         seqNo++;          // Note that seqNo started at -1!
959         pushOnRdhSeq (data, buf, i, dim, sep);
960         i += dim + 1;
961     }
962     else
963     {
964         i++;
965     }
966 }
967
968 /*
969     printRdhSeq(data, 0, stdout);
970 */
971 return data;
972 }
```

B.19.0.261 int printRdhSeq (*rdh_t* * *data*, int *seqNo*, FILE * *FH*)

This function is used to print out a real valued data sequence in a pretty manner. The function takes three parameters: a pointer to the real data holder *data*; an integer which is the sequence to be printed out *seqNo*; and a pointer to a file handle which is where the output will be printed *FH*.

Definition at line 710 of file realIo.c.

References `getRdhDim()`, `getRdhLabel()`, `getRdhSeqLength()`, and `getRdhValue()`.

```

711 {
712     int i, j;
713     int len;
714     int dim;
715     len = getRdhSeqLength (data, seqNo);
716     dim = getRdhDim (data);
717     fprintf (FH, "%s\n", getRdhLabel (data, seqNo));
718     for (i = 0; i < len; i++)
719     {
720         for (j = 0; j < dim; j++)
721         {
722             fprintf (FH, "%3.1f ", getRdhValue (data, seqNo, i, j));
723         }
724         fprintf (FH, "\n");
725     }
726     return 0;
727 }
```

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()`.

```

864 {
865     int i, j, k;
866     int numFields;
867
868     // NOTE THAT THESE ARE STATIC VARIABLES!!!!!
869     // That is, they retain their last value on
870     // each call to this function!
871     static int seqNo = 0;
872
873     /*
874         static int indexNo=0;
875     */
876     i = startLine;
877
878     // Assume that the sequence has at least
879     // one row (should have called checkRealDataFormat!
880     numFields = countFields (buf[i + 1], sep);
881
882     // Initialize the gsl_matrix object for this
883     // sequence in 'data'
884     //
```

```

885 // NOTE THAT WE STORE THE TRANPOSE OF WHAT'S IN
886 // THE INPUT FILE -- x,y = position x, dimension y
887 initRdhGslMat (data, seqNo, numFields, dim);
888
889 // Set the sequence label
890 setRdhLabel (data, seqNo, buf[i]);
891
892 // Read in 'dim' rows
893 for (j = i + 1, k = 0; j < i + 1 + dim; j++, k++)
894 {
895
896     /*
897         printf("%d\n", countFields(buf[j], sep));
898     */
899
900     // Set the k-th dimension of this sequence
901     // STILL NOTE THE TRANPOSE!
902     setRdhColFromString (data, seqNo, k, buf[j], sep);
903 }
904
905 /*
906     for ( l=0 ; l<numFields ; l++ ){ setRdhIndex(data, seqNo, l, indexNo); indexNo++;
907 }
908 */
909 seqNo++;
910
911 // Augment indexNo once more to have a -1 between each sequence!
912 /*
913     indexNo++;
914 */
915 return 0;
916 }
```

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 reallo.c.

References [checkRealDataFormat\(\)](#), [parseRealData\(\)](#), and [ReadFile\(\)](#).

Referenced by [main\(\)](#).

```

984 {
985     char **buf = NULL;
986     int nl;
987     int i;
988     char sep = ' ';
989     int numSeq = 0;
990     int dimensions = 0;
991     int status = 1;
992     rdh_t *data = NULL;
993
994     // Read the entire INPUT file and put it's
995     // contents into 'buf'. This function also
996     // alters the contents of the location pointed
997     // to by &nl. Now nl is the number of lines
998     // in the file (or the size of the buff array.
999     buf = ReadFile (INPUT, &nl);
1000    if (buf == NULL)
```

```

1001     {
1002         return NULL;
1003     }
1004     status = checkRealDataFormat (buf, nl, sep, &numSeq, &dimensions);
1005     if (numSeq <= 0 || dimensions <= 0 || status == 0)
1006     {
1007         fprintf (stderr,
1008             "Data file is poorly formatted or no sequences read!\n");
1009         fprintf (stderr,
1010             "Each sequence needs to be the same dimensionality!  QUITTING!\n");
1011         fprintf (stderr, "numSeq = %d, dimensions = %d, status = %d\n",
1012             dimensions, status);
1013         exit (EXIT_FAILURE);
1014     }
1015
1016 // From here on, we assume that the sequence file is well-formatted
1017 // to make the code more simple.
1018 data = parseRealData (buf, nl, sep, numSeq, dimensions);
1019
1020 // Free up our buffer
1021 for (i = 0; i < nl; i++)
1022 {
1023     if (buf[i] != NULL)
1024     {
1025         free (buf[i]);
1026     }
1027 }
1028 if (buf != NULL)
1029 {
1030     free (buf);
1031 }
1032 return data;
1033 }
```

B.19.0.264 int setRdhColFromString (*rdh_t* * *data*, int *seqNo*, int *colNo*, char * *s*, char *sep*)

This function is used to fill in the values of a sequence in a real data holder object by reading them straight from a string, which is assumed to be a series of floating-point values separated by some particular character. The function takes five parameters: a pointer to the real data holder *data*; an integer, which is the sequence number to be set *seqNo*; an integer representing the dimension of the sequence which is to be set *colNo*; a pointer to the string holding the floating-point values *s*; a character, which separates the floating-point values in the string *sep*;

Definition at line 744 of file realIo.c.

References *rdh_t*::*seq*, *setRdhValue()*, and *wordToDouble()*.

Referenced by *pushOnRdhSeq()*.

```

745 {
746     int i;
747     int begin = 0;
748     int end = 0;
749     int status = 0;          // 0 = in sep, 1 = in word
750     int fieldCount = 0;
751     double val;
752
753 // Make sure the string is not null and
754 // the rdh_t gsl_matrix array is not null
```

```

755 // and the selected gsl_matrix is not null
756 if (s == NULL || data->seq == NULL || data->seq[seqNo] == NULL)
757 {
758     fprintf (stderr, "Passed bad data to setRdhColFromString -- error!");
759     fflush (stderr);
760     exit (0);
761 }
762
763 // Loop over the length of the string
764 for (i = 0; i < strlen (s); i++)
765 {
766
767     // The previous state was space
768     if (status == 0)
769     {
770
771         // We hit a word
772         if (s[i] != sep)
773         {
774             begin = i;
775             status = 1;
776         }
777         else
778         {
779             // We hit more space
780             continue;
781         }
782     }
783     else
784     {
785         // The previous state was word
786         if (s[i] != sep)
787         {
788             continue;
789         }
790         else
791         {
792             // We hit a space
793             end = i - 1;
794             status = 0;
795             val = wordToDouble (s, begin, end);
796
797             // Go to the gsl_matrix object data->seq[seqNo]
798             // and set the (fieldCount, colNo) = val;
799             setRdhValue (data, seqNo, fieldCount, colNo, val);
800             fieldCount++;
801         }
802     }
803
804 // At the end, if we were in a word, we have
805 // one more field
806 if (status == 1)
807 {
808     // We're in a word
809     val = wordToDouble (s, begin, strlen (s));
810
811     // Added in, MPS 5/3/05 ---
812     // And don't forget to set the RdhValue!
813     setRdhValue (data, seqNo, fieldCount, colNo, val);
814     fieldCount++;
815 }
816 return fieldCount;
817 }
```

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.

```

601 {
602     if (data == NULL || data->indexToSeq == NULL || data->indexToPos == NULL
603         || index > data->indexSize)
604     {
605         fprintf (stderr, "Passed bad data to getRdhValue -- error!");
606         fflush (stderr);
607         exit (0);
608     }
609
610    /*
611     printf("Setting index %d -> %d, %d\n", index, seqNo, posNo);
612    */
613    /*
614     fflush(stdout);
615    */
616    data->indexToSeq[index] = seqNo;
617    data->indexToPos[index] = posNo;
618    return 0;
619 }
```

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().

```

544 {
545     if (data->seq == NULL || data->label == NULL)
546     {
547         fprintf (stderr, "Passed bad data to setRdhLabel -- error!");
548         fflush (stderr);
549         exit (0);
550     }
551     data->label[seqNo] = strdup (s);
552     if (data->label[seqNo] == NULL)
553     {
554         fprintf (stderr, "\nMemory Error allocating label!\n%s\n",
555                 strerror (errno));
556         fflush (stderr);
557         exit (0);
```

```

558     }
559     return 0;
560 }
```

B.19.o.267 int setRdhValue ([rdh_t](#) * *data*, int *seqNo*, int *posNo*, int *dimNo*, double *val*)

This function will set a particular dimension at a particular position within a specified sequence to a user supplied value. The function takes five parameters: a pointer to the real data holder, *data*, an integer *seqNo* which is the sequence which needs its value set, two integers that specify the position number and the dimension number that needs to be set, and finally a double precision floating point number which is the value to which the the data should be set.

Definition at line 575 of file realIo.c.

References [getRdhDim\(\)](#), [getRdhSeqLength\(\)](#), and [rdh_t::seq](#).

Referenced by [setRdhColFromString\(\)](#).

```

576 {
577     if (data == NULL || data->seq == NULL || data->seq[seqNo] == NULL
578         || posNo > getRdhSeqLength (data, seqNo) || dimNo > getRdhDim (data))
579     {
580         fprintf (stderr, "Passed bad data to setRdhValue -- error!");
581         fflush (stderr);
582         exit (0);
583     }
584     gsl_matrix_set (data->seq[seqNo], posNo, dimNo, val);
585     return 0;
586 }
```

B.19.o.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-formed data files. double

Definition at line 30 of file realIo.c.

Referenced by [countFields\(\)](#), and [setRdhColFromString\(\)](#).

```

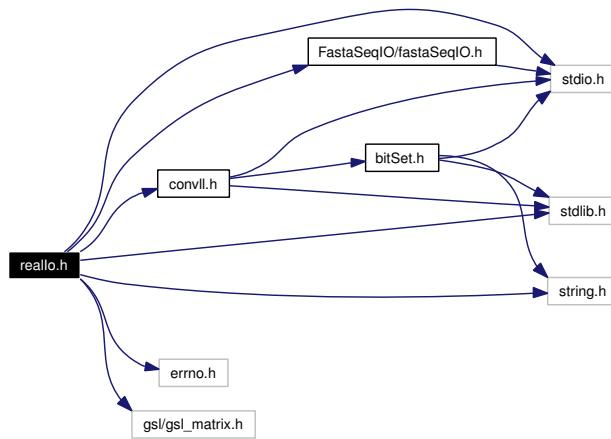
31 {
32     char *str = NULL;
33     char *endptr;
34     double val;
35     int size;
36     int memsize;
37
38     // Check for a sane substring
39     if (end - begin <= 0)
40     {
41         fprintf (stderr, "\nInvalid argument to wordToDouble!\n");
42         fflush (stderr);
43         exit (0);
```

```
44     }
45
46 // Get the required string size
47 memsize = end - begin + 2;      // An extra space in mem for null-termination
48 size = end - begin + 1;
49
50 // Get memory for a temporary string
51 str = (char *) malloc (memsize * sizeof (char));
52 if (str == NULL)
53 {
54     fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
55     fflush (stderr);
56     exit (0);
57 }
58
59 // Make sure the string ends with a null char
60 str[size] = '\0';
61
62 // Copy the word into str
63 str = strncpy (str, s + begin, size);
64
65 // Set endptr to str as initial value
66 endptr = str;
67 val = strtod (str, &endptr);
68
69 // endptr should point to the last char
70 // used in the conversion if strtod worked
71 if (val == 0 && endptr == str)
72 {
73     fprintf (stderr, "\nError making double from string: %s\n", str);
74     fflush (stderr);
75     exit (0);
76 }
77 free (str);
78 return val;
79 }
```

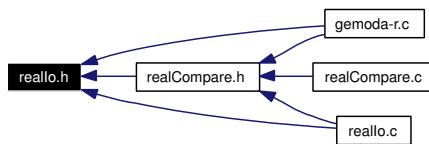
B.20 realIo.h File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <gsl/gsl_matrix.h>
#include "FastaSeqIO/fastaSeqIO.h"
#include "convll.h"
```

Include dependency graph for realIo.h:



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



Data Structures

- struct `rdh_t`

Functions

- `rdh_t * readRealData (FILE *INPUT)`
- `rdh_t * freeRdh (rdh_t *data)`

- int `initRdhIndex` (`rdh_t` **data*, int *wordSize*, int *seqGap*)
- int `getRdhIndexSeqPos` (`rdh_t` **data*, int *index*, int **seq*, int **pos*)
- int `getRdhDim` (`rdh_t` **data*)
- int `outputRealPats` (`rdh_t` **data*, `cll_t` **allPats*, int *L*, FILE **OUTPUT_FILE*, int ***d*)
- int `outputRealPatsWCentroid` (`rdh_t` **data*, `cll_t` **allPats*, int *L*, FILE **OUTPUT_FILE*, double **extraParams*, int *compFunc*)

Function Documentation

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()`.

