

UCSF

UC San Francisco Electronic Theses and Dissertations

Title

Transcription factor binding site modeling in higher organisms

Permalink

<https://escholarship.org/uc/item/54r0x034>

Author

Hon, Lawrence Sean

Publication Date

2005

Peer reviewed|Thesis/dissertation

Transcription Factor Binding Site Modeling in Higher Organisms

by

Lawrence Sean Hon

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Biological and Medical Informatics

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, SAN FRANCISCO

12/11/11

11007 110010101

© 2005
Lawrence Sean Hon

Acknowledgements

This thesis is the embodiment of the many things I have learned while being a Ph.D. student. These include not only the obvious, such as learning about transcriptional regulation and human biology, but also the more subtle but perhaps more important, such as an enthusiasm about basic research, how to do good research, and presenting logical thoughts in writing and in presentations. Beyond that, I realized that the successful Ph.D. student depends heavily on friends and family for moral support. For such a valuable experience I have many people to thank.

My advisor Ajay Jain has been instrumental to my growth. He has been a great teacher and mentor. As we meandered through the various projects, he helped me learn how to think like a scientist. His structured thinking style has really helped me improve how I think about, how I write about, and how I present research. In the moments when we apparently hit dead ends, his continued enthusiasm about new ideas kept us moving forward.

Several other professors have also played an important part in my growth. Patsy Babbitt has been a great advisor. Since I was a first year student with Patsy as my first year advisor, she has always been willing to take the time when I had questions about classes and research. My rotation projects with Ida Sim, Patsy Babbitt, and Ajay Jain were great experiences that opened my eyes to various project areas. Mark Segal, who

11021100101

Abstract

Transcriptional regulation is the control of gene expression, involving interactions between protein transcription factors (TFs), transcription factor binding sites, DNA packing material, and associated genes. Many of the key insights in understanding transcriptional regulation have resulted from wet lab experimentation, but these efforts are often laborious and time consuming. The biological complexity of higher organisms, in terms of larger upstream regions and a larger number of TFs, TF binding sites, and interactions between them, complicates research in this field. Computational modeling, therefore, serves as an important complement to experimental efforts.

This thesis furthers the state of the art in transcription factor binding site modeling, particularly within higher organisms, making use of large-scale computing, machine learning/optimization methods, and high throughput experimental data. The work contributes three important biological results: 1) upstream regions of coexpressed human genes are quantitatively related to the repetitive element structures embedded within these upstream regions; 2) a fast, deterministic motif finder applied to human not only finds annotated binding motifs but also finds biologically relevant co-occurring motifs; and 3) a quantitative model of the TF binding site using a neural network recognizes binding sites better than its position weight matrix counterpart by its ability to

11007 1100 ADV

model positional interdependencies in the binding site. The work was supported by the development of a platform of tools that are well-suited to index-based whole-genome characterizations, which form the basis for very fast algorithms that support direct computation of essentially exact expectation frequencies for large n-mers.

It is hoped that these efforts using large-scale datasets and efficient algorithms will allow further advances in understanding mammalian transcriptional regulation at the sequence level.

Table of Contents

Chapter 1 Introduction	1
1.1. Transcriptional Regulation Biology.....	3
1.2. Higher Organisms	4
1.3. Repetitive Elements	8
1.4. Modeling in Transcriptional Regulation	10
1.4.1. Definition of Modeling	10
1.4.2. Models of the Transcription Factor Binding Site.....	12
1.5. Indexing	14
1.6. Enabling Technologies.....	19
1.6.1. Experimental TF Binding Site Identification.....	19
1.6.2. Expression Microarrays	20
1.6.3. Genome Sequencing	22
1.6.4. Chromatin Immunoprecipitation Arrays.....	23
1.7. Conclusion	23
Chapter 2 Review of Transcription Factor Binding Site Modeling Literature	25
2.1. Introduction.....	25
2.2. Motif Finding	26
2.2.1. Orthogonal Data.....	27
2.2.2. Comparative Genomics.....	28
2.3. Binding Site Recognition.....	31
2.4. Conclusion	32
Chapter 3 HGS: Genomic Mapping	34
3.1. Introduction.....	34
3.2. HGS Implementation	35
3.3. Sensitivity and Specificity.....	38
3.4. Conclusion	38
Chapter 4 Quantitative Relationship of Repetitive Element Structure to Gene Co-expression	39
4.1. Abstract.....	39
4.2. Introduction.....	40
4.3. Results and Discussion.....	40
4.4. Materials and Methods.....	51
4.4.1. Expression Data and Upstream Sequence.....	51

UCCF 1DDADV

4.4.2. Similarity Metrics	53
4.4.3. Repeat Masked Sequences	54
4.5. Conclusion	54
Chapter 5 MaMF: A Deterministic Motif Finding Algorithm with Application to the Human Genome	56
5.1. Abstract	56
5.2. Introduction	57
5.3. MaMF Algorithm Summary	60
5.3.1. Scoring Function	63
5.4. Results	64
5.4.1. Lower Organisms: MaMF Performance	64
5.4.2. Human Data: MaMF Performance	65
5.4.3. Biological Significance of High Scoring Incorrect Motifs	74
5.4.4. Co-occurring Transcription Factor Motifs	78
5.5. Discussion	82
5.6. Materials and Methods	87
5.6.1. Background Model	87
5.6.2. Data	88
5.6.3. Motif Similarity	90
5.6.4. Algorithm Comparison	91
5.6.5. Enrichment Ratio	92
5.6.6. TRANSFAC TF Motifs	93
5.7. Conclusion	94
Chapter 6 ANNFoRM: Nonlinear Transcription Factor Binding Site Recognition Using Neural Networks.....	95
6.1. Abstract	95
6.2. Introduction	96
6.3. Algorithm and Implementation	97
6.3.1. Overview	97
6.3.2. Network structure	97
6.3.3. Negative Training data	98
6.3.4. Training the Network	100
6.3.5. Implementation	100
6.4. Materials and Methods	101
6.4.1. Data	101
6.4.2. General protocol	102
6.5. Results	103
6.6. Conclusion	109
Chapter 7 Conclusion	111
Bibliography	115
Appendix: Documentation of Code and Data	121

Table of Figures

Figure 1. Illustration of the Actors in Transcriptional Regulation.....	4
Figure 2. Types of Transposable Elements in Mammals (Lander, Linton et al. 2001)	9
Figure 3. Sequence Comparison Using Indexes	16
Figure 4. Growth of Genbank	18
Figure 5. Overview of cDNA Microarrays.....	21
Figure 6. Sample output from HGS of the mapping of a sequence	35
Figure 7. Binning Step of HGS Algorithm	37
Figure 8. HGS Performance on Randomly Mutated Sequences.....	38
Figure 9. Illustration of sequence comparison functions	44
Figure 10. Separation using function F on the Staunton and Golub data sets.	45
Figure 11. Expression Correlation in Gene Pairs with Equivalent Alu Count	47
Figure 12. Annotated comparison of p21 and TGFA.	49
Figure 13. MaMF algorithm walkthrough of the CREB/ATF data set.....	62
Figure 14. Performance of MaMF using various parameter settings.....	70
Figure 15. Comparison of motif finding performance between algorithms.	72
Figure 16. Scatterplot of enrichment ratio to motif similarity for the E2F gene set.....	76
Figure 17. Neural network structure.	98
Figure 18. Average ROC areas of ANNFoRM across SCPD data sets.	103
Figure 19. Comparison of ANNFoRM and PWM using Rap1 Lieb data set.	105
Figure 20. ROC plot comparison of a position weight matrix versus ANNFoRM of a representative split from the Rap1 Lieb data set.....	106
Figure 21. Representation of the core six nucleotides of Rap1 RPG set of binding sites found by Bioprospector (w=11).....	107

UCSF LIBRARY

List of Tables

Table 1. IUPAC recommendations for incomplete specification of bases in nucleic acid sequences	13
Table 2. A Position Weight Matrix (PWM) of the CACCCA motif	14
Table 3. Performance of F (ROC area) under various conditions.	46
Table 4. MaMF run on benchmark yeast and <i>e. coli</i> data sets.....	65
Table 5. MaMF run on eight TRANSFAC gene sets.....	67
Table 6. Selected motifs found by MaMF that are highly similar to the annotated motif.....	68
Table 7. Comparison between MaMF, Bioprosector, and Consensus on the Tompa data set.	73
Table 8. Enrichment ratio (ER) skew of MaMF motifs and elevated enrichment ratio of correct motifs on the eight TRANSFAC gene sets.....	77
Table 9. Transcription factor motifs predicted to co-occur with annotated TRANSFAC gene sets	79
Table 10. <i>AHR/HIF</i> target genes that are also responsive to <i>MYC</i>	81
Table 11. Transcription factors predicted to bind onto the <i>MYC</i> promoter.....	82
Table 12. Background models used in the comparison algorithms.	84
Table 13. Average ROC areas compared between ANNFoRM (ANN) and the PWM, for Mcm1, Rap1, and Urs1.	104
Table 14. Four sequences that differ in positions 3 and 6.	108

11007 1100ADV

Chapter 1

Introduction

As of this writing, a confluence of biological and technological innovations continues to sweep the bioinformatics landscape. The three billion nucleotide human genome has been fully sequenced (Consortium 2004) and available to the public. More than 266 other genomes, from *Acinetobacter calcoaceticus* to *Zymomonas mobilis mobilis* have also been sequenced¹. Gene annotation continues at its exponential rate, with Genbank containing over 50 billion bp². Various high throughput biological technologies, including the expression microarray and ChIP arrays, have become increasingly prominent, allowing whole genome analysis of experimental conditions across thousands of genes. Microprocessors have maintained their exponential transistor growth rates, following Moore's law, with recently announced consumer game machines predicted to reach a teraflop in computing capacity.

Progress in each of these areas has enabled bioinformatics to play a greater role in helping to understand biology. Some of this work involves basic computational

¹ <http://www.genomesonline.org/index.cgi?want=Published+Complete+Genomes>

² <http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html>

infrastructure to enable further work, such as building databases of drug interactions that may adversely affect patients, databases of 3D protein structures that precisely pinpoint the atomic positions of each amino acid constituent, and databases of microarray expression experiments that create tens of thousands of data points per experiment. Once the data have been organized, they allow additional projects, such as modeling the physical interactions of small molecules against larger proteins, modeling the metabolic states of enzymes and proteins within a cell, or even simulating entire bodily organs *in silico*.

In each of these cases, two trends are fueling the need for bioinformatics. First, with the advent of high throughput technologies, the availability of biological data is increasing exponentially. Without computers biologists are limited in their ability to process and understand these data. Second, despite these high throughput technologies, some experiments remain difficult and costly to run, limiting the amount of biological experimentation in some areas. These two phenomena have motivated my work in transcriptional regulation.