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 reallo.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 reallo.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 reallo.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*;

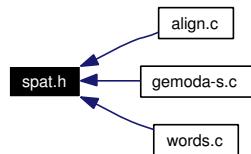
Definition at line 850 of file reallo.c.

References checkRealDataFormat(), parseRealData(), and ReadFile().

Referenced by main().

B.21 spat.h File Reference

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



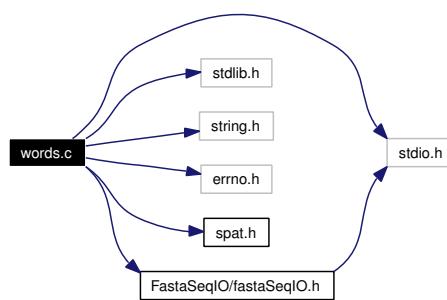
Data Structures

- 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 `hashpjw` (char *s)
- `sHash_t` `initSHash` (int n)
- `sHashEntry_t *` `searchSHash` (`sHashEntry_t` *newEntry, `sHash_t` *thisHash, int create)
- int `destroySHash` (`sHash_t` *thisHash)
- int `printSHash` (`sHash_t` *thisHash, FILE *FH)
- int `printSPats` (`sPat_t` *a, int n)
- int `destroySPatA` (`sPat_t` *words, int wc)

- [sPat_t * countWords2 \(fSeq_t *seq, int numSeq, int L, int *numWords\)](#)

Detailed Description

This file defines functions that are used in the processing of string based sequences. There are a number of functions defined in this file better used for hashing strings so that the comparison phase can be sped up by only comparing unique words. Heuristically, we have noticed that for sequences in which there is a large degree of redundancy these hashing functions can significantly speed up the comparison phase.

Definition in file [words.c](#).

Define Documentation

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\(\)](#).

```

374 {
375     int i, j;
376     int totalChars = 0;
377     int hashSize;
378     sHashEntry_t newEntry;
379     sHashEntry_t *ep;
380     sHash_t wordHash;
381     sPat_t *words = NULL;
382     int wc = 0;
383     int prev = -1;
384     int l;
385
386
387     // Count the total number of characters. This
388     // is the upper limit on how many words we can have
389     for (i = 0; i < numSeq; i++)
390     {

```

```
391     totalChars += strlen (seq[i].seq);
392 }
393
394 // Get a prime number for the size of the hash table
395 hashSize = sieve3 ((long) (2 * totalChars));
396 wordHash = initSHash (hashSize);
397
398 // Chop up each sequence and hash out the words of size L
399 for (i = 0; i < numSeq; i++)
400 {
401     prev = -1;
402
403     // skip sequences that are too short to have
404     // a pattern
405     if (strlen (seq[i].seq) < L)
406     {
407         continue;
408     }
409     for (j = 0; j < strlen (seq[i].seq) - L + 1; j++)
410     {
411
412         // Make a hash table entry for this word
413         newEntry.key = &(seq[i].seq[j]);
414         newEntry.data = 1;
415         newEntry.idx = wc;
416         newEntry.L = L;
417
418         // Check to see if it's already in the hash table
419         ep = searchSHash (&newEntry, &wordHash, 0);
420         if (ep == NULL)
421         {
422
423             // If it's not, create an entry for it
424             ep = searchSHash (&newEntry, &wordHash, 1);
425
426             // Increase the size of our word array
427             words = (sPat_t *) realloc (words, (wc + 1) * sizeof (sPat_t));
428             if (words == NULL)
429             {
430                 fprintf (stderr, "Error!\n");
431                 fflush (stderr);
432             }
433             // Add the new word
434             words[wc].string = &(seq[i].seq[j]);
435             words[wc].length = L;
436             words[wc].support = 1;
437             words[wc].offset =
438             (sOffset_t *) malloc (1 * sizeof (sOffset_t));
439             if (words[wc].offset == NULL)
440             {
441                 fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
442                 fflush (stderr);
443                 exit (0);
444             }
445             words[wc].offset[0].seq = i;
446             words[wc].offset[0].pos = j;
447             words[wc].offset[0].prev = prev;
448             words[wc].offset[0].next = -1;
449
450             if (prev != -1)
451             {
452                 words[prev].offset[words[prev].support - 1].next = wc;
453             }
454             prev = wc;
455             wc++;
456
457         }
458     else
```

```

459     {
460         // 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 }
486 }
487
488
489 destroySHash (&wordHash);
490 *numWords = wc;
491 return words;
492 }
```

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.22.0.279 int destroySPatA ([sPat_t](#) * *words*, int *wc*)

This function is used to free up the memory allocated in an array of [sPat_t](#) space objects. The function returns a null pointer.

Definition at line 352 of file words.c.

References [sPat_t::offset](#).

```
353 {
354     int i;
355     for (i = 0; i < wc; i++)
356     {
357         if (words[i].offset != NULL)
358         {
359             free (words[i].offset);
360             words[i].offset = NULL;
361         }
362     }
363     free (words);
364     words = NULL;
365     return 0;
366 }
```

B.22.0.280 unsigned long hash1 ([unsigned char](#) * *str*)

A hashing function that returns an integer, given a pointer to a null characterterminated string.

Definition at line 73 of file words.c.

Referenced by [searchSHash\(\)](#).

```
74 {
75     unsigned long hash = 5381;
76     int c;
77
78     while ((c = *str++))
79         hash = ((hash << 5) + hash) + c; /* hash * 33 + c */
80
81     return hash;
82 }
```

B.22.0.281 int hashpjw ([char](#) * *s*)

A hashing function that returns an integer, given a pointer to a null characterterminated string.

Definition at line 89 of file words.c.

```
90 {
91     char *p;
92     unsigned int h, g;
93
94     h = 0;
95     for (p = s; *p != '\0'; p++)
96     {
97         h = (h << 4) + *p;
```

```

98     if ((g = h & 0xFFFFFFFF) == 0)
99     {
100         h ^= g >> 24;
101         h ^= g;
102     }
103 }
104 return h;
105 }
```

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().

```

156 {
157     int i = 0;
158     int step = 0;
159     sHash_t this;
160
161     this.totalSize = n;
162     this.hashSize = (int *) malloc (n * sizeof (int));
163     if (this.hashSize == NULL)
164     {
165         fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
166         fflush (stderr);
167         exit (0);
168     }
169     this.iHashSize = (int *) malloc (n * sizeof (int));
170     if (this.iHashSize == NULL)
171     {
172         fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
173         fflush (stderr);
174         exit (0);
175     }
176     this.hash = (sHashEntry_t **) malloc (n * sizeof (sHashEntry_t *));
177     if (this.hash == NULL)
178     {
179         fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
180         fflush (stderr);
181         exit (0);
182     }
183     for (i = 0; i < n; i++)
184     {
185         this.hash[i] = NULL;
186         this.hashSize[i] = 0;
187         this.iHashSize[i] = step;
188     }
189     return this;
190 }
```

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`.

```

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     }
313 }
314     return 0;
315 }
```

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`.

```

322 {
323     char *s = NULL;
324     int i, j;
325     int size = 0;
326     for (i = 0; i < n; i++)
327     {
328         if (a[i].length > size)
329     {
330         s = (char *) realloc (s, a[i].length * sizeof (char));
331     }
332         strncpy (s, a[i].string, a[i].length);
333         s[a[i].length] = '\0';
334         printf ("%d: %s\n", i, s);
335         for (j = 0; j < a[i].support; j++)
336     {
337         printf ("\t%d %d -> (%d, %d)\n", a[i].offset[j].seq,
338             a[i].offset[j].pos, a[i].offset[j].prev,
339             a[i].offset[j].next);
340     }
341         printf ("\n");
342     }
343     free (s);
344     return 0;
345 }
```

B.22.0.285 `sHashEntry_t* searchSHash (sHashEntry_t * newEntry, sHash_t * thisHash, int create)`

This function has two purposes. It searches for entries in the hash table and it puts new entries in.

Definition at line 198 of file words.c.

References `sHash_t::hash`, `hash1()`, `sHash_t::hashSize`, `sHash_t::iHashSize`, `sHashEntry_t::key`, `sHashEntry_t::L`, `SHASH_MAX_KEY_SIZE`, and `sHash_t::totalSize`.

Referenced by `countWords2()`.

```

199 {
200     char string[SHASH_MAX_KEY_SIZE];
201     unsigned long (*hashFunction) () = &hash1;
202     int i, thisIndex;
203     int status = 0;
204
205     // A string to store the key
206     strncpy (string, newEntry->key, newEntry->L);
207     string[newEntry->L] = '\0';
208
209     // The index that this key hashes to
210     thisIndex = hashFunction ((unsigned char *) string) % thisHash->totalSize;
211
212     // For each member that has this index, check to see
213     // if the key is the same
214     for (i = 0; i < thisHash->hashSize[thisIndex]; i++)
215     {
216         if (strcmp (thisHash->hash[thisIndex][i].key, string, newEntry->L) ==
217             0)
218         {
219             // We found a match
220             /*
221                 printf("\t%s already in hash table!\n");
222             */
223             status = 1;
224             return &(thisHash->hash[thisIndex][i]);
225             break;
226         }
227     }
228 }
229 }
230
231 // If we didn't find the key and we're told to create it,
232 // then allocate new memory for the hashEntry and put it in
233 if (status == 0 && create != 0)
234 {
235
236     // Allocate space for the new entry at this index
237     if (thisHash->iHashSize[thisIndex] == 0)
238     {
239         thisHash->hash[thisIndex] =
240             (sHashEntry_t *) malloc (sizeof (sHashEntry_t));
241     }
242     else
243     {
244         thisHash->hash[thisIndex] =
245             (sHashEntry_t *) realloc (thisHash->hash[thisIndex],
246                                     (thisHash->iHashSize[thisIndex] +
247                                         1) * sizeof (sHashEntry_t));
248     }
249     if (thisHash->hash[thisIndex] == NULL)

```

```

250     {
251         fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
252         fflush (stderr);
253         exit (0);
254     }
255     // Increase our record of the size
256     i = thisHash->hashSize[thisIndex];
257     thisHash->hash[thisIndex][i] = *newEntry;
258     thisHash->iHashSize[thisIndex]++;
259     thisHash->hashSize[thisIndex]++;
260
261     // 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 27 of file words.c.

Referenced by countWords2().

```

28 {
29     int i, p, j;
30     int *a;
31     a = (int *) malloc ((n + 1) * sizeof (int));
32     if (a == NULL)
33     {
34         fprintf (stderr, "\nMemory Error\n%s\n", strerror (errno));
35         fflush (stderr);
36         exit (0);
37     }
38     a[0] = 0;
39     a[1] = 0;
40     for (i = 2; i < n; i++)
41     {
42         a[i] = 1;
43     }
44     p = 2;
45     do
46     {
47         j = 2 * p;
48         do
49         {
50             a[j] = 0;
51             j = j + p;
52         }
53         while (j <= n);
54         p = p + 1;
55     }
56     while (p * p < 2 * n);
57     for (i = n; i > 2; i--)
58     {
59         if (a[i])
```

```
60      {
61          free (a);
62          return i;
63      }
64      }
65      free (a);
66      return 0;
67 }
```

Appendix C

Gemoda data structure documentation

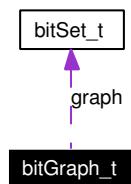
C.1 Introduction

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 $\text{size} \times \text{size}$. 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().

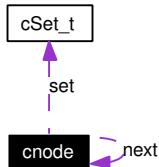
The documentation for this struct was generated from the following file:

- [bitSet.h](#)

C.4 cnode Struct Reference

```
#include <convll.h>
```

Collaboration diagram for cnode:



Data Fields

- `cSet_t * set`
- `int id`
- `int length`
- `cnode * next`
- `double stat`

Detailed Description

This data structure is a linked list for storing cliques. Each member of the linked list has a set, an ID number, a length (which gives the number of characters in the motif), a pointer to the next member of the linked list, and a floating-point number for storing statistical information.

Definition at line 35 of file convll.h.

Field Documentation

C.4.o.293 int cnode::id

Identification number for this member.

Definition at line 38 of file convll.h.

Referenced by `addToStacks()`, `printCll()`, `printCllPattern()`, `pushCll()`, `removeSupers()`, `single-CliqueConv()`, `sortByStats()`, `swapNodecSet()`, `uniqClique()`, `wholeCliqueConv()`, `whole-RoundConv()`, and `yankCll()`.

C.4.o.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.o.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.o.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.o.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(), calcStatCliq(), checkCliquecSet(), findCliqueCentroid(), getLargestSupport(), main(), mllToCSet(), outputRealPats(), outputRealPatsWCentroid(), printCll(), printCllPattern(), printCSet(), pruneCll(), removeSupers(), singleCliqueConv(), uniqClique(), and wholeCliqueConv().

The documentation for this struct was generated from the following file:

- [convll.h](#)

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 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 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 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 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 int rdh_t::size

The number of sequences stored in this data structure.

Definition at line 27 of file realIo.h.

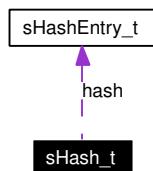
Referenced by initRdh(), initRdhIndex(), and main().

The documentation for this struct was generated from the following file:

- [realIo.h](#)

C.9 sHash_t Struct Reference

Collaboration diagram for sHash_t:



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.o.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.o.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 countWords2(), printSHash(), and searchSHash().

The documentation for this struct was generated from the following file:

- [words.c](#)

C.11 sOffset_t Struct Reference