Transcriptional regulation is the control of the expression of genes using protein transcription factors (TFs) that bind onto specific DNA sequences near the genes, called transcription factor binding sites. These DNA sequences are short (5-15 bp) and can be highly variable. While many subtly different copies of a DNA motif may appear by chance in the genome, transcription factors selectively bind to a specific subset of these sequences. Understanding the determinants of site-specific binding to genomic DNA is critical to elucidating the logic and mechanisms of transcriptional regulation.

11007 1100ADV

The drivers for this work, specifically, are the availability of technologies and data that allow the modeling of aspects of transcriptional regulation, with an emphasis on whole genome computation and higher organisms, particularly human. These include the sequencing of the human genome (and other genomes), availability of large numbers of genes and their annotations, and large-scale experiments such as expression microarrays and ChIP arrays. In this work, I discuss efforts in building transcription factor (TF) binding motifs using neural networks, quantitatively correlating human gene promoter sequences to expression microarray data, and furthering the state of the art in motif finding, specifically by addressing motif finding in human.

To introduce these topics, I discuss the biology of transcriptional regulation in Section 1.1, the motivation to delve into higher organisms in Section 1.2, repetitive elements in Section 1.3, the philosophy and application of modeling in Section 1.4, nmer indexing as the unifying computational strategy for this work in Section 1.5, and finally biological advances that enable this work in Section 1.6.

1.1. Transcriptional Regulation Biology

The genome of an organism has been often described as the ‘blueprint’ of that organism, because it theoretically contains all the information necessary to build a live organism, just as the blueprint of a house contains all the information necessary to build that house. The genome blueprint contains DNA sequence, consisting of a four-letter alphabet. Embedded within the genome are short stretches of DNA called genes. Each gene codifies the information necessary to generate a protein, a fundamental unit in the cell that carries out a specific function to grow and maintain the cell. Because every gene has a specific set of functions, only a subset of genes needs to be expressed as proteins at

11021100101

any given point in time. There is specific cellular machinery to control which genes get expressed, and this system is called transcriptional regulation.

The two primary actors in transcriptional regulation are transcription factors (TFs) and TF binding sites (illustration shown below). TFs are proteins that bind onto specific TF binding sites near the genes, often in the promoter. A given TF will bind to a small set of related binding sites, which are short sequences (5-15 bp long) that differ by several bases. For a gene to be expressed, a set of TFs will bind onto their respective binding sites on the promoter. The correct set of TFs will trigger transcription initiation, which in turn causes a cascade of events leading to RNA polymerase II to produce the relevant mRNA.

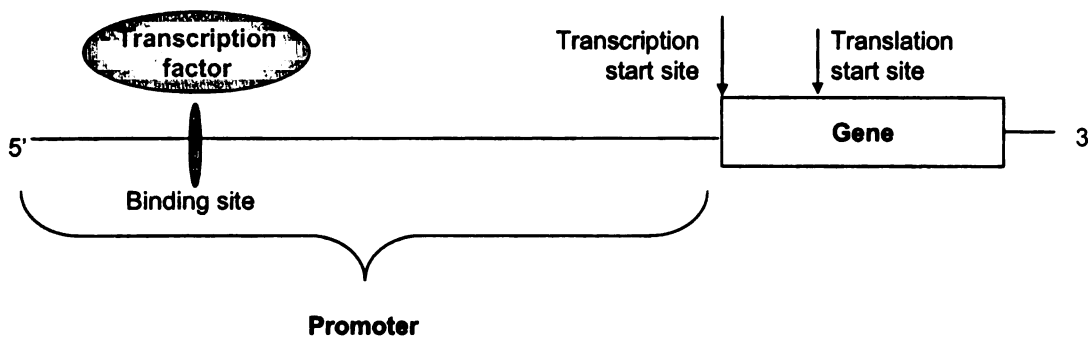


Figure 1. Illustration of the Actors in Transcriptional Regulation

1.2. Higher Organisms

Thus far, computational studies of transcriptional regulation have been limited primarily to lower organisms. Ultimately, we would like these efforts to scale to humans, for multiple reasons. One reason to study humans is that it is a unique genome in many respects, and many complex diseases are governed in part by transcriptional processes (notably many aspects of human cancer). So, deeper understanding of transcriptional regulation may have important implications in development of new therapeutics. From a

theoretical perspective, higher organisms are interesting precisely *because* they have correspondingly richer and more complicated regulatory regions. With respect to the binding site itself, human TF binding sites are much more degenerate than yeast sites, are more likely to have gaps in between, and can be found not only in the 5' of the ATG but also in the introns and even the 3' region (for instance, the P53 target gene P2XM (Urano, Nishimori et al. 1997)). While the regulatory region has increased in size substantially, the binding site remains small and degenerate. With the number of genes not substantially more in human compared with other species, human organism complexity is likely due to more sophisticated regulatory regions, in part by having more complicated transcription factor complex interactions and more elaboration of *cis*-regulatory DNA sequences (Levine and Tjian 2003).

To compare the scope of differences between yeast and human, some numbers are informative. The yeast genome is 12 megabases (Goffeau, Barrell et al. 1996), while human is 2.916 gigabases bases (Lander, Linton et al. 2001), 250 times larger. Despite the dramatically larger size, there are only roughly five times as many genes in human as in yeast (30,000 vs. 6,000). While the average gene length between the two species is approximately the same (1.074 kb for yeast ORFS¹ vs. 1,340 bp in human (Goffeau, Barrell et al. 1996)), human genes have a large number of introns, with a mean of 8.8 introns per gene, with 3,365 bp per intron. By contrast, yeast has a *total* of 220 introns. If the entire genomic extent of the human gene is included, this yields an area of 27 kb per gene. The increase in complexity of the human gene of 25-fold does not account for the

¹ <http://www.dna-res.kazusa.or.jp/10/3/01/HTMLS/node3.html>

250-fold genome expansion of human over yeast. The remainder therefore lies in the intergenic regions, where in yeast only approximately 30% of the genome is intergenic. In human, the intergenic regions contain $3 \text{ Gb} - 1 \text{ kb} \times 30,000 \text{ genes} = 2.97 \text{ Gb}$ or over 99% intergenic. Alternately stated, one expects to see one gene every 2 kb in yeast ($12 \text{ Mb} \div 6,000 \text{ genes}$), and one gene every 100 kb in human ($3 \text{ Gb} \div 30,000 \text{ genes}$). A large portion (44%) of this extra sequence consists of repetitive elements, which include simple sequence repeats such as $(AC)^n$ and $(AT)^n$ and more complicated transposable elements such as the *Alu* element, a short interspersed element (SINE). Factoring out repeats for every gene, one still has to consider 35 times as much intergenic sequence ($100 \text{ kb} - 27 \text{ kb/gene} = 73 \text{ kb}$ in human vs $2 \text{ kb} - 1 \text{ kb/gene} = 1 \text{ kb}$) in human than in yeast when mining for TF binding sites, assuming binding sites normally do not reside in repetitive elements (not always true (Hon and Jain 2003)).

Using existing algorithms tailored for lower organisms will yield minimal success in human because the binding site signal (which is about the same size as that of yeast) is hidden within 35 times more noise, even after ignoring intronic and downstream sequence. In the yeast Rap1 binding site, the conserved core is six bases long (CACCCA), which contains 12 bits of information (where a base contains 2 bits of information). In a uniform distribution of upstream sequence, the oligomer is expected to appear once every 4096 bases. This is sufficiently unique in yeast, which has an average upstream of 1000 bases (from above), but in human with 30000 bases of non-repetitive upstream region, the same sequence is expected to appear six times. A relatively complicated binding site in human is the p53 binding site (representing a relatively high information content), which has two pairs of the consensus 5'-

UICEL IIDDADVA

PuPuPuC(A/T)(T/A)GPyPyPy-3' separated by up to 13 bp (el-Deiry, Kern et al. 1992), encoding 12 bits per half. If the typical deviation from the consensus is three bits, and the two halves can be arranged in 14-choose-2 ways to account for the variable gaps, the sequence is expected to appear one in $2^{24-3} / 91 = 23000$ bp (though probably even more common if considering highly degenerate sites). Thus a random upstream region is expected to contain more than one sequence that matches well with the p53 binding site. Therefore in the human case, finding a true binding site even with high information content is particularly difficult, and since the regulatory mechanism is presumed to be more complicated than yeast, the difficulty only increases.

Several other higher organisms' genomes have been published, serving as potential organisms to study in case human proved to be intractable. One of these is the mouse, which has a genome that is remarkably similar to that of human. Both mouse and human genomes contain about 30,000 genes, where the proportion of mouse genes with a single human orthologue lying in a similar conserved syntenic interval in the human genome is about 80% (Waterston, Lindblad-Toh et al. 2002), but about 99% of human genes have a mouse orthologue. The large number of shared genes makes it possible to look at upstream regions in a systematic manner. Since about 40% the mouse genome can be aligned with the human genome, TF binding sites are potentially enriched by 250% with respect to signal-to-noise ratio, though not all binding sites are expected to be conserved between the two species (if their function recently evolved in one organism or has lost its function). This represents a good intermediary level of evolutionary distance. As the neutral substitution rate has been about half a nucleotide substitution per site since the divergence of the two species, a random 15-mer should be mutated in 7.5 places,

There are four types of interspersed repeats (Lander, Linton et al. 2001). Long Interspersed Elements (LINEs) are the most ancient within eukaryotic genomes. They are 6-8 kb long, embedding an entire reverse transcriptase gene to allow autonomous transcription. Short Interspersed Elements (SINEs) are shorter repeats, 100-300 bp long, embedding an internal polymerase III promoter. As such, they require reverse transcriptase from other sources to cause transposition. Long Terminal Repeats (LTR) are transposon-derived repeats, containing repeats on the same orientation on both sides of a span of transcriptional regulatory elements. Finally, DNA transposons look like bacterial transposons, encoding a transposase. These transposable elements are summarized in Figure 2.

Classes of interspersed repeat in the human genome			Length	Copy number	Fraction of genome
LINEs	Autonomous	ORF1 ORF2 (pol) AAA	6-8 kb	850,000	21%
	Non-autonomous	A B AAA	100-300 bp		
Retrovirus-like elements	Autonomous	gag pol (env)	6-11 kb	450,000	8%
	Non-autonomous	(gag)	1.5-3 kb		
DNA transposon fossils	Autonomous	transposase	2-3 kb	300,000	3%
	Non-autonomous	{ }	80-3,000 bp		

Figure 2. Types of Transposable Elements in Mammals (Lander, Linton et al. 2001)

The highly repetitive nature of repetitive elements affected results in multiple chapters. In Chapter 3, the genomic mapping algorithm requires additional heuristics to address the situation where a short sequence repeat (SSR) may appear many times in a small region of DNA because of SSRs' highly repetitive nature. Another problem of repetitive elements is that they are even more common than many DNA features such as TF binding sites. With their relatively low variability, a set of repetitive elements in a set of promoters looks highly conserved and can be mistaken for TF binding sites. This is a

11021DD1D1

problem in Chapter 5, which was addressed using repeat masking and genomic background probabilities. Repetitive elements do not always cause problems, fortunately. In Chapter 4, a similarity metric found that there is a relationship between coexpressed genes and the repetitive elements contained within them. Specifically, the Alu element, a SINE that comprises 10% of the genome, was a major contributor of this signal.

1.4. Modeling in Transcriptional Regulation

A lot of scientific research involves modeling, but when one stops to think of what modeling means, it is often difficult to explain what that actually entails. What does it mean to “model” transcriptional regulation? What is the point of modeling something if one can experimentally look at it and prod it? At the same time, the tools of modeling, such as neural networks and ontologies, are concrete methods and concepts that allow interesting things to be done, but how do modeling and these computational tools relate to each other? This section introduces what modeling is and the various tools we have used to build our models.

1.4.1. *Definition of Modeling*

In a research rotation, I was introduced to a paper entitled “What is Knowledge Representation?”(Davis, Shrobe et al. 1993). While this paper addresses computer methods in organizing and representing concepts and knowledge, in the broader sense it is modeling knowledge. As such, their definition of knowledge representation is applicable to modeling in any field. The authors break down knowledge representation into five roles, quoted below:

First, a knowledge representation is most fundamentally a surrogate, a substitute for the thing itself, that is used to enable an entity to determine consequences by

UNCF IDP/AD/

sequence is the most important element in the physical structure of DNA, and that we wish to learn more about how this sequence fits with the biology of an organism. With respect to the fourth role, modeling DNA using letters has been an efficient way to analyze and compute on DNA, allowing the entire human genome to fit in a small portion of a typical hard drive. Finally, taken from an anthropological perspective, the fifth role suggests that modeling DNA as letters is a form of human expression because as humans we think in terms of language and understanding. If DNA is an alphabet for a biological language where our genome contains the evolutionary history of humans and our ancestor species, by understanding the genome we understand our past and ourselves.

1.4.2. *Models of the Transcription Factor Binding Site*

This thesis is entitled “Transcription Factor Binding Site Modeling in Higher Organisms” because we have taken the TF binding site and represented it using various models to facilitate computation on it. We refer to the space of different binding sites to which a TF binds as a TF binding motif, and the following descriptions are models of this concept. In its simplest form, the TF motif can be represented as the average DNA sequence to which the TF binds, for instance CACCCA for the Rap1 motif in yeast (Moretti, Freeman et al. 1994). This consensus sequence is often used in papers to describe a TF motif because it is easy to understand and print in a paper. Because a TF binds onto a set of different binding sites, a simple extension to the consensus sequence is to allow mismatches at a given position, using IUPAC symbols shown below in Table 1. In Chapter 4 we develop a similarity metric that is based on the idea that biologically relevant features in two promoter regions of coexpressed genes, such as TF binding sites, might be detected if they shared the exact same sequence. Because coexpression is likely

to result from similar transcriptional machinery being activated, we hypothesize that this similarity metric would be sensitive enough to identify shared TF consensus sequences. The assumption is that one can find TF binding motifs using a consensus model where an exact n -mer match is equivalent of finding a shared consensus. While this does not turn out to be true, we find that the presence of repetitive elements was very related to coexpression.

Table 1. IUPAC recommendations for incomplete specification of bases in nucleic acid sequences¹

Symbol	Meaning	Origin of designation
G	G	Guanine
A	A	Adenine
T	T	Thymine
C	C	Cytosine
R	G or A	puRine
Y	T or C	pYrimidine
M	A or C	aMino
K	G or T	Keto
S	G or C	Strong interaction (3 H bonds)
W	A or T	Weak interaction (2 H bonds)
H	A or C or T	not-G, H follows G in the alphabet
B	G or T or C	not-A, B follows A
V	G or C or A	not-T (not-U), V follows U
D	G or A or T	not-C, D follows C
N	G or A or T or C	aNy

Since an important quality of TF binding motifs is that there are often nucleotide mismatches, a more convenient representation that makes it possible to specify how common nucleotide mismatches are is the Position Weight Matrix (PWM). Each position in the binding motif now contains a probability for each of the four nucleotides, creating a $4 \times n$ matrix for a motif of width n . An example of the CACCCA motif is shown in

¹ <http://www.chem.qmul.ac.uk/iubmb/misc/naseq.html>

IUCR IUPAC

Table 2, with the relevant positions in boldface. Thus one can specify the likelihood of seeing a C at position 5 for the Rap1 motif from above. In Chapter 5, we describe a motif finder that uses a PWM as the model of a binding motif.

Table 2. A Position Weight Matrix (PWM) of the CACCCA motif

	1	2	3	4	5	6
A	0.01	0.8	0.03	0.04	0.1	0.8
C	0.9	0.05	0.95	0.8	0.85	0.03
G	0.04	0.1	0.01	0.07	0.04	0.06
T	0.05	0.05	0.01	0.09	0.01	0.11

One important limitation of the PWM is that it cannot represent interdependencies between positions in the binding motif (discussed in detail in Section 2.3). For instance, a PWM cannot model the condition that if position 1 is a C then position 2 is a G. Our work in this area (Chapter 6) uses a neural network to model the TF binding motif for the purpose of TF binding site recognition. In general the tradeoff between these models is that the more sophisticated models require more computational resources to use them effectively and more data to define them adequately, but make fewer assumptions about the underlying physical reality. As such, understanding the philosophical issues of modeling prepares the researcher for understanding given the model what type of questions can be asked and what the computational limitations are.

1.5. Indexing

Bioinformatics has a long tradition of adopting algorithms from computer science and related fields, and adapting them for biological analyses. Some of these include dynamic programming, neural networks, Gibbs sampling, and Hidden Markov Models. While there is a large body of work devoted to these methods, simpler algorithms that are just as important tend to get less attention. One such algorithm is the index (or hashing),

IISSE LIBRARY

which are commonly used within computer science and critical to many programs. For instance, the Perl programming language has a hash variable type built-in. Additionally, a fundamental performance enhancement for databases is the use of indices, which allow quick access to specific rows within a table.

The general formulation of a hash table is the following: Given a set of key/value pairs of arbitrary size, a hash allows the constant time insertion and lookup of the value of any key. The way the constant time lookup is achieved is to have a hash function that condenses a key into a numerical value of limited range $0 \dots h$. For example, the MD5 hash function¹ converts data of arbitrary size into a 16 byte integer and is used to quickly compare the identity of two blocks of data. Using an array also of size h , we can put the key and value into the array at the corresponding position of the hash output. When the value for the key is asked for again, we can use the hash function to convert the key into a number, and retrieve the data from the corresponding cell in the array. Since the possible values for a key tends to be substantially larger than the size of h , many keys may map to the same value using the hash function. Using a carefully chosen hash function minimizes such key collisions, but in the event they do occur there are ways to keep track of the data and efficiently retrieve the correct key/value pair.

In this work, indexing is an important yet simple strategy for computing exact sequence alignments. The goal is, for a pair of sequences of length x and y , to calculate all alignments of n mers of size n (typically 4-6 bp). To do this, the first step is to create an index of n mers for each sequence. Our hash function is a one-to-one conversion of an

¹ <http://www.ietf.org/rfc/rfc1321.txt>

nmer to a number: the nucleotide sequence can be thought of as a base-4 number with A=0, C=1, G=2, and T=3 and then converted to a decimal. As such, we do not have to worry about key collisions. The index consists of an array of size 4^n , where each cell in the array contains the positions (0 or more) in the sequence that represent the nmer equivalent of the cell. The index is populated by scanning through a sequence, converting each nmer to an index offset, and recording the position of the nmer into the index. At this point one can ask, in constant time, at what positions can a particular nmer be found in a sequence?

<u>Sequence A</u>			Two sequences, A and B, are shown. Each has two occurrences of the 4-mer ACGT, which is entered in the index along with other 4-mers (the rest not shown). ACGT matches can easily be computed by enumerating the combination pairs possible, in this case $2*2=4$.
ACGT (251)	ACGT (624)		
<u>Sequence B</u>			
ACGT (347)	ACGT (478)	AAAA (892)	
Seq A Index	Seq B Index	ACGT Matches	
AAAA	AAAA = 892	Seq A, 251 and Seq B, 347	
...	...	Seq A, 251 and Seq B, 478	
ACGT = 251, 624	ACGT = 347, 478	Seq A, 624 and Seq B, 347	
ACTA	ACTA	Seq A, 624 and Seq B, 478	
...	...		
TTTT	TTTT		

Figure 3. Sequence Comparison Using Indexes

Given indices of two sequences, the corollary question is, where do the two sequences align and have a given nmer shared between the two sequences? This question is similarly easy to answer using the indices. If the nmer in question is ACGT (for $n=4$), we can quickly obtain the positions $x_1 \dots x_a$ in sequence 1 that have ACGT and also quickly obtain the positions $y_1 \dots y_b$ in sequence 2 that have ACGT. The alignments between sequence 1 and sequence 2 that have ACGT is simply the combination of both sets of positions, $(x_1, y_1), (x_1, y_2), \dots, (x_a, y_b)$ giving $a \cdot b$ combinations. By enumerating all nmers of size n and computing the alignments for each nmer, we can calculate all exact

UCSF LIBRARY

nmer matches between sequence 1 and sequence 2. An example of this process is shown above in Figure 3.

The time complexity of this operation is as follows. For two sequences of length s , the creation of the indices takes time $s \times 2$, since we traverse both sequences in linear time to create the index. To generate all exact nmer matches of nmer size n , we iterate through all 4^n nmers. The expected number of positions present for a given nmer for a given sequence is $s/4^n$, and the expected number of exact nmer matches for a given nmer is therefore $(s/4^n)^2$. To do this for all nmers, we expect $4^n \cdot (s/4^n)^2$ operations. Putting this all together, the rough time complexity is $2s + 4^n \cdot (s/4^n)^2 \approx O\left(\frac{s^2}{4^n}\right)$. From this it is apparent that computing exact nmer matches using this method is fundamentally quadratic time relative to the size of the input sequences (like many sequence comparison methods), but larger nmer sizes strongly mitigate this cost, at exponential rates. In practice, nmer sizes of 4 and up show significant performance benefits. In Chapter 4 we show that we can compute all exact nmer matches for $n=6$ for two sequences of size $s=10000$ in several seconds.