```
#include <spat.h>
```

Data Fields

- int `seq`
- int `pos`
- int `next`
- int `prev`

Detailed Description

This object is used to store the location of a particular word and a set of sequences. That is if we hash a word, we would like to know where it came from. This data structure provides that information.

Definition at line 13 of file spat.h.

Field Documentation

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().

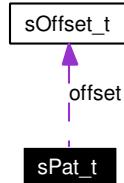
The documentation for this struct was generated from the following file:

- [spat.h](#)

C.12 sPat_t Struct Reference

```
#include <spat.h>
```

Collaboration diagram for sPat_t:



Data Fields

- char * [string](#)
- int [length](#)
- int [support](#)
- [sOffset_t](#) * [offset](#)

Detailed Description

This data structure is used to store the locations of all the instances of a particular word of length *length* in a set of sequences. This data structure is used principally by the string based version of Gemoda and is used to store words that are hashed before the comparison phase.

Definition at line 36 of file spat.h.

Field Documentation

C.12.o.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.o.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

- [1] *GNU Scientific Library Reference Manual - Second Edition*. Network Theory Ltd., 2003.
[126](#), [225](#)
- [2] W. T. Adams and T. R. Skopek. Statistical test for the comparison of samples from mutational spectra. *J Mol Biol*, 194(3):391–6, Apr 1987. [201](#)
- [3] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume I: Parsing of *Series in Automatic Computation*. Prentice Hall, Englewood Cliffs, New Jersey, 1979. ISBN 0-13-914556-7. [32](#)
- [4] N. N. Alexandrov. SARFing the PDB. *Protein Eng*, 9(9):727–732, Oct 1996. [140](#)
- [5] N. N. Alexandrov and D. Fischer. Analysis of topological and nontopological structural similarities in the PDB: new examples with old structures. *Proteins*, 25(3):354–365, Aug 1996. [140](#)
- [6] F. Alimoglu and E. Alpaydin. Methods of combining multiple classifiers based on different representations for pen-based handwriting recognition. In *Proceedings of the Fifth Turkish Artificial Intelligence and Artificial Neural Networks Symposium*, Istanbul, Turkey, 1996. [184](#)
- [7] F. Alimoglu and E. Alpaydin. Combining multiple representations and classifiers for pen-based handwritten digit recognition. In *Proceedings of the Fourth International Conference on Document Analysis and Recognition*, Ulm, Germany, 1997. [184](#)
- [8] H. Alper, C. Fischer, E. Nevoigt, and G. Stephanopoulos. Tuning genetic control

- through promoter engineering. *Proc Natl Acad Sci USA*, 102(36):12678–83, Sep 2005. [201](#), [203](#), [204](#)
- [9] S. F. Altschul. Amino acid substitution matrices from an information theoretic perspective. *J Mol Biol*, 219:555–65, 1991. [168](#), [186](#)
- [10] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215:403–10, 1990. [160](#), [163](#), [167](#), [177](#)
- [11] S. F. Altschul, T. L. Madden, A. A. S. Zhang, J., Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res*, 25:3389–402, 1997. [89](#), [155](#), [177](#)
- [12] D. Amsterdam. *Susceptibility testing of antimicrobials in liquid media*, pages 52–111. Antibiotics in Laboratory Medicine. Williams & Wilkins, Baltimore, MD, 4th edition, 1996. [92](#)
- [13] L. Aravind and E. V. Koonin. The HD domain defines a new superfamily of metal-dependent phosphohydrolases. *Trends Biochem Sci*, 23(12):469–472, 1998. [138](#)
- [14] P. Argos, J. K. Rao, and P. A. Hargrave. Structural prediction of membrane-bound proteins. *Eur J Biochem*, 128(2-3):565–575, Nov. 1982. [117](#)
- [15] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-d point sets. *IEEE Trans. Pattern Anal. Mach. Intell.*, 9(5):698–700, 1987. ISSN 0162-8828. [140](#)
- [16] W. R. Atchley, J. Zhao, A. D. Fernandes, and T. Drüke. Solving the protein sequence metric problem. *Proc Natl Acad Sci USA*, 102(18):6395–400, May 2005. [189](#)
- [17] T. K. Attwood, P. Bradley, D. R. Flower, A. Gaulton, N. Maudling, A. L. Mitchell, G. Moulton, A. Nordle, K. Paine, P. Taylor, A. Uddin, and C. Zygouri. PRINTS and its automatic supplement, prePRINTS. *Nucleic Acids Res*, 31:400–2, 2003. [160](#)
- [18] P. Bagossi, T. Sperka, A. Fehér, J. Kádas, G. Zahuczky, G. Miklóssy, P. Boross, and J. Tözsér. Amino acid preferences for a critical substrate binding subsite of retroviral

- proteases in type 1 cleavage sites. *J Virol*, 79(7):4213–4218, Apr 2005. URL <http://www.ncbi.nlm.nih.gov/pubmed/15767422>. 190
- [19] T. L. Bailey and C. Elkan. Fitting a mixture model by expectation maximization to discover motifs in biopolymers. *Proc Int Conf Intell Syst Mol Biol*, 2:28–36, 1994. 67, 112, 148, 156
- [20] A. Bairoch. Prosite: a dictionary of sites and patterns in proteins. *Nucleic Acids Res*, 19 Suppl:2241–2245, Apr 1991. URL <http://www.ncbi.nlm.nih.gov/pubmed/164>?uids=2041810. 164
- [21] A. Bairoch. The ENZYME database in 2000. *Nucleic Acids Res*, 28(1):304–305, Feb 2000. 136, 137
- [22] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL in 2000. *Nucleic Acids Res*, 28(1):45–48, Jan 2000. 77, 138, 161, 177
- [23] A. Bairoch and B. Boeckmann. The swiss-prot protein sequence data bank. *Nucleic Acids Res*, 20 Suppl:2019–2022, May 1992. URL <http://www.ncbi.nlm.nih.gov/pubmed/164>?uids=1598233. 164
- [24] C. Barillas-Mury, B. Wizel, and Y. S. Han. Mosquito immune responses and malaria transmission: lessons from insect model systems and implications for vertebrate innate immunity and vaccine development. *Insect Biochem Mol Biol*, 30(6):429–442, June 2000. 73
- [25] A. Bateman, L. Coin, R. Durbin, R. D. Finn, V. Hollich, S. Griffiths-Jones, A. Khanna, M. Marshall, S. Moxon, E. L. L. Sonnhammer, D. J. Studholme, C. Yeats, and S. R. Eddy. The Pfam protein families database. *Nucleic Acids Res*, 32 Database issue:138–141, Feb 2004. 138
- [26] Z. Beck, L. Hervio, P. Dawson, J. Elder, and E. Madison. Identification of efficiently cleaved substrates for HIV-1 protease using a phage display library and use in inhibitor

- development. *Virology*, 274(2):391–401, Sep 2000. URL <http://www.hubmed.org/display.cgi?uids=10964781>. 190
- [27] Z. Beck, Y. Lin, and J. Elder. Molecular basis for the relative substrate specificity of human immunodeficiency virus type 1 and feline immunodeficiency virus proteases. *J Virol*, 75(19):9458–9469, Oct 2001. URL <http://www.hubmed.org/display.cgi?uids=11533208>. 190
- [28] Z. Beck, G. Morris, and J. Elder. Defining HIV-1 protease substrate selectivity. *Curr Drug Targets Infect Disord*, 2(1):37–50, Mar 2002. URL <http://www.hubmed.org/display.cgi?uids=12462152>. 190
- [29] G. Bell and P.-H. Gouyon. Arming the enemy: the evolution of resistance to self-proteins. *Microbiology*, 149(Pt 6):1367–1375, Jun 2003. 109
- [30] K. Bennett and E. Bredensteiner. Duality and geometry in svms. In P. Langley, editor, *Proc. of 17th international Conference on Machine Learning*, pages 65–72. Morgan Kaufmann, 2000. 195
- [31] K. P. Bennett and C. Campbell. Support vector machines: Hype or hallelujah? *SIGKDD Explorations*, 2(2):1–13, 2000. URL <http://citeseer.ist.psu.edu/bennett03support.html>. 194, 195
- [32] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman , J. Ostell, B. A. Rapp, and D. L. Wheeler. GenBank. *Nucleic Acids Res*, 28:15–8, 2000. 177
- [33] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, and D. L. Wheeler. GenBank. *Nucleic Acids Res*, 34(Database issue):D16–20, Jan 2006. 19, 21
- [34] H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The Protein Data Bank. *Nucleic Acids Res*, 28(1):235–242, Feb 2000. 15, 190
- [35] C. Bi and P. K. Rogan. Bipartite pattern discovery by entropy minimization-based multiple local alignment. *Nucleic Acids Res*, 32(17):4979–91, 2004. 67

- [36] C. C. Bigelow. On the average hydrophobicity of proteins and the relation between it and protein structure. *J Theor Biol*, 16(2):187–211, Aug 1967. 189
- [37] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002. ISSN 0098-3500. URL <http://doi.acm.org/10.1145/567806.567807>. 126, 225
- [38] C. L. Blake and C. J. Merz. UCI repository of machine learning databases, 1998. <http://www.ics.uci.edu/~mlearn/MLRepository.html>. 178, 182
- [39] D. Boden and M. Markowitz. Resistance to human immunodeficiency virus type 1 protease inhibitors. *Antimicrob Agents Chemother*, 42(11):2775–2783, Nov 1998. URL <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC19203/>. 190
- [40] H. G. Boman. Antibacterial peptides: basic facts and emerging concepts. *J Intern Med*, 254(3):197–215, Sep 2003. 72
- [41] Brazma, Jonassen, Vilo, and Ukkonen. Pattern discovery in biosequences. In *ICGI: International Colloquium on Grammatical Inference and Applications*, 1998. URL citeseer.nj.nec.com/brazma98pattern.html. 58
- [42] S. E. Brenner, P. Koehl, and M. Levitt. The ASTRAL compendium for protein structure and sequence analysis. *Nucleic Acids Res*, 28(1):254–6, Jan 2000. 164, 166
- [43] J. Bresnan, R. M. Kaplan, S. Peters, and A. Zaenen. Cross-serial dependencies in Dutch. *Linguistic Inquiry*, 13(4):613–635, 1982. Reprinted in W. Savitch et al. (eds) *The Formal Complexity of Natural Language*, 286–319. Dordrecht: D. Reidel. 40
- [44] A. Brik and C. Wong. HIV-1 protease: mechanism and drug discovery. *Org Biomol Chem*, 1(1):5–14, Jan 2003. URL <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC12929379/>. 190

- [45] J. Buhler and M. Tompa. Finding motifs using random projections. In *Proceedings of the fifth annual international conference on Computational biology*, pages 69–76. ACM Press, 2001. ISBN 1-58113-353-7. 112, 146, 147
- [46] J. Buhler and M. Tompa. Finding motifs using random projections. *J Comput Biol*, 9(2):225–42, 2002. 59, 147, 157
- [47] H. J. Bussemaker, H. Li, and E. D. Siggia. Building a dictionary for genomes: identification of presumptive regulatory sites by statistical analysis. *Proc Natl Acad Sci U S A*, 97(18):10096–100, Aug 2000. 59
- [48] Y.-D. Cai, X.-J. Liu, X.-B. Xu, and K.-C. Chou. Support Vector Machines for predicting HIV protease cleavage sites in protein. *J Comput Chem*, 23(2):267–274, Jan 2002. 190
- [49] Y. D. Cai, H. Yu, and K. C. Chou. Artificial neural network method for predicting HIV protease cleavage sites in protein. *J Protein Chem*, 17(7):607–15, Oct 1998. 190
- [50] G. H. Cassell and J. Mekalanos. Development of antimicrobial agents in the era of new and reemerging infectious diseases and increasing antibiotic resistance. *JAMA*, 285:601–5, 2001. 72
- [51] P. Casteels and P. Tempst. Apidaecin-type peptide antibiotics function through a non-poreforming mechanism involving stereospecificity. *Biochem Biophys Res Commun*, 199(1):339–345, Feb 1994. 94
- [52] Y. Chen, X. Xu, S. Hong, J. Chen, N. Liu, C. B. Underhill, K. Creswell, and L. Zhang. RGD-Tachyplesin inhibits tumor growth. *Cancer Res*, 61(6):2434–2438, Mar 2001. 73
- [53] N. Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956. 31, 38
- [54] N. Chomsky. *Syntactic Structures*. Mouton and Co., The Hague, 1957. 31
- [55] N. Chomsky. *Aspects of the theory of syntax*. MIT Press, Cambridge, Massachusetts, 1965. 31

- [56] K. C. Chou. Prediction of human immunodeficiency virus protease cleavage sites in proteins. *Anal Biochem*, 233(1):1–14, Jan 1996. [190](#)
- [57] P. Y. Chou and G. D. Fasman. Prediction of the secondary structure of proteins from their amino acid sequence. *Adv Enzymol Relat Areas Mol Biol*, 47:45–148, 1978. [189](#)
- [58] G. K. Christophides, E. Zdobnov, C. Barillas-Mury, E. Birney, S. Blandin, C. Blass, P. T. Brey, F. H. Collins, A. Danielli, G. Dimopoulos, C. Hetru, N. T. Hoa, J. A. Hoffmann, S. M. Kanzok, I. Letunic, E. A. Levashina, T. G. Loukeris, G. Lycett, S. Meister, K. Michel, L. F. Moita, H.-M. Muller, M. A. Osta, S. M. Paskewitz, J.-M. Reichhart, A. Rzhetsky, L. Troxler, K. D. Vernick, D. Vlachou, J. Volz, C. von Mering., J. Xu, L. Zheng, P. Bork, and F. C. Kafatos. Immunity-related genes and gene families in *Anopheles gambiae*. *Science*, 298(5591):159–165, Oct. 2002. [73](#)
- [59] P. C. Cirino, K. M. Mayer, and D. Umeno. Generating mutant libraries using error-prone PCR. *Methods Mol Biol*, 231:3–9, 2003. [204](#)
- [60] J. C. Clemente, R. E. Moose, R. Hemrajani, L. R. S. Whitford, L. Govindasamy, R. Reutzel, R. McKenna, M. Agbandje-McKenna, M. M. Goodenow, and B. M. Dunn. Comparing the accumulation of active- and nonactive-site mutations in the HIV-1 protease. *Biochemistry*, 43(38):12141–51, Sep 2004. [190](#)
- [61] F. S. Collins, M. Morgan, and A. Patrinos. The Human Genome Project: lessons from large-scale biology. *Science*, 300(5617):286–90, Apr 2003. [20](#)
- [62] D. J. Crisp and C. J. C. Burges. A geometric interpretation of v-svm classifiers. In *NIPS*, pages 244–250, 1999. [195](#)
- [63] N. Cristianini and J. Shawe-Taylor. *An introduction to support Vector Machines: and other kernel-based learning methods*. Cambridge University Press, New York, NY, USA, 2000. ISBN 0-521-78019-5. [195](#)
- [64] L. V. Danilova, V. A. Lyubetsky, and M. S. Gelfand. An algorithm for identification of regulatory signals in unaligned DNA sequences, its testing and parallel implementation. *In Silico Biol*, 3(1–2):33–47, 2003. [59](#)

- [65] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. In M. O. Dayhoff, editor, *Atlas of Protein Structure*, volume 5(Suppl. 3), pages 345–352. National Biomedical Research Foundation, Silver Spring, Md., 1978. 163, 186, 188, 189
- [66] S. Dietmann and L. Holm. Identification of homology in protein structure classification. *Nat Struct Biol*, 8(11):953–957, Nov 2001. 138
- [67] T. G. Dietterich and G. Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995. 185
- [68] J. Dongarra. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard I. *The International Journal of High Performance Computing Applications*, 16(1):1–111, Spring 2002. ISSN 1094-3420. 126, 225
- [69] J. Dongarra. Preface: Basic Linear Algebra Subprograms Technical (Blast) Forum Standard II. *The International Journal of High Performance Computing Applications*, 16(2):115–199, Summer 2002. 126, 225
- [70] T. A. Down and T. J. P. Hubbard. NestedMICA: sensitive inference of over-represented motifs in nucleic acid sequence. *Nucleic Acids Res*, 33(5):1445–53, 2005. 67
- [71] M. Dsouza, N. Larsen, and R. Overbeek. Searching for patterns in genomic data. *Trends Genet*, 13:497–8, 1997. 160
- [72] S. R. Eddy. Profile hidden Markov models. *Bioinformatics*, 14(9):755–763, 1998. 125
- [73] I. Eidhammer, I. Jonassen, and W. R. Taylor. Structure comparison and structure patterns. *J Comput Biol*, 7(5):685–716, 2000. 138
- [74] H. M. Ellerby, W. Arap, L. M. Ellerby, R. Kain, R. Andrusiak, G. D. Rio, S. Krajewski, C. R. Lombardo, R. Rao, E. Ruoslahti, D. E. Bredesen, and R. Pasqualini. Anti-cancer activity of targeted pro-apoptotic peptides. *Nat Med*, 5(9):1032–1038, Sep 1999. 73
- [75] R. M. Epand and H. J. Vogel. Diversity of antimicrobial peptides and their mechanisms of action. *Biochim Biophys Acta*, 1462:11–28, 1999. 73, 74

- [76] Eric Sayers and David Wheeler. Building Customized Data Pipelines Using the Entrez Programming Utilities. <http://www.ncbi.nlm.nih.gov/books/>, Accessed in Dec 2005. 22
- [77] E. Eskin. From profiles to patterns and back again: a branch and bound algorithm for finding near optimal motif profiles. In *RECOMB '04: Proceedings of the eighth annual international conference on Resaerch in computational molecular biology*, pages 115–124, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-755-9. 66, 67
- [78] E. Eskin and P. A. Pevzner. Finding composite regulatory patterns in DNA sequences. *Bioinformatics*, 18 Suppl 1:354–363, 2002. Evaluation Studies. 59, 112, 155
- [79] G. Fasman, editor. *Physical Chemical Data*, volume 1 of *CRC Handbook of Biochemistry and Molecular Biology*. CRC Press, Cleveland, Ohio, 1976. 192
- [80] J. L. Fauchère, M. Charton, L. B. Kier, A. Verloop, and V. Pliska. Amino acid side chain parameters for correlation studies in biology and pharmacology. *Int J Pept Protein Res*, 32(4):269–78, Oct 1988. 192
- [81] J. Felsenstein. Phylogeny inference package (version 3.2). *Cladistics*, 5:164–166, 1989. 181
- [82] A. Floratos. *Pattern Discovery in Biology: Theory and Applications*. PhD thesis, New York University, New York, Jan. 1999. 58, 59, 60
- [83] G. B. Fogel, D. G. Weekes, G. Varga, E. R. Dow, H. B. Harlow, J. E. Onyia, and C. Su. Discovery of sequence motifs related to coexpression of genes using evolutionary computation. *Nucleic Acids Res*, 32(13):3826–35, 2004. 59
- [84] J. E. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Inc., Sebastopol, California, 1997. 44
- [85] N. Friedman, M. Linial, I. Nachman, and D. Pe'er. Using bayesian networks to analyze expression data. In *4th Annual International Conference on Computational Molecular*

- Biology (RECOMB 2000)*, pages 127–135, Apr 2000. URL citeseer.ist.psu.edu/friedman99using.html. 194
- [86] M. C. Frith, U. Hansen, J. L. Spouge, and Z. Weng. Finding functional sequence elements by multiple local alignment. *Nucleic Acids Res*, 32(1):189–200, 2004. 59
- [87] D. J. Galas, M. Eggert, and M. S. Waterman. Rigorous pattern-recognition methods for DNA sequences. Analysis of promoter sequences from *Escherichia coli*. *J Mol Biol*, 186(1):117–28, Nov 1985. 59
- [88] R. Ganesh, D. A. Siegelle, and T. R. Ioerger. Mopac: motif finding by preprocessing and agglomerative clustering from microarrays. *Pac Symp Biocomput*, pages 41–52, 2003. URL <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC12603016/>. 59
- [89] T. Ganz. Defensins: antimicrobial peptides of innate immunity. *Nat Rev Immunol*, 3:710–20, 2003. 72
- [90] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979. 58, 60, 124
- [91] Y. Ge, D. L. MacDonald, K. J. Holroyd, C. Thornsberry, H. Wexler, and M. Zasloff. In vitro antibacterial properties of pexiganan, an analog of magainin. *Antimicrob Agents Chemother*, 43(4):782–788, Apr. 1999. 72, 77
- [92] A. Gelman and D. B. Rubin. Inference from iterative simulation using multiple sequences. *Statistical Science*, 7:457–472, 1992. 68
- [93] Gerstein Laboratory, Yale University. Omes Table. <http://bioinfo.mbb.yale.edu/what-is-it/omes/omes.html>, Dec 2005. 22
- [94] A. Giangaspero, L. Sandri, and A. Tossi. Amphipathic alpha helical antimicrobial peptides. *Eur J Biochem*, 268:5589–600, 2001. 73
- [95] H. Giladi, D. Goldenberg, S. Koby, and A. B. Oppenheim. Enhanced activity of the bacteriophage lambda PL promoter at low temperature. *Proc Natl Acad Sci U S A*, 92(6):2184–8, Mar 1995. 204, 205

- [96] H. Giladi, S. Koby, G. Prag, M. Engelhorn, J. Geiselmann, and A. B. Oppenheim. Participation of IHF and a distant UP element in the stimulation of the phage lambda PL promoter. *Mol Microbiol*, 30(2):443–51, Oct 1998. [204](#), [205](#)
- [97] H. Giladi, K. Murakami, A. Ishihama, and A. B. Oppenheim. Identification of an UP element within the IHF binding site at the PL₁-PL₂ tandem promoter of bacteriophage lambda. *J Mol Biol*, 260(4):484–91, Jul 1996. [204](#), [205](#)
- [98] A. Glieder, E. T. Farinas, and F. H. Arnold. Laboratory evolution of a soluble, self-sufficient, highly active alkane hydroxylase. *Nat Biotechnol*, 20(11):1135–9, Nov 2002. [200](#)
- [99] A. Goffeau. Genomic-scale analysis goes upstream? *Nat Biotechnol*, 16(10):907–8, Oct 1998. [67](#)
- [100] M. Gribskov and N. L. Robinson. Use of receiver operating characteristic (ROC) analysis to evaluate sequence matching. *Computers in Chemistry*, 20(1):25–33, 1996. [166](#)
- [101] D. GuhaThakurta and G. D. Stormo. Identifying target sites for cooperatively binding factors. *Bioinformatics*, 17(7):608–21, Jul 2001. [67](#)
- [102] R. E. Hancock and A. Patrzykat. Clinical development of cationic antimicrobial peptides: from natural to novel antibiotics. *Curr Drug Targets Infect Disord*, 2:79–83, 2002. [77](#)
- [103] D. J. Hand and K. Yu. Idiot's bayes – not so stupid after all? *International Statistical Review*, 69(3):385–399, 2001. [194](#)
- [104] K. A. Hargreaves and K. Berry. Regex. Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, Sept. 1992. [160](#)
- [105] A. J. Hartemink, D. K. Gifford, T. Jaakkola, and R. A. Young. Bayesian methods for elucidating genetic regulatory networks. *IEEE Intelligent Systems*, 17(2):37–43, 2002. [194](#)

- [106] D. Heckerman. A tutorial on learning with bayesian networks, 1995. URL citeseer.ist.psu.edu/heckerman96tutorial.html. 194
- [107] J. G. Henikoff. Fred Hutchinson Cancer Research Center. Personal communication, October 2005. 174