UICST LIBRARY

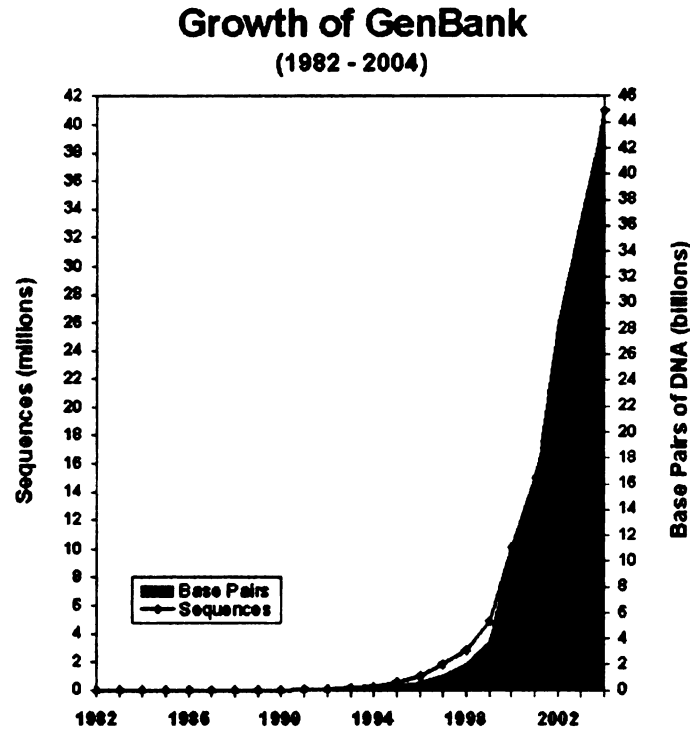


Figure 4. Growth of Genbank¹

This indexing technique is most amenable to problems that require high speed with a tradeoff in sensitivity. Within bioinformatics, the BLAST algorithm (Altschul, Madden et al. 1997) may be the most well known algorithm to use indexing. The problem BLAST addresses is, given an input sequence, to find other similar sequences within a large database of sequences. As of 2004, there are 40 million sequences in Genbank, and this number is growing at an exponential rate shown in Figure 4. The straightforward method of calculating sequence identity between the input sequence and every entry in the database, one by one, is clearly infeasible. Therefore, BLAST instead looks for “hints” of similarity inside the database by searching for exact matches (typically of size

¹ <http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html>

15) between the input sequence and sequences inside the database, using a similar indexing strategy above. If there is a hit, it measures the sequence identity by extending the alignment outwards from the exact nmer match. BLAST manages a tradeoff between sensitivity and speed by quickly finding potentially interesting sequences and weeding out sequences without any exact nmer matches, but increases sensitivity by doing further analysis on a matching sequence.

Multiple algorithms in this work used indexing as an important computational tool to increase speed. Chapter 3 describes HGS, a genomic mapping algorithm that uses 9-mers of the target genome to quickly pinpoint likely locations of the input sequence. Chapter 4 uses indexing as a basis for creating several high-speed sequence similarity metrics. Finally, Chapter 5 presents MaMF, a motif finding algorithm that uses indexing to quickly generate sequence alignments to be used to build motifs.

1.6. Enabling Technologies

This section discusses the primary experimental technologies used to generate the biological data used in this work. Experimentalists have used the techniques described in Section 1.6.1 to generate the datasets for genes regulated by the same TF used in Chapter 5 and Chapter 6. Microarray data were used extensively in Chapter 4 and Chapter 5. The human and yeast genomes play an important part in every project pursued. ChIP arrays were used in Chapter 6.

1.6.1. *Experimental TF Binding Site Identification*

The general strategy for determining the binding site of a particular TF on a particular gene is to perform a series of experiments that isolate the region of the

promoter to which the TF binds. The promoter region in question is attached to a reporter gene. By using various restriction enzymes to cleave portions of the promoter region, one can examine if the TF binding site lies within the remaining sequence by checking for remaining transcriptional activity. Once reduced to the minimal promoter, DNase footprinting provides a closer view of which region the TF binds. DNase I cuts DNA at random locations, so that by applying DNase I on promoter sequence with and without the bound TF, it is possible to isolate the region of DNA that is protected by the bound TF. Finally mutation studies can be performed to analyze which specific nucleotides confer transcriptional activity. From this description, it is apparent that in order to obtain accurate TF binding site locations a large amount of detailed experimental work needs to be done, preventing large scale analysis of TF binding sites using these methods.

1.6.2. *Expression Microarrays*

Expression microarrays are a powerful method of measuring the expression of a large number of genes for a given cell samples under specific conditions. The basic principle is that once the sequence of the gene is known, one can create short probes that bind to this gene using the reverse complement of the gene sequence. Given probes for all the genes of interest, these probes can be printed onto an array using robotics. Cell samples containing mRNA are fluorescently labeled and then introduced to the array. mRNA from highly expressed genes will bind to their appropriate spot in large quantities and fluoresce brightly. The output is an image recording the amount of fluorescence at each spot on the array.

There are two implementations of microarrays that are currently used, the cDNA microarray, and the oligo microarray (Affymetrix). The cDNA microarray have several

properties different from the oligo microarrays. The cDNA microarray uses longer stretches of cDNA (60-70 bp long) as its probes, so that in general a probe uniquely identifies the gene. The microarray is also printed on glass slides. Additionally, expression is measured as a ratio because two cell populations are considered: the control population (normal cells) and the treated population (the cells of interest). After isolating the RNA for each population, different colored dyes (Cy3 and Cy5) are attached to the RNA. The two sets of RNA are mixed and then applied to the microarray. The resulting image measures the ratio of the two dye colors, typically in red and green.

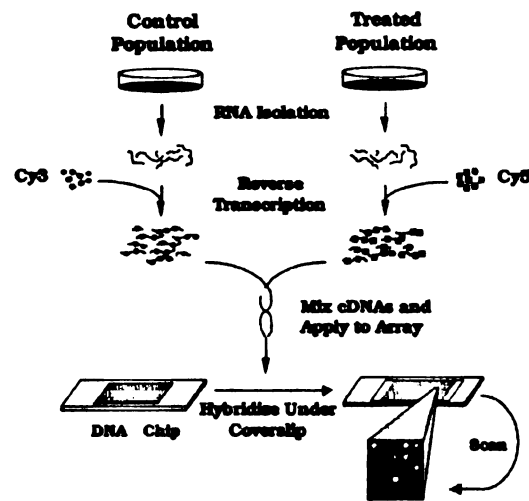


Figure 5. Overview of cDNA Microarrays¹

In contrast, Affymetrix oligo microarrays use a slightly different strategy. Instead of a single longer probe, shorter 25 bp probes are used. The shorter probes have two important consequences: 1) short sequences do not bind onto their complement as well as

¹ <http://bioinformatics.mdanderson.org/MicroarrayCourse/Lectures/>, Lecture 1, 31 August 2004,

Introduction to Microarrays, slide 27

UCSF LIBRARY

longer sequences, and 2) they may cross-hybridize with unintended targets. Affymetrix addresses this in two ways. First, they use a perfect match probe along with a mismatch probe that differs from the perfect match probe by one nucleotide; the mismatch probe serves as a control to measure the background amount of cross-hybridation. Second, since a given probe may bind onto multiple genes, multiple probes are used. Since oligonucleotide probes are short and small, they can be printed onto silicon using photolithography, borrowing a technique from the microprocessor world. Because oligo microarrays do not use a two-spot technique, the images contain a single fluorescent dye that results in a single absolute number. In addition, since multiple probes are used, Affymetrix has an algorithm to merge the values of multiple probes into a composite expression value.

1.6.3. *Genome Sequencing*

The general strategy of sequencing a genome is to break apart the genome into reasonably sized fragments, sequence each of these fragments, and then piece the sequenced fragments back together. The basic technologies for large scale genome sequencing have been around since the 1970s. Soon after DNA sequencing methods were invented (Sanger and Coulson 1975; Maxam and Gilbert 1977), the shotgun sequencing strategy was introduced (Anderson 1981). Since then a series of various improvements have eventually allowed the sequencing of longer DNA, with the milestone of the human genome sequenced in 2001 (Lander, Linton et al. 2001). These advancements include improvements in protocols for fragmenting and cloning DNA, the ability to apply sequencing to larger DNA molecules, automation in collecting raw DNA sequences,

UCSF LIBRARY

strategies to deal with repetitive sequence in higher organisms, and software to analyze and assemble sequences (Odom, Zizlsperger et al. 2004).

1.6.4. Chromatin Immunoprecipitation Arrays

A promising high throughput method to identify genes responsive to a particular TF is the chromatin immunoprecipitation array, or ChIP array. The general procedure is to use chromatin immunoprecipitation to enrich DNA that has the protein of interest bound to it, and then identify what the DNA is using a microarray. Specifically, starting with cells *in vivo*, the TF is bound to the DNA using a crosslinking procedure (with formaldehyde). Afterwards, the cells are broken open and DNA sheared into smaller pieces using sonication. The DNA is enriched using immunoprecipitation to a specific antibody, and then the bound TF is removed using reverse-crosslinking. Finally, using a microarray containing the promoter regions of genes, the locations of the enriched DNA are identified. In yeast, the entire intergenic regions can be used since they are relatively short, but in human current efforts have focused on the 1000 bases around the transcription start site of genes (Odom, Zizlsperger et al. 2004).

1.7. Conclusion

Chapter 1 introduced the topic of transcriptional regulation and some of the biological and computational issues discussed in this work. In terms of the biology, Sections 1.2 and 1.3 discussed the challenges of higher organisms and repetitive elements, respectively. On the computational side, Sections 1.4 and 1.5 discussed modeling transcriptional regulation and the use of indexing make certain problems

UCSF LIBRARY

computationally tractable. Finally, Section 1.6 gives a brief overview of the various enabling technologies that provide the data used in the thesis.

The next chapter goes into more depth about existing efforts in modeling transcription factor binding sites.

UCSF LIBRARY

Chapter 2

Review of Transcription Factor

Binding Site Modeling Literature

2.1. Introduction

Computational approaches to understanding TFs and their binding sites hinge on building the best model for the job. There are two primary strategies to developing better models. One strategy is to create more sophisticated models that better represent the physical interaction of TF to binding site. Another strategy is use additional data sources to enable a more detailed model to be created.

The two central problems that comprise TF binding site modeling have generally used different strategies. Motif finding aims to predict binding motifs shared by a set of coregulated genes assumed to be governed by the same TF. In general, because the search space is large, simpler models such as the PWM are preferred to reduce the computational requirements, and so emphasis has been put on utilizing new sources of data, the most promising of which is comparative genomics. These approaches are discussed in Section 2.2. The second problem, binding site recognition, aims to identify new biologically relevant binding sites given known binding sites responsive to a TF.

UCSF LIBRARY

Because of the physical complexity of the TF binding interface, many efforts in this area focus on building better models that more accurately reflect the interactions of binding sites and TFs. These efforts are discussed in Section 2.3, which was introduced in Section 1.4 (Modeling in Transcriptional Regulation).