- [108] S. Henikoff and J. G. Henikoff. Automated assembly of protein blocks for database searching. *Nucleic Acids Res*, 19:6565–72, 1991. 160
- [109] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc Natl Acad Sci U S A*, 89(22):10915–10919, Nov 1992. 89, 136, 155, 163, 164, 168, 170, 171, 176, 183, 186, 188, 189
- [110] S. Henikoff and J. G. Henikoff. Performance evaluation of amino acid substitution matrices. *Proteins*, 17:49–61, 1993. 163, 164, 167, 170, 171, 186
- [111] S. Henikoff and J. G. Henikoff. Protein family classification based on searching a database of blocks. *Genomics*, 19(1):97–107, Jan 1994. URL <http://www.ncbi.nlm.nih.gov/pubmed/8188249>. 163
- [112] S. Henikoff and J. G. Henikoff. *Amino Acid Substitution Matrices*, volume 54 of *Advances in Protein Chemistry*, pages 73–98. Academic Press, San Diego, 2000. 163, 186
- [113] S. Henikoff, J. G. Henikoff, W. J. Alford, and S. Pietrokovski. Automated construction and graphical presentation of protein blocks from unaligned sequences. *Gene*, 163(2):GC17–26, Oct 1995. 112, 155, 156
- [114] D. Hernandez, R. Gras, and R. Appel. MoDEL: an efficient strategy for ungapped local multiple alignment. *Comput Biol Chem*, 28(2):119–28, Apr 2004. 59
- [115] G. Z. Hertz and G. D. Stormo. Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. *Bioinformatics*, 15(7-8):563–577, Jul 1999. 67, 112, 148, 156
- [116] D. G. Higgins, A. J. Bleasby, and R. Fuchs. CLUSTAL V: improved software for multiple sequence alignment. *Comput Appl Biosci*, 8:189–91, 1992. 181

- [117] K. Hilpert, R. Volkmer-Engert, T. Walter, and R. E. W. Hancock. High-throughput generation of small antibacterial peptides with improved activity. *Nat Biotechnol*, 23(8):1008–12, Aug 2005. [109](#)
- [118] K. Hofmann, P. Bucher, L. Falquet, and A. Bairoch. The PROSITE database, its status in 1999. *Nucleic Acids Res*, 27:215–9, 1999. [45](#), [77](#), [125](#), [160](#), [163](#)
- [119] L. Holm, C. Ouzounis, C. Sander, G. Tuparev, and G. Vriend. A database of protein structure families with common folding motifs. *Protein Sci*, 1(12):1691–1698, 1992. [112](#)
- [120] L. Holm and C. Sander. Protein structure comparison by alignment of distance matrices. *J Mol Biol*, 233(1):123–138, Oct 1993. [138](#), [155](#)
- [121] L. Holm and C. Sander. Enzyme HIT. *Trends Biochem Sci*, 22(4):116–117, May 1997. Letter. [14](#), [140](#), [144](#)
- [122] B. K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America A*, 4(4):629–642, Apr 1987. [140](#)
- [123] P. Horton. Tsukuba BB: a branch and bound algorithm for local multiple alignment of DNA and protein sequences. *J Comput Biol*, 8(3):283–303, 2001. [59](#)
- [124] J. D. Hughes, P. W. Estep, S. Tavazoie, and G. M. Church. Computational identification of cis-regulatory elements associated with groups of functionally related genes in *Saccharomyces cerevisiae*. *J Mol Biol*, 296:1205–14, 2000. [67](#)
- [125] C. G. Hunter and S. Subramaniam. Protein fragment clustering and canonical local shapes. *Proteins*, 50(4):580–588, Apr 2003. Evaluation Studies. [140](#)
- [126] D. Hwang, A. G. Rust, S. Ramsey, J. J. Smith, D. M. Leslie, A. D. Weston, P. de Atauri, J. D. Aitchison, L. Hood, A. F. Siegel, and H. Bolouri. A data integration methodology for systems biology. *Proc Natl Acad Sci U S A*, 102(48):17296–301, Nov 2005. [25](#)
- [127] T. Ideker and D. Lauffenburger. Building with a scaffold: emerging strategies for high-to low-level cellular modeling. *Trends Biotechnol*, 21(6):255–62, Jun 2003. [25](#)

- [128] J. S. Jacobs Anderson and R. Parker. Computational identification of cis-acting elements affecting post-transcriptional control of gene expression in *Saccharomyces cerevisiae*. *Nucleic Acids Res*, 28(7):1604–17, Apr 2000. [59](#)
- [129] K. L. Jensen, M. P. Styczynski, and G. N. Stephanopoulos. All of your blast searches are wrong... sort of. *Bioinformatics*, page submitted, 2006. [164](#)
- [130] G. H. John and P. Langley. Estimating continuous distributions in Bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, 2005. URL citeseer.ist.psu.edu/john95estimating.html. [194](#)
- [131] M. Johnson, S. Morris, A. Chen, E. Stavnezer, and J. Leis. Selection of functional mutations in the U₅-IR stem and loop regions of the Rous sarcoma virus genome. *BMC Biol*, 2(1):8, May 2004. [201](#)
- [132] I. Jonassen, J. F. Collins, and D. G. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Sci*, 4(8):1587–1595, Aug 1995. [59](#), [112](#)
- [133] I. Jonassen, I. Eidhammer, D. Conklin, and W. R. Taylor. Structure motif discovery and mining the PDB. *Bioinformatics*, 18(2):362–367, Feb 2002. [140](#)
- [134] D. Jurafsky and J. H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, Upper Saddle River, New Jersey, 2000. [40](#), [77](#)
- [135] S. Kawashima, H. Ogata, and M. Kanehisa. AAindex: Amino Acid Index Database. *Nucleic Acids Res*, 27(1):368–9, Jan 1999. [189](#)
- [136] U. Keich and P. A. Pevzner. Finding motifs in the twilight zone. *Bioinformatics*, 18(10):1374–1381, Oct 2002. Evaluation Studies. [112](#)
- [137] U. Keich and P. A. Pevzner. Subtle motifs: defining the limits of motif finding algorithms. *Bioinformatics*, 18(10):1382–1390, Oct 2002. Evaluation Studies. [59](#)

- [138] S. M. Kiełbasa, J. O. Korbel, D. Beule, J. Schuchhardt, and H. Herzel. Combining frequency and positional information to predict transcription factor binding sites. *Bioinformatics*, 17(11):1019–26, Nov 2001. 59
- [139] D. M. Kim and C. Y. Choi. A semicontinuous prokaryotic coupled transcription/translation system using a dialysis membrane. *Biotechnol Prog*, 12:645–9, 1996. 92
- [140] S. Kim, S. S. Kim, Y.-J. B. Kim, Seong-Jin, and B. J. Lee. In vitro activities of native and designed peptide antibiotics against drug sensitive and resistant tumor cell lines. *Peptides*, 24(7):945–953, 2003. 73
- [141] D. A. Kimbrell and B. Beutler. The evolution and genetics of innate immunity. *Nat Rev Genet*, 2:256–67, 2001. 72
- [142] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983, 220, 4598:671–680, 1983.* URL citesearcj.nj.nec.com/kirkpatrick83optimization.html. 68
- [143] P. Koehl and M. Levitt. Structure-based conformational preferences of amino acids. *Proc Natl Acad Sci U S A*, 96(22):12524–9, Oct 1999. 192
- [144] R. Kolodny, P. Koehl, and M. Levitt. Comprehensive evaluation of protein structure alignment methods: scoring by geometric measures. *J Mol Biol*, 346(4):1173–88, Mar 2005. 140
- [145] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, R. Funke, D. Gage, K. Harris, A. Heaford, J. Howland, L. Kann, J. Lehoczky, R. LeVine, P. McEwan, K. McKernan, J. Meldrim, J. P. Mesirov, C. Miranda, W. Morris, J. Naylor, C. Raymond, M. Rosetti, R. Santos, A. Sheridan, C. Sougnez, N. Stange-Thomann, N. Stojanovic, A. Subramanian, D. Wyman, J. Rogers, J. Sulston, R. Ainscough, S. Beck, D. Bentley, J. Burton, C. Clee, N. Carter, A. Coulson, R. Deadman, P. Deloukas, A. Dunham, I. Dunham, R. Durbin, L. French, D. Grafham, S. Gregory, T. Hubbard, S. Humphray, A. Hunt, M. Jones,

C. Lloyd, A. McMurray, L. Matthews, S. Mercer, S. Milne, J. C. Mullikin, A. Mungall, R. Plumb, M. Ross, R. Shownkeen, S. Sims, R. H. Waterston, R. K. Wilson, L. W. Hillier, J. D. McPherson, M. A. Marra, E. R. Mardis, L. A. Fulton, A. T. Chinwalla, K. H. Pepin, W. R. Gish, S. L. Chissoe, M. C. Wendl, K. D. Delehaunty, T. L. Miner, A. Delehaunty, J. B. Kramer, L. L. Cook, R. S. Fulton, D. L. Johnson, P. J. Minx, S. W. Clifton, T. Hawkins, E. Branscomb, P. Predki, P. Richardson, S. Wenning, T. Slezak, N. Doggett, J. F. Cheng, A. Olsen, S. Lucas, C. Elkin, E. Uberbacher, M. Frazier, R. A. Gibbs, D. M. Muzny, S. E. Scherer, J. B. Bouck, E. J. Sodergren, K. C. Worley, C. M. Rives, J. H. Gorrell, M. L. Metzker, S. L. Naylor, R. S. Kucherlapati, D. L. Nelson, G. M. Weinstock, Y. Sakaki, A. Fujiyama, M. Hattori, T. Yada, A. Toyoda, T. Itoh, C. Kawagoe, H. Watanabe, Y. Totoki, T. Taylor, J. Weissenbach, R. Heilig, W. Saurin, F. Artiguenave, P. Brottier, T. Bruls, E. Pelletier, C. Robert, P. Wincker, D. R. Smith, L. Doucette-Stamm, M. Rubenfield, K. Weinstock, H. M. Lee, J. Dubois, A. Rosenthal, M. Platzer, G. Nyakatura, S. Taudien, A. Rump, H. Yang, J. Yu, J. Wang, G. Huang, J. Gu, L. Hood, L. Rowen, A. Madan, S. Qin, R. W. Davis, N. A. Federspiel, A. P. Abola, M. J. Proctor, R. M. Myers, J. Schmutz, M. Dickson, J. Grimwood, D. R. Cox, M. V. Olson, R. Kaul, C. Raymond, N. Shimizu, K. Kawasaki, S. Minoshima, G. A. Evans, M. Athanasiou, R. Schultz, B. A. Roe, F. Chen, H. Pan, J. Ramser, H. Lehrach, R. Reinhardt, W. R. McCombie, M. de la Bastide, N. Dedhia, H. Blocker, K. Hornischer, G. Nordsiek, R. Agarwala, L. Aravind, J. A. Bailey, A. Bateman, S. Batzoglou, E. Birney, P. Bork, D. G. Brown, C. B. Burge, L. Cerutti, H. C. Chen, D. Church, M. Clamp, R. R. Copley, T. Doerks, S. R. Eddy, E. E. Eichler, T. S. Furey, J. Galagan, J. G. Gilbert, C. Harmon, Y. Hayashizaki, D. Haussler, H. Hermjakob, K. Hokamp, W. Jang, L. S. Johnson, T. A. Jones, S. Kasif, A. Kaspryzk, S. Kennedy, W. J. Kent, P. Kitts, E. V. Koonin, I. Korf, D. Kulp, D. Lancet, T. M. Lowe, A. McLysaght, T. Mikkelsen, J. V. Moran, N. Mulder, V. J. Pollara, C. P. Ponting, G. Schuler, J. Schultz, G. Slater, A. F. Smit, E. Stupka, J. Szustakowski, D. Thierry-Mieg, J. Thierry-Mieg, L. Wagner, J. Wallis, R. Wheeler, A. Williams, Y. I. Wolf, K. H. Wolfe, S. P. Yang, R. F. Yeh, F. Collins, M. S. Guyer, J. Peterson, A. Felsenfeld, K. A. Wetterstrand, A. Patrinos, M. J. Morgan, P. de Jong, J. J. Catanese, K. Osoegawa, H. Shizuya, S. Choi, and Y. J. Chen. Initial

- sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, Feb 2001. [25](#)
- [146] N. Landwehr, M. Hall, and E. Frank. *Logistic model trees*, volume 2837 of *Lecture Notes in Artificial Intelligence*, pages 241–252. Springer–Verlag, 2003. [193](#)
- [147] C. E. Lawrence, S. F. Altschul, M. S. Boguski, J. S. Liu, A. F. Neuwald, and J. C. Wootton. Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment. *Science*, 262(5131):208–214, Oct 1993. [66](#), [67](#), [68](#), [69](#), [112](#), [148](#), [156](#)
- [148] H. C. M. Leung and F. Y. L. Chin. Finding exact optimal motifs in matrix representation by partitioning. *Bioinformatics*, 21 Suppl 2:ii86–ii92, Sep 2005. [67](#)
- [149] M. Y. Leung, G. M. Marsh, and T. P. Speed. Over- and underrepresentation of short DNA words in herpesvirus genomes. *J Comput Biol*, 3(3):345–60, 1996. [67](#)
- [150] W. Li, L. Jaroszewski, and A. Godzik. Clustering of highly homologous sequences to reduce the size of large protein databases. *Bioinformatics*, 17(3):282–3, Mar 2001. [102](#)
- [151] S. Liang, M. P. Samanta, and B. A. Biegel. cWINNOWER algorithm for finding fuzzy dna motifs. *J Bioinform Comput Biol*, 2(1):47–60, Mar 2004. [59](#)
- [152] C. D. Lima, K. L. D’Amico, I. Naday, G. Rosenbaum, E. M. Westbrook, and W. A. Hendrickson. MAD analysis of FHIT, a putative human tumor suppressor from the HIT protein family. *Structure*, 5(6):763–774, Jul 1997. [140](#)
- [153] D. Liu and W. F. DeGrado. *De novo* design, synthesis, and characterization of antimicrobial beta-peptides. *Journal of the American Chemical Society*, 123(31):7553–7559, 2001. [98](#)
- [154] J. Liu. The collapsed gibbs sampler in bayesian computations with applications to a gene regulation problem. *Journal of the American Statistical Association*, 89(427):958–966, 1994. [67](#)
- [155] J. S. Liu. *Monte Carlo Strategies in Scientific Computing*. Springer-Verlag, New York, 2001. [66](#), [68](#)

- [156] Y. Liu, M. P. Vincenti, and H. Yokota. Principal component analysis for predicting transcription-factor binding motifs from array-derived data. *BMC Bioinformatics*, 6:276, 2005. [67](#)
- [157] I. S. Lossos, R. Tibshirani, B. Narasimhan, and R. Levy. The inference of antigen selection on Ig genes. *J Immunol*, 165(9):5122–6, Nov 2000. [201](#)
- [158] R. Lutz and H. Bujard. Independent and tight regulation of transcriptional units in *Escherichia coli* via the LacR/O, the TetR/O and AraC/I₁-I₂ regulatory elements. *Nucleic Acids Res*, 25(6):1203–10, Mar 1997. [202](#), [204](#), [205](#)
- [159] K. D. Macisaac, D. B. Gordon, L. Nekludova, D. T. Odom, J. Schreiber, D. K. Gifford, R. A. Young, and E. Fraenkel. A hypothesis-based approach for identifying the binding specificity of regulatory proteins from chromatin immunoprecipitation data. *Bioinformatics*, 22(4):423–9, Feb 2006. [67](#)
- [160] T. Madej, J. F. Gibrat, and S. H. Bryant. Threading a database of protein cores. *Proteins*, 23(3):356–369, Nov 1995. [138](#)
- [161] D. Maier. The complexity of some problems on subsequences and supersequences. *J ACM*, 25(2):322–336, 1978. [58](#), [60](#)
- [162] J. V. Maizel and R. P. Lenk. Enhanced graphic matrix analysis of nucleic acid and protein sequences. *Proc Natl Acad Sci U S A*, 78(12):7665–9, Dec 1981. [102](#)
- [163] A. Mancheron and I. Rusu. Pattern discovery allowing wild-cards, substitution matrices, and multiple score functions. In *Algorithms in Bioinformatics, Proceedings Lecture notes in Bioinformatics*, pages 124–138. Springer–Verlag, 2003. [116](#)
- [164] H. J. Mangalam. tacg—a grep for DNA. *BMC Bioinformatics*, 3:8, 2002. [160](#)
- [165] A. Marchler-Bauer, J. B. Anderson, C. DeWeese-Scott, N. D. Fedorova, L. Y. Geer, S. He, D. I. Hurwitz, J. D. Jackson, A. R. Jacobs, C. J. Lanczycki, C. A. Liebert, C. Liu, T. Madej, G. H. Marchler, R. Mazumder, A. N. Nikolskaya, A. R. Panchenko, B. S. Rao, B. A. Shoemaker, V. Simonyan, J. S. Song, P. A. Thiessen, S. Vasudevan, Y. Wang, R. A.

- Yamashita, J. J. Yin, and S. H. Bryant. CDD: a curated Entrez database of conserved domain alignments. *Nucleic Acids Res*, 31(1):383–387, Feb 2003. [138](#)
- [166] M. Markstein, R. Zinzen, P. Markstein, K.-P. Yee, A. Erives, A. Stathopoulos, and M. Levine. A regulatory code for neurogenic gene expression in the Drosophila embryo. *Development*, 131(10):2387–94, May 2004. [59](#)
- [167] L. Marsan and M. F. Sagot. Algorithms for extracting structured motifs using a suffix tree with an application to promoter and regulatory site consensus identification. *J Comput Biol*, 7(3–4):345–62, 2000. [59](#)
- [168] K. A. Martemyanov, V. A. Shirokov, O. V. Kurnasov, A. T. Gudkov, and A. S. Spirin. Cell-free production of biologically active polypeptides: application to the synthesis of antibacterial peptide cecropin. *Protein Expr Purif*, 21(3):456–461, Apr 2001. [92](#)
- [169] G. Mengeritsky and T. F. Smith. Recognition of characteristic patterns in sets of functionally equivalent DNA sequences. *Comput Appl Biosci*, 3(3):223–7, Sep 1987. [59](#)
- [170] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, and A. H. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21(6):1087–1092, Jun 1953. [66](#)
- [171] S. Mitaku, T. Hirokawa, and T. Tsuji. Amphiphilicity index of polar amino acids as an aid in the characterization of amino acid preference at membrane-water interfaces. *Bioinformatics*, 18(4):608–16, Apr 2002. [192](#)
- [172] L. Moerman, S. Bosteels, W. Noppe, J. Willems, E. Clynen, L. Schoofs, K. Thevissen, J. Tytgat, J. Van Eldere., J. Van Der Walt., and F. Verdonck. Antibacterial and antifungal properties of alpha-helical, cationic peptides in the venom of scorpions from southern Africa. *Eur J Biochem*, 269(19):4799–4810, Oct 2002. [72](#)
- [173] F. Mueller. A library implementation of POSIX threads under unix. In *Proceedings of the Winter 1993 USENIX Technical Conference and Exhibition*, pages 29–41, San Diego, CA, USA, 1993. [160](#)

- [174] V. L. Murthy and G. D. Rose. RNABase: an annotated database of RNA structures. *Nucleic Acids Res*, 31(1):502–504, Jan 2003. 112
- [175] A. G. Murzin, S. E. Brenner, T. Hubbard, and C. Chothia. SCOP: a structural classification of proteins database for the investigation of sequences and structures. *J Mol Biol*, 247(4):536–40, Apr 1995. 166
- [176] K. Nakai, A. Kidera, and M. Kanehisa. Cluster analysis of amino acid indices for prediction of protein structure and function. *Protein Eng*, 2(2):93–100, Jul 1988. 189
- [177] A. Narayanan, X. Wu, and Z. R. Yang. Mining viral protease data to extract cleavage knowledge. *Bioinformatics*, 18 Suppl 1:S5–13, 2002. 190
- [178] G. Navarro. NR-grep: a fast and flexible pattern-matching tool. *Software Practice and Experience*, 31(13):1265–1312, ????, 2001. 160
- [179] A. Neuwald and P. Green. Detecting patterns in protein sequences. *Journal of Molecular Biology*, 239:698–712, 1994. 59
- [180] H. Ney and S. Ortmanns. Progress in dynamic programming search for LVCSR. *Proceedings of the IEEE*, 88(8):1244–1240, 2000. 178
- [181] P. C. Ng and S. Henikoff. Predicting deleterious amino acid substitutions. *Genome Res*, 11(5):863–874, May 2001. URL <http://www.ncbi.nlm.nih.gov/pubmed/11337480>. 163
- [182] B. H. Normark and S. Normark. Evolution and spread of antibiotic resistance. *J Intern Med*, 252:91–106, 2002. 72
- [183] C. Notredame, D. G. Higgins, and J. Heringa. T-coffee: A novel method for fast and accurate multiple sequence alignment. *J Mol Biol*, 302(1):205–217, Sep 2000. URL <http://www.ncbi.nlm.nih.gov/pubmed/10964570>. 163
- [184] I. of Medicine. *Antimicrobial Resistance: Issues and Options*. National Academy Press, 1998. 72

- [185] C. A. Orengo and W. R. Taylor. SSAP: sequential structure alignment program for protein structure comparison. *Methods Enzymol.*, 266:617–635, 1996. [138](#)
- [186] H. A. Orr. A minimum on the mean number of steps taken in adaptive walks. *J Theor Biol.*, 220(2):241–7, Jan 2003. [211](#)
- [187] A. R. Ortiz, C. E. M. Strauss, and O. Olmea. MAMMOTH (matching molecular models obtained from theory): an automated method for model comparison. *Protein Sci.*, 11(11):2606–2621, Nov 2002. Evaluation Studies. [138](#)
- [188] J. Palau, P. Argos, and P. Puigdomenech. Protein secondary structure. Studies on the limits of prediction accuracy. *Int J Pept Protein Res.*, 19(4):394–401, Apr 1982. [192](#)
- [189] L. Parida, I. Rigoutsos, and A. Floratos. MUSCA: an algorithm for constrained alignment of multiple data sequences. In *Proceedings of the Twelfth International Conference on Genome Informatics (GIW)*, Tokyo, 1998. [64](#)
- [190] J. Parrish-Novak, S. R. Dillon, A. Nelson, A. Hammond, C. Sprecher, J. A. Gross, J. Johnston, K. Madden, W. Xu, J. West, S. Schrader, S. Burkhead, M. Heipel, C. Brandt, J. L. Kuijper, J. Kramer, D. Conklin, S. R. Presnell, J. Berry, F. Shiota, S. Bort, K. Hambley, S. Mudri, C. Clegg, M. Moore, F. J. Grant, C. Lofton-Day, T. Gilbert, F. Rayond, A. Ching, L. Yao, D. Smith, P. Webster, T. Whitmore, M. Maurer, K. Kaushansky, R. D. Holly, and D. Foster. Interleukin 21 and its receptor are involved in NK cell expansion and regulation of lymphocyte function. *Nature*, 408(6808):57–63, Nov 2000. [45](#)
- [191] G. Pavesi, P. Mereghetti, G. Mauri, and G. Pesole. Weeder Web: discovery of transcription factor binding sites in a set of sequences from co-regulated genes. *Nucleic Acids Res.*, 32(Web Server issue):W199–203, Jul 2004. [59](#)
- [192] W. R. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods Enzymol.*, 183:63–98, 1990. [166](#)
- [193] W. R. Pearson. Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. *Genomics*, 11(3):635–50, Nov 1991. [166, 167](#)

- [194] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85:2444–2448, 1988. 160, 163, 177
- [195] P. A. Pevzner and S. Sze. Combinatorial approaches to finding subtle signals in DNA sequences. In *Proceedings International Conference on Intelligent Systems for Molecular Biology*, pages 269–278. AAAI Press, 2000. 59, 116, 143, 146, 148, 150
- [196] W. W. Piegorsch and A. J. Bailer. Statistical approaches for analyzing mutational spectra: some recommendations for categorical data. *Genetics*, 136(1):403–16, Jan 1994. 201
- [197] S. Pietrokovski. Searching databases of conserved sequence regions by aligning protein multiple-alignments. *Nucleic Acids Res*, 24(19):3836–3845, Oct 1996. URL http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=NucleicAcidsRes&list_uids=8871566. 163
- [198] M. Prabhakaran. The distribution of physical, chemical and conformational properties in signal and nascent peptides. *Biochem J*, 269(3):691–6, Aug 1990. 192
- [199] A. Price, S. Ramabhadran, and P. A. Pevzner. Finding subtle motifs by branching from sample strings. *Bioinformatics*, 19 Suppl 2:II149–II155, Oct 2003. 59, 112
- [200] G. A. Price, G. E. Crooks, R. E. Green, and S. E. Brenner. Statistical evaluation of pairwise protein sequence comparison with the bayesian bootstrap. *Bioinformatics*, 21(20):3824–3831, Oct 2005. URL http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve&db=Bioinformatics&list_uids=16105900. 166, 168, 172