2.2. Motif Finding

Traditional experimental methods to derive binding sites for a given transcription factor are labor intensive (as shown in Section 1.6.1); consequently many computational techniques have arisen to identify binding sites within promoter regions. The general form of the problem is, given a set of promoters corresponding to genes either postulated or known to be responsive to a particular TF, to find a sequence motif that represents the true biological target of the TF. The general solution is to find conserved sequences within the collection of promoter regions that are under-represented relative to the genome. There are a large variety of published techniques resulting from the maturity of this field.

Motif finding algorithms can be classified into two broad groups. Since computational capacity has only recently been plentiful, local search algorithms utilizing clever ways to reduce the search space have traditionally been the most studied. The forefathers of many of these approaches are CONSENSUS (Stormo and Hartzell 1989), which uses a greedy strategy to derive a position weight matrices of TF binding sites; MEME (Bailey and Elkan 1994), which uses an expectation maximization technique to fit a model to sequences; and Gibbs sampler (Lawrence, Altschul et al. 1993), which uses Gibbs sampling statistical technique. Gibbs sampling has perhaps created the most interest, yielding variant algorithms including AlignACE (Roth, Hughes et al. 1998) and

UCSF LIBRARY

Bioprospector (Liu, Brutlag et al. 2001). Down and Hubbard (Down and Hubbard 2005) have contributed improvements to expectation maximization in their NestedMICA algorithm. MaMF, described in Chapter 5 uses the greedy search method. Since computing power has been increasing exponentially, enumerative approaches, which generate all possible motifs up to a certain length to find the best motif, have recently become possible (van Helden, Andre et al. 1998; Jensen and Knudsen 2000; Birnbaum, Benfey et al. 2001; Sinha and Tompa 2003).

2.2.1. *Orthogonal Data*

There is often not enough information embedded in the sequence alone to make reliable computational binding site predictions. Furthermore, algorithms that work in lower organisms like yeast become less effective in more complicated organisms like human. To address these issues, researchers have employed additional information to help algorithms focus on more interesting sequences:

- The availability of full genome sequences allows researchers to calculate background frequencies of the genome, which algorithms like Bioprospector (Liu, Brutlag et al. 2001) and ANN-Spec (Workman and Stormo 2000) employ. The background frequency helps filter out common sequences like repetitive elements so that algorithms can focus on overrepresented, uncommon sequences.
- Expression microarrays have allowed the clustering of coexpressed genes. Assuming that a cluster of genes with similar expression profiles are governed by the same transcription factors, algorithms can search for binding sites shared by all the genes (Keles, van der Laan et al. 2002; Zhu, Pilpel et al. 2002; Conlon, Liu et al. 2003). A limitation with this approach is that a given gene may be regulated

UCSF LIBRARY

by multiple transcription factors, but can only belong to one cluster, which motivated a fuzzy clustering technique (Gasch and Eisen 2002).

- ChIP arrays generate exhaustive data about which genes a particular transcription factor targets. MDScan (Liu, Brutlag et al. 2002) exploits the fact that some genes bind more tightly to the transcription factor and are therefore more likely to be true targets, by first considering the top genes first and then expanding the search.
- Bacterial genomes often have binding sites that have two parts separated by a gap. By using this domain knowledge, one approach has focused on finding overrepresented dimers solely using genome sequence (Li, Rhodius et al. 2002).

2.2.2. *Comparative Genomics*

One of the most promising sources of information is the availability of genomes of multiple species. A possible approach to finding TF binding sites is to use phylogenetic footprinting (or comparative genomics), which is the comparison of shared sequences between multiple closely related species. As genome sequencing technology has matured, full genomic sequences of multiple genomes are becoming available more rapidly. Since evolution predicts that functionally active sequences in the genome (genes, transcription factor binding sites, etc.) will be conserved, one expects that the intersection of upstream regions of gene orthologues will contain conserved sequences.

Deciding which species to use for comparative genomics can be complicated. McCue et al. (2002) observe that three species related to *E. coli* were required for 74% of motif predictions to match with the experimentally reported binding sites. Similarly, Cliften et al. (2001) note that within *S. cerevisiae* at least three related species are required to be able to align orthologous sequence and extract statistically significant

conserved sequences. Furthermore, the evolutionary distance of the genomes can affect the utility of the approach. Unrelated species will have highly divergent sequences that are not amenable to alignment, and similar species will have nearly identical sequence that make it difficult to differentiate functionally conserved sequence and nonfunctional unconserved sequence. A pair of genomes with intermediate evolutionary distance would maximize the conserved functional signal while minimizing the noise from unconserved nonfunctional sequence. Because of this complexity, several approaches have used multiple genomes on which to perform comparative genomics:

- McCue et al. (2001) have applied a Gibbs sampling technique to find conserved binding sites from gene orthologues in as many as nine gamma proteobacteria species. The predictions were made without knowledge of transcription factors governing a set of genes, but incorporate knowledge of palindromic patterns common in *E. coli*, the probability of seeing a given motif (i.e. a position-specific background model), and expectations of positional spacing of binding sites.
- Cliften et al. (2003) fully sequenced six genomes related to *S. cerevisiae* in order to perform comparative genomics. Utilizing multiple sequence alignments using CLUSTALW, exact matches between the four sensu stricto species or between all six species found about 8000 conserved 6- to 30- oligomers in both cases, with the confidence level increasing with nmer size. Thus with multiple genomes it is possible to find conserved sequences using a relatively simple procedure.
- Traditional motif finders have also been extended to support multiple species, such as CompareProspector (Liu, Liu et al. 2004), which was based on

UCSF LIBRARY

Bioprospector, and PhyloCon (Wang and Stormo 2003), which was based on Consensus.

- Phylogenetic shadowing, the comparison of sequences of closely related species, was coined by Boffelli et al. (2003) in their comparison of Old World and New World monkeys and hominoids. They analyzed apolipoprotein, a protein specific to primates, by sequencing a 1.6 kb region in 18 Old World monkeys and hominoids, and were able to verify that several conserved regions were functionally active, using electrophoretic mobility-shift assays and transfection analysis.
- Footprinter is an algorithm that explicitly incorporates the phylogenetic tree in its computation of conserved sites within a set of upstream regions of orthologous genes (Blanchette and Tompa 2002). It uses a dynamic programming approach to calculate alignments. It was successful in extracting binding sites from datasets containing multiple species.
- Comparative genomics has been used successfully human to systematically find regulatory elements in promoters and 3' UTRs (Xie, Lu et al. 2005). As in the case of yeast, they required several organisms (human, mouse, rat, and dog) to do a meaningful alignment to find conserved binding sites.

With the recent availability of several fully sequenced organisms evolutionarily close to human, comparative genomics in the near future will likely play a large role in human transcription factor binding site prediction.

UCSF LIBRARY

2.3. Binding Site Recognition

A motivation to build a TF binding site model is so that ultimately the model can be used to identify new binding sites that are biologically relevant. To identify a potential binding site, the binding site model needs to have an associated scoring function that determines the similarity of the binding site to the model. A binding site that scores highly therefore shares many characteristics with the model. For example, using a consensus model of the binding site, the number of matches between the putative binding site and the consensus sequence indicates the amount of similarity shared. Likewise, using a position weight matrix (PWM), using the instantiation of the putative binding site from the PWM yields gives the score similarity.

The consensus model and PWM have been used extensively for database searches of binding sites within promoter regions. TFSEARCH uses weight matrices derived from TRANSFAC to search for binding sites in input sequences¹. TESS is a web tool that uses consensus sequences and PWMs to search for binding sites (Schug and Overton 1997).

A major limitation of the consensus sequence and PWM is that they assume that the nucleotides within a binding site have independent effects. For instance, a PWM cannot represent the conditional case “if position 2 in the binding site is an A, then position 5 should be a G.” It has been shown that *in vivo* the independence assumption does not hold (Benos, Bulyk et al. 2002). However, because current algorithms work with limited data, others argue that independence is not a bad assumption to make, as it

¹ http://www.cbrc.jp/papia/howtouse/howtouse_tfsearch.html, TFSEARCH: DNA Transcription Factor

reduces the parametric complexity of the induced models (Benos, Bulyk et al. 2002).

With the availability of increasing amounts of data (see Section 1.6), limited data is becoming less of a problem.

Several algorithms have attempted to address the dependency issue within binding sites. Zhou et al. (2004) have extended the PWM to include pairs of correlated positions and then use a Markov chain Monte Carlo algorithm to generate a model that fits the model space. Elrott et al. (2002) used a Markov chain optimization method to build models of the HNF4A binding site and find new binding sites that were later experimentally verified.

Other algorithms have integrated external data into novel models of the binding site. One method used a boosting approach by integrating ChIP array data (see Section 1.6.4) into their motif finding method and then created a nonlinear classifier of true binding sites by using several weight matrices (Hong, Liu et al. 2005). Another strategy aims to consider diverse characteristic binding site properties, such as the binding site position relative to the transcription start site and structural properties, in developing a Bayesian network (Pudimat, Schukat-Talamazzini et al. 2005). A support vector machine approach was used in estimating the sequence-specific binding site energy of a particular TF in Djordjevic et al. (Djordjevic, Sengupta et al. 2003).

2.4. Conclusion

In summary, TF binding site modeling has had an increasingly rich history as new technologies have arisen. Motif finding and binding site recognition algorithms have used these new data to build more sophisticated models, both in terms of novel representations of the TF binding site and more detailed models integrating the larger amounts of data. I

UCSF LIBRARY

discuss efforts in motif finding in Chapter 5, which describes a motif finder that uses a greedy approach similar to Consensus. Chapter 6 discusses TF binding site recognition using neural networks, providing another way to model interdependencies in the binding site.

The next chapter presents a real world application of indexing, introduced in Section 1.5, by applying it to genomic mapping. Indexing is a major theme of this thesis, and variants of the approach are further explored in Chapter 4 and Chapter 5.

UCSF LIBRARY

Chapter 3

HGS: Genomic Mapping

3.1. Introduction

This chapter gives an example of indexing in use in a real world program.

Genomic mapping is the problem of locating specific genomic sequence, like a gene, within a target genome, given only the sequence. It is useful for identifying the genomic locations of a collection of genes or for obtaining contextual sequence around a gene, for instance. In the first case, the genomic locations of a collection of genes have been used to link chromosomal genomic hybridization (CGH) arrays with microarray expression data. Specifically, one can map the genomic locations of genes used in the expression microarray to the BAC clone locations in the CGH array to measure the relationship between expression of genes and chromosomal aberrations in the same genomic loci. In the second case, obtaining contextual sequence has been important for my work since promoter sequence has not always been readily available. However, given the relative abundance of gene sequences and fully sequenced genomes, one can calculate the upstream sequence of genes using genomic mapping. This technique was used extensively in Chapter 4 to obtain 10,000 bp of sequence upstream of human genes.