- [201] K. Putsep, G. Carlsson, H. G. Boman, and M. Andersson. Deficiency of antibacterial peptides in patients with morbus Kostmann: an observation study. *Lancet*, 360:1144–9, 2002. 72
- [202] C. Queen, M. Wegman, and L. Korn. Improvements to a program for dna analysis: a procedure to find homologies among many sequences. *Nucleic Acids Research*, 10:449–456, 1982. 59
- [203] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Francisco, CA, USA, 1992. 193

- [204] B. Raphael, L.-T. Liu, and G. Varghese. A uniform projection method for motif discovery in dna sequences. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 1(2):91–94, 2004. ISSN 1545-5963. [67](#)
- [205] J. H. Reif. Computing. Successes and challenges. *Science*, 296(5567):478–9, Apr 2002. [19](#)
- [206] P. Rice, I. Longden, and A. Bleasby. EMBOSS: the European Molecular Biology Open Software Suite. *Trends Genet.*, 16:276–7, 2000. [91](#), [160](#)
- [207] I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in biological sequences: The TEIRESIAS algorithm. *Bioinformatics*, 14:55–67, 1998. [59](#), [60](#), [77](#), [112](#), [116](#), [123](#), [155](#), [156](#), [160](#)
- [208] I. Rigoutsos, A. Floratos, C. Ouzounis, Y. Gao, and L. Parida. Dictionary building via unsupervised hierarchical motif discovery in the sequence space of natural proteins. *Proteins*, 37:264–77, 1999. [64](#), [82](#), [114](#)
- [209] E. Rivas and S. R. Eddy. The language of RNA: a formal grammar that includes pseudoknots. *Bioinformatics*, 16(4):334–40, Apr 2000. [40](#)
- [210] B. Robson and E. Suzuki. Conformational properties of amino acid residues in globular proteins. *J Mol Biol*, 107(3):327–56, Nov 1976. [192](#)
- [211] T. Rognvaldsson and L. You. Why neural networks should not be used for HIV-1 protease cleavage site prediction. *Bioinformatics*, 20(11):1702–1709, Jul 2004. [190](#)
- [212] D. R. Rokyta, P. Joyce, S. B. Caudle, and H. A. Wichman. An empirical test of the mutational landscape model of adaptation using a single-stranded DNA virus. *Nat Genet*, 37(4):441–4, Apr 2005. [211](#)
- [213] J. Rolff and M. T. Siva-Jothy. Invertebrate Ecological Immunology. *Science*, 301(5632):472–475, 2003. [72](#)
- [214] T. M. Rose, E. R. Schultz, J. G. Henikoff, S. Pietrokovski, C. M. McCallum, and S. Henikoff. Consensus-degenerate hybrid oligonucleotide primers for amplification

- of distantly related sequences. *Nucleic Acids Res*, 26(7):1628–1635, Apr 1998. URL <http://www.ncbi.nlm.nih.gov/pubmed/9512532>. 163
- [215] M. Sagot, A. Viari, and H. Soldano. Multiple sequence comparison — A peptide matching approach. *Theoretical Computer Science*, 180(1–2):115–137, 1997. 59
- [216] C. Salazar, J. Schütze, and O. Ebenhöh. Bioinformatics meets systems biology. *Genome Biol*, 7(1):303, 2006. 25
- [217] H. Salgado, S. Gama-Castro, A. Martinez-Antonio, E. Diaz-Peredo, F. Sanchez-Solano, M. Peralta-Gil, D. Garcia-Alonso, V. Jimenez-Jacinto, A. Santos-Zavaleta, C. Bonavides-Martinez, and J. Collado-Vides. RegulonDB (version 4.0): transcriptional regulation, operon organization and growth conditions in *Escherichia coli* K-12. *Nucleic Acids Res*, 32(Database issue):303–306, Jan 2004. 153
- [218] N. H. Salzman, D. Ghosh, K. M. Huttner, Y. Paterson, and C. L. Bevins. Protection against enteric salmonellosis in transgenic mice expressing a human intestinal defensin. *Nature*, 422:522–6, 2003. 72
- [219] B. Schittekkat, R. Hipfel, B. Sauer, J. Bauer, H. Kalbacher, S. Stevanovic, M. Schirle, K. Schroeder, N. Blin, F. Meier, G. Rassner, and C. Garbe. Dermcidin: a novel human antibiotic peptide secreted by sweat glands. *Nat Immunol*, 2:1133–7, 2001. 72
- [220] B. Schuster-Böckler, J. Schultz, and S. Rahmann. HMM Logos for visualization of protein families. *BMCS Bioinformatics*, 5:7, Jan 2004. 141
- [221] M. S. Scott, D. Y. Thomas, and M. T. Hallett. Predicting subcellular localization via protein motif co-occurrence. *Genome Res*, 14(10A):1957–66, Oct 2004. 194
- [222] D. B. Searls. The computational linguistics of biological sequences. In L. Hunter, editor, *Artificial Intelligence and Molecular Biology*, pages 47–120. AAAI Press, 1992. 31
- [223] D. B. Searls. Linguistic approaches to biological sequences. *Comput Appl Biosci*, 13:333–44, 1997. 31
- [224] D. B. Searls. Reading the book of life. *Bioinformatics*, 17(7):579–580, 2001. 31

- [225] D. B. Searls. The language of genes. *Nature*, 420:211–7, 2002. 77
- [226] Y. Shai. Mode of action of membrane active antimicrobial peptides. *Biopolymers*, 66: 236–48, 2002. 74, 94, 95
- [227] J. Shendure, R. D. Mitra, C. Varma, and G. M. Church. Advanced sequencing technologies: methods and goals. *Nat Rev Genet*, 5(5):335–44, May 2004. 19, 20
- [228] S. M. Shieber. Evidence against the context-freeness of natural language. *Linguistics and Philosophy*, 8:333–343, 1985. 40
- [229] R. Siddharthan, E. D. Siggia, and E. van Nimwegen. PhyloGibbs: a gibbs sampling motif finder that incorporates phylogeny. *PLoS Comput Biol*, 1(7):e67, Dec 2005. 67
- [230] M. Simmaco, G. Mignogna, and D. Barra. Antimicrobial peptides from amphibian skin: What do they tell us? *Biopolymers*, 47(6):435–450, 1999. 72
- [231] S. Sinha. Discriminative motifs. *J Comput Biol*, 10(3-4):599–615, 2003. 59
- [232] S. Sinha and M. Tompa. Discovery of novel transcription factor binding sites by statistical overrepresentation. Department of Computer Science and Engineering, University of Washington, 2002. 59
- [233] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997. 44
- [234] J. J. Smith, S. M. Travis, E. P. Greenberg, and M. J. Welsh. Cystic fibrosis airway epithelia fail to kill bacteria because of abnormal airway surface fluid. *Cell*, 85:229–36, 1996. 72
- [235] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–7, Mar 1981. 166
- [236] C. Solem and P. R. Jensen. Modulation of gene expression made easy. *Appl Environ Microbiol*, 68(5):2397–403, May 2002. 200
- [237] R. Staden. Methods for discovering novel motifs in nucleic acid sequences. *Comput Appl Biosci*, 5(4):293–8, Oct 1989. 59

- [238] G. D. Stormo and G. W. Hartzell. Identifying protein-binding sites from unaligned DNA fragments. *Proc Natl Acad Sci U S A*, 86(4):1183–7, Feb 1989. 67
- [239] M. B. Strom, B. E. Haug, M. L. Skar, W. Stensen, T. Stiberg, and J. S. Svendsen. The pharmacophore of short cationic antibacterial peptides. *J Med Chem*, 46:1567–70, 2003. 77
- [240] P. Sumazin, G. Chen, N. Hata, A. D. Smith, T. Zhang, and M. Q. Zhang. DWE: discriminating word enumerator. *Bioinformatics*, 21(1):31–8, Jan 2005. 59
- [241] A. L. Swain, M. M. Miller, J. Green, D. H. Rich, J. Schneider, S. B. Kent, and A. Wlodawer. X-ray crystallographic structure of a complex between a synthetic protease of human immunodeficiency virus 1 and a substrate-based hydroxyethylamine inhibitor. *Proc Natl Acad Sci U S A*, 87(22):8805–9, Nov 1990. 15, 190
- [242] C. C. Tappert, C. Y. Suen, and T. Wakahara. The state of the art in online handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(8):787–808, 1990. 177
- [243] K. Tharakaraman, L. Mariño-Ramírez, S. Sheetlin, D. Landsman, and J. L. Spouge. Alignments anchored on genomic landmarks can aid in the identification of regulatory elements. *Bioinformatics*, 21 Suppl 1:i440–i448, Jun 2005. 67
- [244] The computational biology and functional genomics laboratory at the Dana–Farber Cancer Institute and Harvard School of Public Health. The -omics revolution count. http://biocomp.dfci.harvard.edu/tgi/omics_count.html, Dec 2005. 22
- [245] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res*, 22(22):4673–4680, Nov 1994. 163
- [246] E. Tiozzo, G. Rocco, A. Tossi, and D. Romeo. Wide-spectrum antibiotic activity of

- synthetic, amphipathic peptides. *Biochem Biophys Res Commun*, 249:202–6, 1998. 77, 109
- [247] K. Tomii and M. Kanehisa. Analysis of amino acid indices and mutation matrices for sequence comparison and structure prediction of proteins. *Protein Eng*, 9(1):27–36, Jan 1996. 189
- [248] E. Tomita, A. Tanaka, and H. Takahasi. An optimal algorithm for finding all the cliques. *SIG Algorithms*, 12:91–98, 1989. 124
- [249] M. Tompa, N. Li, T. L. Bailey, G. M. Church, B. De Moor, E. Eskin, A. V. Favorov, M. C. Frith, Y. Fu, W. J. Kent, V. J. Makeev, A. A. Mironov, W. S. Noble, G. Pavese, G. Pesole, M. Regnier, N. Simonis, S. Sinha, G. Thijs, J. van Helden, M. Vandenbogaert, Z. Weng, C. Workman, C. Ye, and Z. Zhu. Assessing computational tools for the discovery of transcription factor binding sites. *Nat Biotechnol*, 23(1):137–144, Jan 2005. 68, 114
- [250] A. Tossi. Antimicrobial sequences database (AMSDb), 2002. <http://www.bbcm.univ.trieste.it/tossi/amsdb.html>. 77, 101
- [251] A. Tossi, L. Sandri, and A. Giangaspero. Amphipathic, alpha-helical antimicrobial peptides. *Biopolymers*, 55:4–30, 2000. 73, 109
- [252] J. van Helden, B. André, and J. Collado-Vides. Extracting regulatory sites from the upstream region of yeast genes by computational analysis of oligonucleotide frequencies. *J Mol Biol*, 281(5):827–42, Sep 1998. 59
- [253] J. van Helden, A. F. Rios, and J. Collado-Vides. Discovering regulatory elements in non-coding sequences by analysis of spaced dyads. *Nucleic Acids Res*, 28(8):1808–18, Apr 2000. 59
- [254] V. N. Vapnik. *The nature of statistical learning theory*. Springer-Verlag New York, Inc., New York, NY, USA, 1995. ISBN 0-387-94559-8. 195

- [255] V. N. Vapnik. *Statistical learning theory*. Wiley, 1998. ISBN 0-471-03003-1. VAP v 98:1 1.Ex. 195