UCSF LIBRARY

Because techniques to perform genomic mapping were not readily available till recently, we implemented a genomic mapping algorithm, called HGS (short for Human Genome Search), to facilitate my work and others' in the Cancer Center. Because the algorithm relies on the assumption that a near exact copy of the genomic sequence can be found within the target genome, the indexing techniques described above complement that algorithm extremely well. HGS employs a disk-based index of the genome and additional heuristics to pinpoint the location of a query sequence. For each FASTA sequence entered, HGS returns the chromosome number and base positions the sequence spans, which strand the sequence appears, and supporting information that indicates the confidence of the result (shown below). Furthermore, the program supports a batch operation that allows multiple sequences to be processed at a time. Additionally, compared with BLAT (Kent 2002), a program with similar function but which uses a very large memory-resident index for human, HGS can run on Windows using a typical desktop computer, making it more user-friendly. Typical query sequences against the full 3 Gb human genome take 10 seconds on standard hardware, enabling practical batch mapping of thousands of sequences.

```
!>D20015 gi|500912|dbj|D20015.1|D20015 HUMGS00986 Human promyelocyte Homo sapiens cDNA
clone pm2347 3'
  Iter  Score Coverage Start-End   Qsize Identity Strand Chr:      Start-End
!10    342    99.8%      1-499     500    68.5%   -  chr17: 45177621-45179185
  10     99    52.8%     236-499   500    37.5%   +  chr1: 89713697-89713955
  10     81    52.2%     236-496   500    31.0%   -  chr1: 88854026-88854281
```

Figure 6. Sample output from HGS of the mapping of a sequence

3.2. HGS Implementation

To illustrate how the indexing techniques can be used, the implementation of HGS is outlined below. HGS is broken into two parts. The first is a one-time indexing procedure that indexes the target genome to provide quick lookup in the actual search

UCSF LIBRARY

procedure. This corresponds exactly to the indexing of a sequence described in Section 1.5, except in this case each sequence is a chromosome. As suggested earlier, the choice of nmer size affects the speed and sensitivity of the algorithm. Since speed is the more important consideration in this algorithm, an nmer size of 9 was used. Given the human genome has 3 billion bp and there are 4^9 slots in the index, a given nmer on average appears 1000 times in the genome. Information about a set of nmers, obtained from the input sequence, could then pinpoint that sequence on the genome.

The second part of the algorithm is the main search procedure, which can be run repeatedly given the pre-computed index. This procedure has two steps, a binning step followed by a chaining step. The binning step isolates the regions that the input sequence maps to, by creating bins for every 2000 bp of genomic sequence. The input sequence is broken into 9 bp segments, and the locations of each of these segments are retrieved from the index. For each location, a tally for the corresponding bin is incremented, so that after all segments are processed bins with high scores likely correspond to matches to the input sequence. This is shown below in Figure 7.

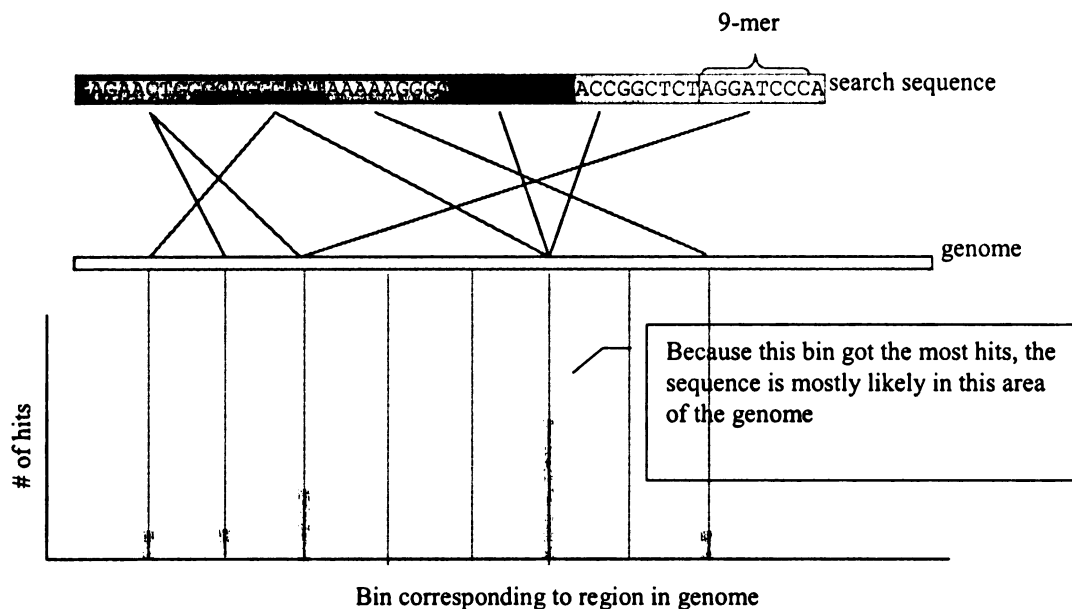


Figure 7. Binning Step of HGS Algorithm

There are two complications that require the second chaining step. First, repetitive sequence may skew the bin scores such that false positives could arise. Second, the input may map to multiple regions on the genome since the primary input for this algorithm is gene sequence, which generally is stored with introns removed. Thus a 1000 bp gene sequence might map to tens of thousands of bp.

HGS uses several strategies to handle these complications. To minimize the effects of repetitive sequence, a second binning is performed on the high scoring primary bins, where these secondary bins are 9 bases and allow at most one hit. Highly repetitive sequences, such as a poly-A tail, therefore would register only one hit every 9 bases, instead of 9 hits every 9 bases in the primary binning step. To account for sequence spanning over large distances, HGS analyzes the nmer matches within each high scoring bin to make sure the nmer matches corresponding to the input sequence occur in order and address the full extent of the input sequence. For each high scoring bin not processed, HGS first verifies that the nmer matches occur in order, and then considers nearby bins

that contain ordered nmer matches that can be chained to the previous bins. After one set of bins has been processed, HGS continues with the other high scoring bins. The results are presented with accurate identity and coverage information calculated by performing a Smith-Waterman (Smith and Waterman 1981) alignment over the matching sequence.

3.3. Sensitivity and Specificity

While HGS was designed primarily for speed, it was also designed to tolerate a reasonable amount of sequence deviation. This tolerance is an important feature because the sequence of the same gene obtained from different sources will often have mismatches, due to various experimental and technology issues. As such we measured HGS's ability to tolerate these bases mismatches in genomic mapping using the following procedure. Taking random sequences of varying lengths from the genome, we mutated these sequences various amounts before running them through HGS. Figure 8 shows these results, where sequences that are greater than 250 bp long and have less than 15% mutation compared with the genome can be found quite reliably.

		Mutation Percentage			
		0	1	10	20
Sequence Length	1000	96	96	98	87
	500	96	95	96	62
	250	93	95	87	42
	50	79	79	20	5

Figure 8. HGS Performance on Randomly Mutated Sequences

3.4. Conclusion

This chapter introduced a real world application of indexing in the form of HGS, a genome mapping algorithm. It was used extensively in Chapter 4, and provides the code base for additional work using indexing in the following chapters.

UCSF LIBRARY

Chapter 4

Quantitative Relationship of Repetitive Element Structure to Gene Co-expression

4.1. Abstract

A sequence similarity metric operating on 10 kilobase upstream regions of gene pairs quantitatively predicts a portion of co-variation of expression of gene pairs in large-scale gene expression studies in human tumors and tumor-derived cell lines. The signal on which the metric depends most strongly originates in the compositional structure of repetitive genomic sequences (particularly *Alu* elements) present in these upstream regions. This effect is completely separable from effects of isochore composition on gene expression. The results implicate repetitive elements with some functional role in transcriptional regulation of the specific genes in whose promoter regions they reside and lend credence to suggestions that the general phenomenon of repetitive element insertions may be a fundamental evolutionary mechanism for modulating gene transcription.

UCSF LIBRARY

4.2. Introduction

The previous chapter discussed a method for genomic mapping using a simple indexing technique introduced in Section 1.5. This chapter elaborates on this method by applying indexing to sequence comparison of co-expressed genes, in the hopes of identifying DNA features such as TF binding sites shared between co-expressed genes. Many methods have been reported exploring the potential for identifying features in DNA that are responsible for co-expression of gene pairs or gene families (Stormo 2000). However, even in the context of lower organisms such as yeast, there is a significant problem of specificity in the computational models of transcription factor binding (Stormo 2000). The potential requirement of cooperation among multiple transcription factors is being studied computationally by many researchers, again primarily in lower organisms (Wagner 1999; GuhaThakurta and Stormo 2001; Liu, Brutlag et al. 2001).

Here, we take a different approach in two respects. First, we consider *human* gene transcription quantified by expression arrays. Second, rather than looking for specific transcription factor binding sites, we develop a sequence similarity metric for *any pair* of genes to predict the likelihood that the gene pair will be co-expressed. A similarity metric that depends most strongly on the compositional structure of repetitive genomic sequences (particularly *Alu* elements) is able to partially separate gene pairs that are co-expressed from gene pairs that are not.

4.3. Results and Discussion

We considered three microarray data sets: cDNA-based expression data from human cancer derived cell lines, the Ross set (Ross, Scherf et al. 2000); oligo-based

expression data on the same cell lines, the Staunton set (Staunton, Slonim et al. 2001); and oligo-based expression data from acute leukemias, the Golub set (Golub, Slonim et al. 1999). The first was used for the development of gene sequence similarity functions, and the remaining two were used as confirmatory sets. The data sets contained expression levels for 6200, 6817, and 7129 genes, respectively. The first two sets quantified gene expression across 60 tumor-derived cell lines and the third across 72 primary leukemias. We determined the upstream regions of the genes as follows: 1) We obtained high quality curated transcripts by finding the equivalent RefSeq NM accession via Unigene and verified the accuracy of the result by a Blast sequence comparison between the two accessions (Altschul, Madden et al. 1997; Pruitt and Maglott 2001). For the Ross data set, we kept a gene only when both 5' and 3' accessions agreed. 2) We mapped the transcript sequences of curated genes onto the August 2001 freeze of the human genome (Haussler 2001) using custom software and kept only those sequences that included the start codon (transcriptional starts were variably annotated, so we chose to use translational starts to provide a uniform coordinate system). 3) We retrieved the upstream regions that contained at least 80% valid sequence. After these quality control filters and elimination of duplicates, this yielded a set of 2592 genes for the Ross data set, 3192 genes for the Staunton set, and 3196 genes for the Golub data set. Using Pearson's correlation, we then constructed two gene pair sets for each data set, one containing gene pairs whose expression was positively correlated, and the other containing gene pairs whose expression was uncorrelated. For the Ross set, we used a cutoff of 0.4 to define a set of approximately 12,000 positively correlated gene pairs. For the Staunton and Golub sets, which had a much larger number of correlated gene pairs, we used a cutoff of 0.5 to

UCSF LIBRARY

define sets of 44,000 and 34,000 gene pairs, respectively. (Cutoffs were chosen to yield between 10,000 and 50,000 gene pairs. Precise choice of cutoffs did not affect the results presented.) In each case, we created a control set of uncorrelated gene pairs by randomly selecting gene pairs with absolute Pearson's correlation less than 0.2, to make an equivalently sized set to the respective positive pairs set. In what follows, we consider the ability of functions of gene pair promoter sequence similarity to yield higher scores on the positively correlated pair sets than the corresponding control sets. Separation of the distributions of scores on positive and negative pair sets was done using ROC area, with p values computed by permutation analysis. In the permutation analysis, the computed scores for a given function were fixed, with the set memberships of the gene pairs being permuted 1,000 times to produce the null distribution of ROC areas.

With respect to the correlated and uncorrelated gene pair sets, understanding the calculation of expression correlation is important. We are not considering *absolute* gene expression. We are considering *variation* in gene expression from sample to sample. Two genes whose expression is high, but is constant (modulo some noise) across a data set, will have low correlation. For high correlation to be computed, two things must be true. First, the genes' expression must vary enough that differences from sample to sample exceed the noise of measurement. Second, the variation of both genes' expression must coincide. That is, if gene A has high expression in a number of cell lines, but low in others, gene B's expression must quantitatively match in order for the AB pair to be part of the positive pairs set. If either the variation is too low or the variation is not coincident, the gene pair will not be in the positive pairs set. The drivers behind gene expression variation between the cell lines and between the leukemia samples include: genomic

UCSF LIBRARY

rearrangements, promoter methylation, mutations, LOH, and a host of other changes. These result in reproducible differences in gene expression that are substantial from sample to sample. Our results were not sensitive to either the precise threshold on correlation or to the correlation statistic (e.g. Kendall's Tau, a non-parametric rank statistic, yields similar results to those reported below).

We explored a variety of scoring functions of pairwise promoter sequence similarity, beginning with straightforward counting of matching n -mers (a sequence of n bases) in sequence pairs and ending with a function that focuses on long stretches of concordant n -mer matches. Figure 9 illustrates two functions (denoted N and F). Function N is essentially a straight count of dots in a dotplot. Function F eliminates matches from N that occur in isolation, thus concentrating on concordant regions of n -mer similarity. These functions are efficiently computed based on explicit indexing (see <http://jainlab.ucsf.edu> for additional details, code, and data). On the Ross data set, counting matching n -mers (function N) yielded a significant degree of separation between the distribution of scores computed on positively correlated gene pairs and the distribution of scores computed on uncorrelated gene pairs, with a maximum separation using 1000 upstream bases with an n -mer size of 7 (ROC AUC 0.54, $p < 0.01$, by permutation analysis). To a degree, this paralleled results seen in yeast, where the first 1000 bp upstream of the ATG were shown to contain several n -mers implicated in cell cycle regulation (Wolfsberg, Gabrielian et al. 1999). However, additional upstream context diluted the signal (the number of expected randomly matching n -mers increases quadratically with upstream window size).

UCSF LIBRARY

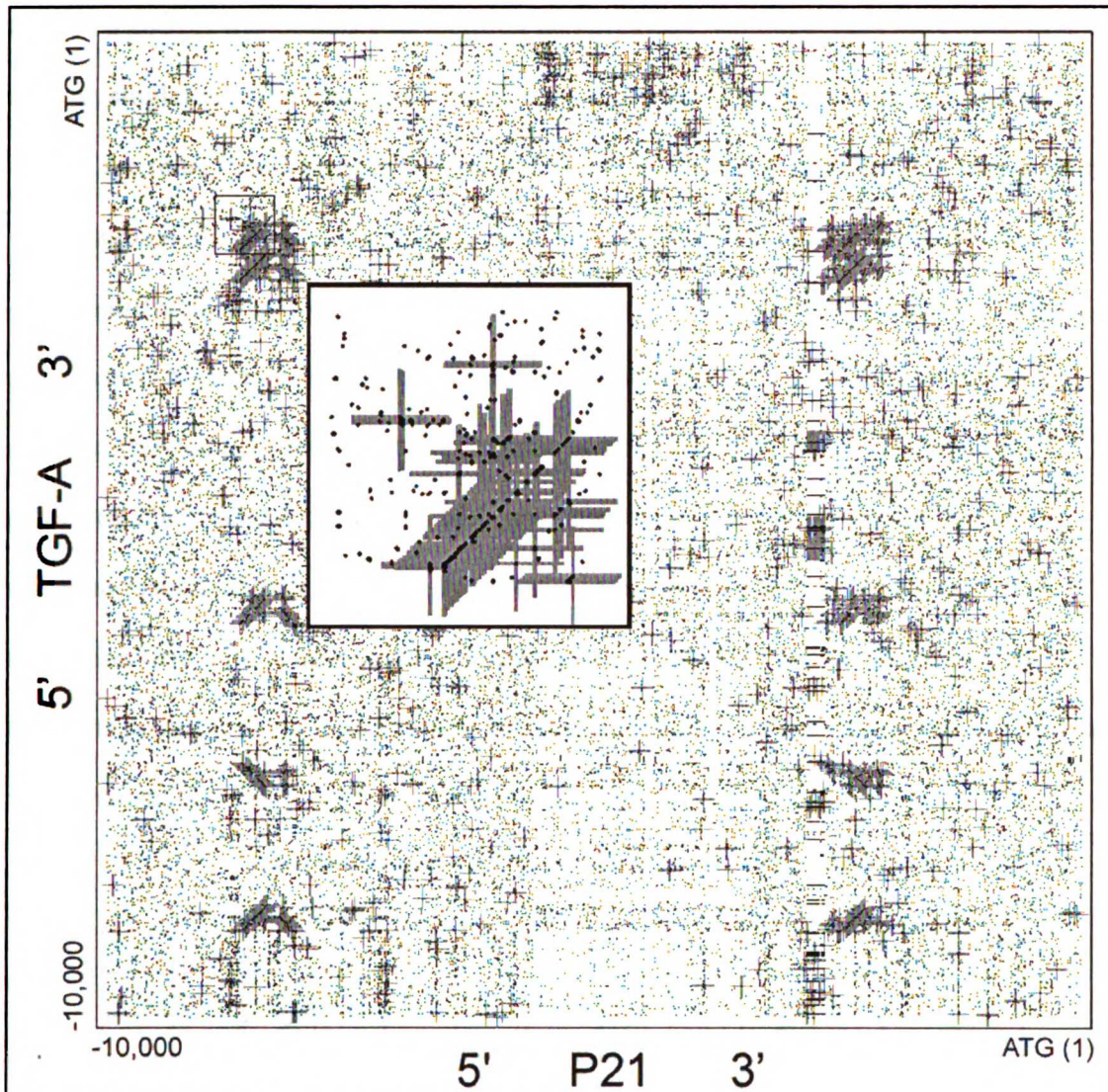


Figure 9. Illustration of sequence comparison functions.

TGF α is plotted on the Y axis, from 10,000 bp upstream to the ATG, and *p21* is plotted on the X axis, from 10,000 bp upstream to the ATG. The inset represents approximately 500 bp. Exact 6-mer matches are indicated with small black dots. Function N is simply the count of all such dots, yielding an integer given the sequences of two promoter regions. Function F counts *concordant* matches. That is, within a particular window size (default 200), if a threshold of 6-mer matches (default 4) is met, where the 6-mers have the same relative offset between sequences, a match is counted for F. These matches are indicated by gray plus signs. The non-concordant matches, which are the majority, do not contribute to F. F will find arbitrarily ordered local regions of high sequence similarity in two promoter sequences. The inset shows a stretch of over 200 bp where a large number of concordant matches exist. Note: for both functions, matches for the reverse complement of the second sequence against the first sequence are included in the score. In the plots, the reverse complement matches are plotted by their position on the sense strand.

Several transcription factors in humans have been shown to act specifically on DNA much further upstream (White 2001). Inspection of dot plots between large upstream regions of gene pairs known to be transcriptionally governed by the same transcription factor revealed significant regions of sequence similarity. Function F

(Figure 9 above) was designed to focus on these regions while reducing the effects of random singleton n -mer matches. Specifically, function F counts a set of n -mer matches if there are more than a threshold of t matches that fall within a specified window w along each diagonal. So, given a number of regions of local sequence similarity, possibly ordered differently in two promoter regions, function F will score the local matching sequences. This function yielded a significant separation in the Ross data set of positive and uncorrelated pairs with upstream regions as large as 10,000 bases. The optimal parameter choices were n -mer size of 6, w of 200, and t of 4, but the separation was robust to many parameter choices. With function F , we were able to see a better separation than with N using a much larger upstream window, and the *number* of match elements contributing to the score was much smaller and more focused. However, despite being statistically significant, the effects were small (ROC area of 0.52).

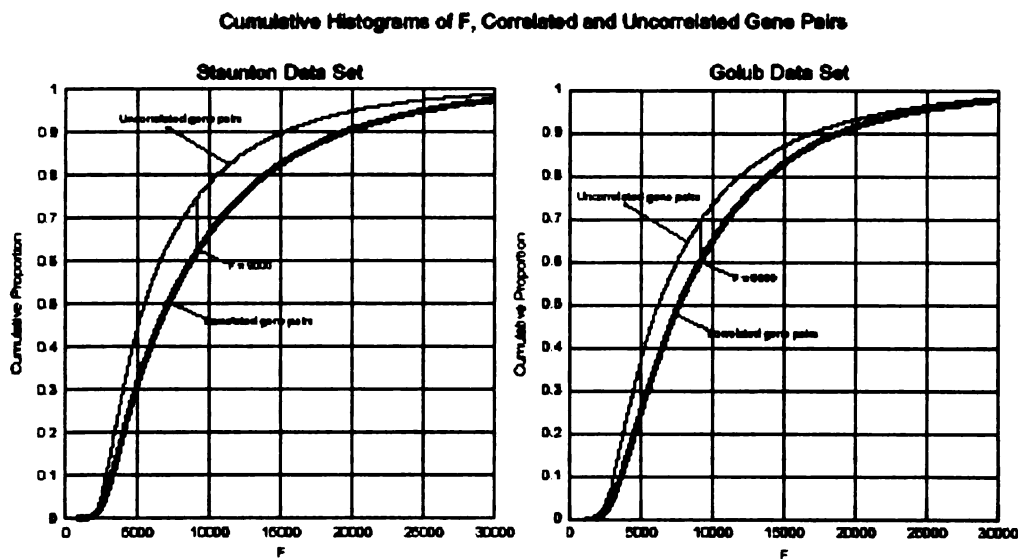


Figure 10. Separation using function F on the Staunton and Golub data sets. The thick black curves are the cumulative proportion (ordinate) of gene pairs with correlated gene expression (Pearson correlation > 0.5) with F less than the value on the abscissa. The thin black curves are the respective plot for uncorrelated gene pairs (Pearson correlation between -0.2 and 0.2). The difference between these distributions is highly statistically significant ($p \ll 0.01$). The dotted lines indicate $F = 9000$ in both plots (see text).