- [256] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, J. D. Gocayne, P. Amanatides, R. M. Ballew, D. H. Huson, J. R. Wortman, Q. Zhang, C. D. Kodira, X. H. Zheng, L. Chen, M. Skupski, G. Subramanian, P. D. Thomas, J. Zhang, G. L. Gabor Miklos, C. Nelson, S. Broder, A. G. Clark, J. Nadeau, V. A. McKusick, N. Zinder, A. J. Levine, R. J. Roberts, M. Simon, C. Slayman, M. Hunkapiller, R. Bolanos, A. Delcher, I. Dew, D. Fasulo, M. Flanigan, L. Florea, A. Halpern, S. Hannenhalli, S. Kravitz, S. Levy, C. Mooney, K. Reinert, K. Remington, J. Abu-Threideh, E. Beasley, K. Biddick, V. Bonazzi, R. Brandon, M. Cargill, I. Chandramouliwaran, R. Charlab, K. Chaturvedi, Z. Deng, V. Di Francesco, P. Dunn, K. Eilbeck, C. Evangelista, A. E. Gabrielian, W. Gan, W. Ge, F. Gong, Z. Gu, P. Guan, T. J. Heiman, M. E. Higgins, R. R. Ji, Z. Ke, K. A. Ketchum, Z. Lai, Y. Lei, Z. Li, J. Li, Y. Liang, X. Lin, F. Lu, G. V. Merkulov, N. Milshina, H. M. Moore, A. K. Naik, V. A. Narayan, B. Neelam, D. Nusskern, D. B. Rusch, S. Salzberg, W. Shao, B. Shue, J. Sun, Z. Wang, A. Wang, X. Wang, J. Wang, M. Wei, R. Wides, C. Xiao, C. Yan, A. Yao, J. Ye, M. Zhan, W. Zhang, H. Zhang, Q. Zhao, L. Zheng, F. Zhong, W. Zhong, S. Zhu, S. Zhao, D. Gilbert, S. Baumhueter, G. Spier, C. Carter, A. Cravchik, T. Woodage, F. Ali, H. An, A. Awe, D. Baldwin, H. Baden, M. Barnstead, I. Barrow, K. Beeson, D. Busam, A. Carver, A. Center, M. L. Cheng, L. Curry, S. Danaher, L. Davenport, R. Desilets, S. Dietz, K. Dodson, L. Doucet, S. Ferriera, N. Garg, A. Gluecksmann, B. Hart, J. Haynes, C. Haynes, C. Heiner, S. Hladun, D. Hostin, J. Houck, T. Howland, C. Ibegwam, J. Johnson, F. Kalush, L. Kline, S. Koduru, A. Love, F. Mann, D. May, S. McCawley, T. McIntosh, I. McMullen, M. Moy, L. Moy, B. Murphy, K. Nelson, C. Pfannkoch, E. Pratts, V. Puri, H. Qureshi, M. Reardon, R. Rodriguez, Y. H. Rogers, D. Romblad, B. Ruhfel, R. Scott, C. Sitter, M. Smallwood, E. Stewart, R. Strong, E. Suh, R. Thomas, N. N. Tint, S. Tse, C. Vech, G. Wang, J. Wetter, S. Williams, M. Williams, S. Windsor, E. Winn-Deen, K. Wolfe, J. Zaveri, K. Zaveri, J. F. Abril, R. Guigo, M. J. Campbell, K. V. Sjolander, B. Karlak, A. Kejariwal, H. Mi,

- B. Lazareva, T. Hatton, A. Narechania, K. Diemer, A. Muruganujan, N. Guo, S. Sato, V. Bafna, S. Istrail, R. Lippert, R. Schwartz, B. Walenz, S. Yooseph, D. Allen, A. Basu, J. Baxendale, L. Blick, M. Caminha, J. Carnes-Stine, P. Caulk, Y. H. Chiang, M. Coyne, C. Dahlke, A. Mays, M. Dombroski, M. Donnelly, D. Ely, S. Esparham, C. Fosler, H. Gire, S. Glanowski, K. Glasser, A. Glodek, M. Gorokhov, K. Graham, B. Gropman, M. Harris, J. Heil, S. Henderson, J. Hoover, D. Jennings, C. Jordan, J. Jordan, J. Kasha, L. Kagan, C. Kraft, A. Levitsky, M. Lewis, X. Liu, J. Lopez, D. Ma, W. Majoros, J. McDaniel, S. Murphy, M. Newman, T. Nguyen, N. Nguyen, M. Nodell, S. Pan, J. Peck, M. Peterson, W. Rowe, R. Sanders, J. Scott, M. Simpson, T. Smith, A. Sprague, T. Stockwell, R. Turner, E. Venter, M. Wang, M. Wen, D. Wu, M. Wu, A. Xia, A. Zandieh, and X. Zhu. The sequence of the human genome. *Science*, 291(5507):1304–1351, Feb 2001.
- [25](#)
- [257] J. Vizioli, P. Bulet, J. A. Hoffmann, F. C. Kafatos, H.-M. Muller, and G. Dimopoulos. Gambicin: A novel immune responsive antimicrobial peptide from the malaria vector *Anopheles gambiae*. *PNAS*, 98(22):12630–12635, 2001. [73](#)
- [258] G. Vogt, T. Etzold, and P. Argos. An assessment of amino acid exchange matrices in aligning protein sequences: the twilight zone revisited. *J Mol Biol*, 249:816–31, 1995. [163, 186](#)
- [259] D. R. Walker, J. P. Bond, R. E. Tarone, C. C. Harris, W. Makalowski, M. S. Boguski, and M. S. Greenblatt. Evolutionary conservation and somatic mutation hotspot maps of p53: correlation with p53 protein structural and functional features. *Oncogene*, 18(1):211–8, Jan 1999. [201](#)
- [260] C. Walsh. Molecular mechanisms that confer antibacterial drug resistance. *Nature*, 406:775–81, 2000. [72](#)
- [261] G. Wang, Y. Li, and X. Li. Correlation of three-dimensional structures with the antibacterial activity of a group of peptides designed based on a nontoxic bacterial membrane anchor. *J Biol Chem*, 280(7):5803–11, Feb 2005. [73, 75](#)

- [262] W. Wang and P. A. Kollman. Computational study of protein specificity: the molecular basis of HIV-1 protease drug resistance. *Proc Natl Acad Sci U S A*, 98(26):14937–42, Dec 2001. [190](#)
- [263] Z. Wang and G. Wang. APD: the Antimicrobial Peptide Database. *Nucleic Acids Res*, 32(Database issue):D590–2, Jan 2004. [100](#)
- [264] M. S. Waterman. Efficient sequence alignment algorithms. *J Theor Biol*, 108:333–7, 1984. [175](#)
- [265] R. H. Waterston, K. Lindblad-Toh, E. Birney, J. Rogers, J. F. Abril, P. Agarwal, R. Agarwala, R. Ainscough, M. Alexandersson, P. An, S. E. Antonarakis, J. Attwood, R. Baertsch, J. Bailey, K. Barlow, S. Beck, E. Berry, B. Birren, T. Bloom, P. Bork, M. Botcherby, N. Bray, M. R. Brent, D. G. Brown, S. D. Brown, C. Bult, J. Burton, J. Butler, R. D. Campbell, P. Carninci, S. Cawley, F. Chiaromonte, A. T. Chinwalla, D. M. Church, M. Clamp, C. Clee, F. S. Collins, L. L. Cook, R. R. Copley, A. Coulson, O. Couronne, J. Cuff, V. Curwen, T. Cutts, M. Daly, R. David, J. Davies, K. D. Delehaunty, J. Deri, E. T. Dermitzakis, C. Dewey, N. J. Dickens, M. Diekhans, S. Dodge, I. Dubchak, D. M. Dunn, S. R. Eddy, L. Elnitski, R. D. Emes, P. Eswara, E. Eyras, A. Felsenfeld, G. A. Fewell, P. Flicek, K. Foley, W. N. Frankel, L. A. Fulton, R. S. Fulton, T. S. Furey, D. Gage, R. A. Gibbs, G. Glusman, S. Gnerre, N. Goldman, L. Goodstadt, D. Grafham, T. A. Graves, E. D. Green, S. Gregory, R. Guigó, M. Guyer, R. C. Hardison, D. Haussler, Y. Hayashizaki, L. W. Hillier, A. Hinrichs, W. Hlavina, T. Holzer, F. Hsu, A. Hua, T. Hubbard, A. Hunt, I. Jackson, D. B. Jaffe, L. S. Johnson, M. Jones, T. A. Jones, A. Joy, M. Kamal, E. K. Karlsson, D. Karolchik, A. Kasprzyk, J. Kawai, E. Keibler, C. Kells, W. J. Kent, A. Kirby, D. L. Kolbe, I. Korf, R. S. Kucherlapati, E. J. Kulp, T. Landers, J. P. Leger, S. Leonard, I. Letunic, R. Levine, J. Li, M. Li, C. Lloyd, S. Lucas, B. Ma, D. R. Maglott, E. R. Mardis, L. Matthews, E. Mauceli, J. H. Mayer, M. McCarthy, W. R. McCombie, S. McLaren, K. McLay, J. D. McPherson, J. Meldrim, B. Meredith, J. P. Mesirov, W. Miller, T. L. Miner, E. Mongin, K. T. Montgomery, M. Morgan, R. Mott, J. C. Mullikin, D. M. Muzny, W. E. Nash, J. O. Nelson, M. N. Nhan, R. Nicol, Z. Ning, C. Nusbaum, M. J. O'Connor,

- Y. Okazaki, K. Oliver, E. Overton-Larty, L. Pachter, G. Parra, K. H. Pepin, J. Peterson, P. Pevzner, R. Plumb, C. S. Pohl, A. Poliakov, T. C. Ponce, C. P. Ponting, S. Potter, M. Quail, A. Reymond, B. A. Roe, K. M. Roskin, E. M. Rubin, A. G. Rust, R. Santos, V. Sapochnikov, B. Schultz, J. Schultz, M. S. Schwartz, S. Schwartz, C. Scott, S. Seaman, S. Searle, T. Sharpe, A. Sheridan, R. Shownkeen, S. Sims, J. B. Singer, G. Slater, A. Smit, D. R. Smith, B. Spencer, A. Stabenau, N. Stange-Thomann, C. Sugnet, M. Suyama, G. Tesler, J. Thompson, D. Torrents, E. Trevaskis, J. Tromp, C. Ucla, A. Ureta-Vidal, J. P. Vinson, A. C. Von Niederhausern, C. M. Wade, M. Wall, R. J. Weber, R. B. Weiss, M. C. Wendl, A. P. West, K. Wetterstrand, R. Wheeler, S. Whelan, J. Wierzbowski, D. Willey, S. Williams, R. K. Wilson, E. Winter, K. C. Worley, D. Wyman, S. Yang, S. P. Yang, E. M. Zdobnov, M. C. Zody, and E. S. Lander. Initial sequencing and comparative analysis of the mouse genome. *Nature*, 420(6915):520–562, Dec 2002. URL <http://www.ncbi.nlm.nih.gov/pubmed/12466850>. 25
- [266] J. E. Wedekind, P. A. Frey, and I. Rayment. The structure of nucleotidylated histidine-166 of galactose-1-phosphate uridylyltransferase provides insight into phosphoryl group transfer. *Biochemistry*, 35(36):11560–11569, Oct 1996. 140
- [267] D. L. Wheeler, T. Barrett, D. A. Benson, S. H. Bryant, K. Canese, V. Chetvernin, D. M. Church, M. DiCuccio, R. Edgar, S. Federhen, L. Y. Geer, W. Helmberg, Y. Kapustin, D. L. Kenton, O. Khovayko, D. J. Lipman, T. L. Madden, D. R. Maglott, J. Ostell, K. D. Pruitt, G. D. Schuler, L. M. Schriml, E. Sequeira, S. T. Sherry, K. Sirotnik, A. Souvorov, G. Starchenko, T. O. Suzek, R. Tatusov, T. A. Tatusova, L. Wagner, and E. Yaschenko. Database resources of the National Center for Biotechnology Information. *Nucleic Acids Res*, 34(Database issue):D173–80, Jan 2006. 25
- [268] C. L. Wilson, A. J. Ouellette, D. P. Satchell, T. Ayabe, Y. S. Lopez-Boado, J. L. Stratman, S. J. Hultgren, L. M. Matrisian, and W. C. Parks. Regulation of intestinal alpha-defensin activation by the metalloproteinase matrilysin in innate host defense. *Science*, 286:113–7, 1999. 72
- [269] I. H. Witten and E. Frank. *Data mining: practical machine learning tools and techniques*

- with Java implementations*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. 192, 194
- [270] F. Wolferstetter, K. French, G. Herrmann, and T. Werner. Identification of functional. *Computer Applications in the Biosciences*, 12(1):71–80, 1996. 59
- [271] D. H. Wolpert and W. G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe, NM, 1995. URL citeseer.ist.psu.edu/wolpert95no.html. 196
- [272] J. C. Wootton and S. Federhen. Statistics of local complexity in amino acid sequences and sequence databases. *Computers in Chemistry*, 17:149–163, 1993. 166
- [273] C. T. Workman and G. D. Stormo. Ann-spec: a method for discovering transcription factor binding sites with improved specificity. *Pac Symp Biocomput*, pages 467–478, 2000. URL <http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1650000/>. 67
- [274] M. Wu and R. E. Hancock. Interaction of the cyclic antimicrobial cationic peptide bactenecin with the outer and cytoplasmic membrane. *J Biol Chem*, 274(1):29–35, Jan 1999. 105
- [275] S. Wu and U. Manber. Agrep — A fast approximate pattern-matching tool. In *Usenix Winter 1992 Technical Conference*, pages 153–162, San Francisco, Jan. 1992. 160
- [276] L. You, D. Garwicz, and T. Rögnvaldsson. Comprehensive bioinformatic analysis of the specificity of human immunodeficiency virus type 1 protease. *J Virol*, 79(19):12477–86, Oct 2005. 190, 191
- [277] M. J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000. URL citeseer.ist.psu.edu/zaki00scalable.html. 116
- [278] M. J. Zaki and M. Ogihara. Theoretical foundations of association rules. In *In Proceedings of 3rd SIGMOD'98 Workshop on Research Issues in Data Mining and Knowledge Dis-*

- covery (DMKD'98), Seattle, Washington, 1998. URL citeseer.ist.psu.edu/zaki98theoretical.html. 116
- [279] M. Zasloff. Antimicrobial peptides in health and disease. *New England Journal of Medicine*, 347(15):1199–1200, Oct. 2002. 72, 77
- [280] M. Zasloff. Antimicrobial peptides of multicellular organisms. *Nature*, 415:389–95, 2002. 72, 73, 74, 77, 94
- [281] L. Zhang, W. Yu, T. He, J. Yu, R. E. Caffrey, E. A. Dalmasso, S. Fu, T. Pham, J. Mei, J. J. Ho, W. Zhang, P. Lopez, and D. D. Ho. Contribution of human alpha-defensin 1, 2, and 3 to the anti-HIV-1 activity of CD8 antiviral factor. *Science*, 298:995–1000, 2002. 73
- [282] W. Zhong, P. Zeng, P. Ma, J. S. Liu, and Y. Zhu. RSIR: regularized sliced inverse regression for motif discovery. *Bioinformatics*, 21(22):4169–75, Nov 2005. 67