Given the subtlety of the effects on the Ross data set, we applied F to the Staunton and Golub sets, which had much larger numbers of correlated gene pairs. For both the Staunton and Golub sets, the ROC area was substantially higher (0.596 and 0.579, respectively) and was highly statistically significant ($p \ll 0.01$, by permutation analysis). The much stronger results on the Staunton set over the Ross set, which used expression data from the same cell lines, were likely due to more robust expression quantification using the more mature oligo-based platform in the former case. Figure 10 shows the cumulative histograms of F for the correlated and non-correlated gene pairs for the Staunton and Golub sets. For the Golub set, 30% more gene pairs had F higher than 9000 in the correlated versus uncorrelated pair sets. For the Staunton set, the increase was 50%. The converse experiment (pair sets defined by F and separation assessed using Pearson's correlation) exhibits nearly as strong a signal in both cases (data not shown). Thus gene pairs with high measured correlation had high F and gene pairs with high F had high measured correlation.

Table 3. Performance of F (ROC area) under various conditions.
Parameters are Upstream 10,000, Nmer = 6, window = 200, Threshold = 4

Condition	Staunton data set	Golub data set
Unmasked sequence	0.596	0.579
Repeat-masked	0.451	0.500
Repeats only	0.594	0.563
<i>Alu</i> -masked	0.579	0.511
<i>Alu</i> elements only	0.555	0.556

Given the significant relationship between F of two genes' upstream sequences and the correlation of expression of the two genes, we explored sources for the signal. We established that some of the highly concordant regions we observed were due to the presence of *Alu* repetitive elements (Schmid 1996). We re-ran F on upstream sequences from the Staunton and Golub sets modified by masking repetitive sequence elements or

UCST LIBRARY

masking everything *but* repetitive sequence elements (see Table 3). Masking all repeat sequences eliminated *all* separation. Masking the inverse (i.e. considering only repetitive sequences) retained nearly all separation. We further considered only *Alu* elements. Considering only *Alu* repetitive elements decreased the score from considering all repeats slightly in the Golub case and markedly in the Staunton case. Masking all *Alu* elements eliminated nearly all separation in the Golub case and did so slightly in the Staunton case. It is not clear why the full unmasked upstream sequence performed better than just the repetitive elements, given that masking them eliminated all signal. Assuming independence between repetitive and non-repetitive sequences, the effects of masking should be additive. We have not quantified dependence between the content of repetitive and non-repetitive sequences.

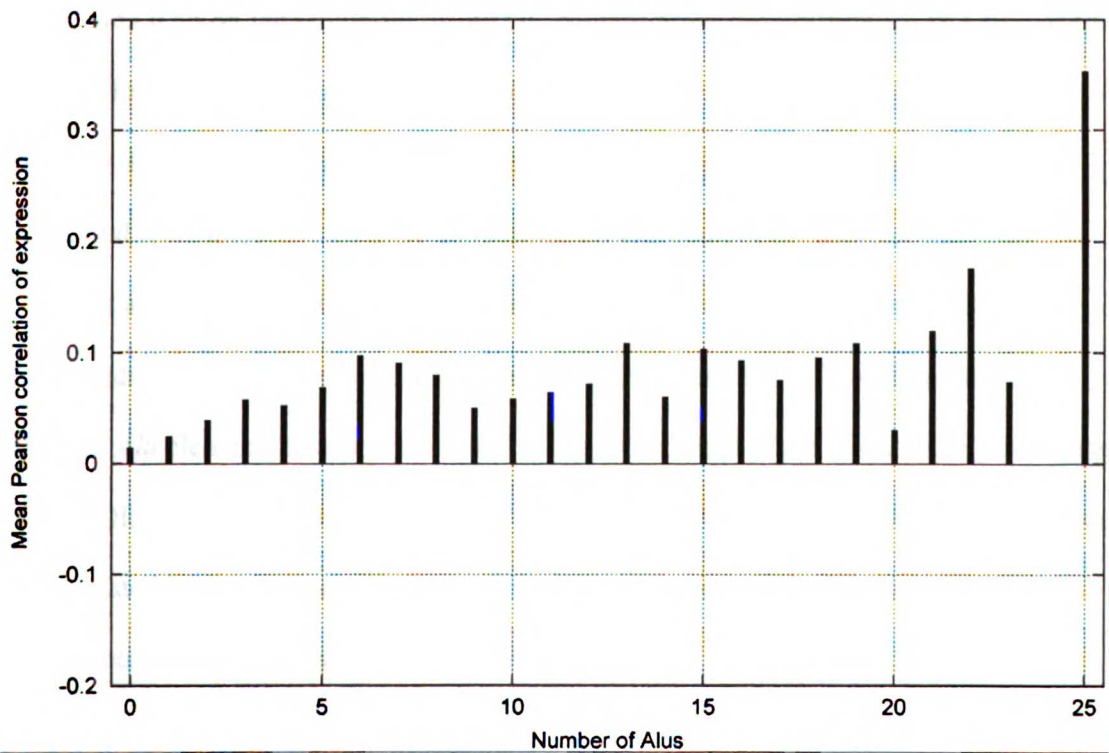


Figure 11. Expression Correlation in Gene Pairs with Equivalent Alu Count

UCSF LIBRARY

To quantify the relationship of the number of Alu elements to expression correlation, we plotted the mean Pearson's correlation of gene pairs with identical numbers of Alu elements in Figure 11. Since the presence of Alu elements was a predictor of expression correlation, we expected that gene pairs with more Alu elements might have higher expression correlation. This trend seemed to hold generally, but since the sample size of the gene pairs with more Alu elements tended to be smaller, it was not possible to make a strong statistical statement.

We considered whether the effects of GC composition were related to F , since both gene content and gene expression variation have been shown to be related to isochore content (Bernardi 1995; Pesole, Bernardi et al. 1999). We confirmed the previous studies in that gene pairs with correlated expression were partially separable from uncorrelated gene pairs based on the GC content of the 10Kb upstream regions (ROC area 0.56 on the Golub set, using GC content proportion difference as a similarity score). *However, GC content difference is not significantly correlated with F .* Also, masking *Alu* elements or all repetitive elements had *no effect* on the separation of correlated from uncorrelated gene pairs based on GC content differences. Interestingly, Oliver et al. have shown that the insertion density of younger *Alu* elements within the human genome is not correlated with isochore GC content nor is the GC content of younger *Alu* elements themselves related to the GC content of the isochores in which they reside (Oliver, Carpena et al. 2002). The signal driving the separation of correlated versus uncorrelated gene pairs using function F was *independent* of GC content differences in the promoter regions considered.

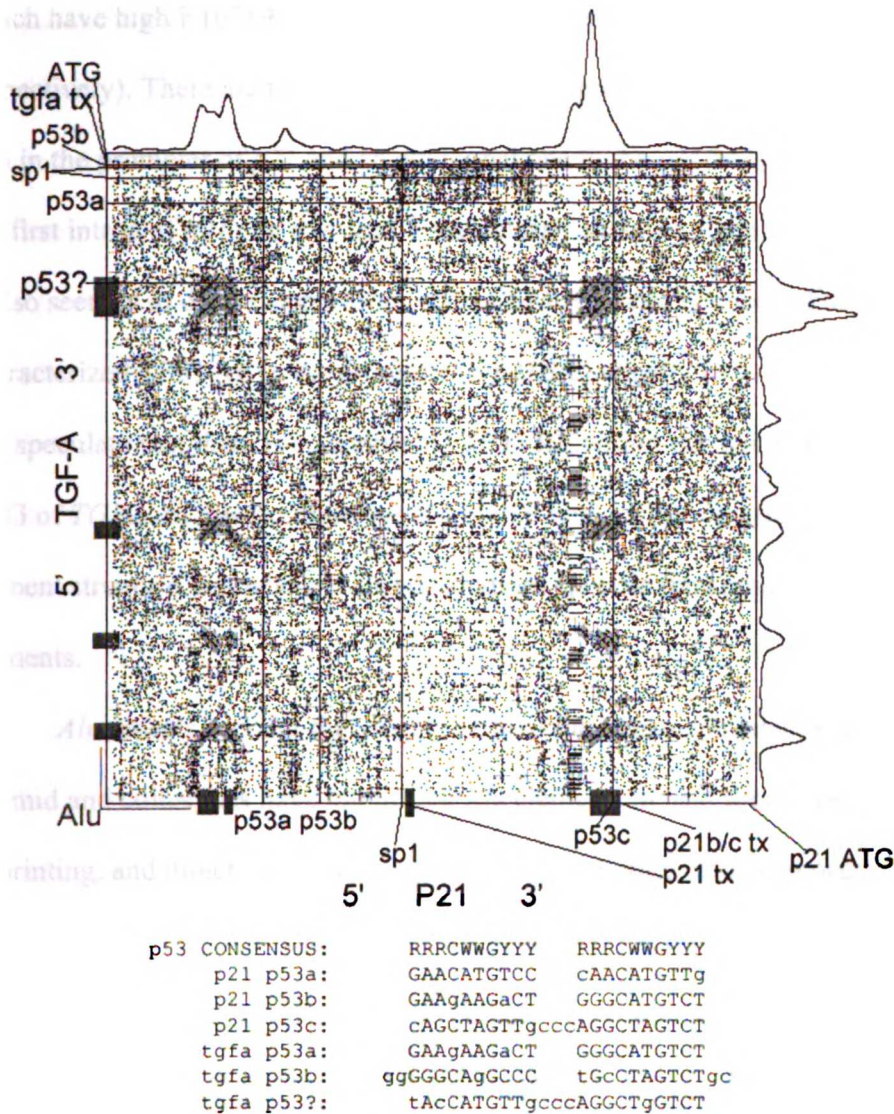


Figure 12. Annotated comparison of p21 and TGFA.

TGFA and p21 are plotted along the same axes and over the same extent as in Figure 9. The contribution of function N is indicated with black dots, and function F is indicated with orange plus signs. Along the top and right of the graph are smoothed histograms indicating the position-specific count of F (peaks indicated sequence regions that contribute disproportionately to F). The positions of Alu elements are marked with red rectangles (5 in TGFA and 4 in p21). The transcriptional start sites for TGFA, p21, and p21B/C are annotated with a "tx". The first exon of p21 is indicated with a green rectangle. The established TP53 (p53a & p53b) and SP1 response elements are indicated by lines. A recently established TP53 RE (p53c) falls within an Alu inside the first intron of p21, which corresponds to the promoter region for p21B and p21C. The position of a putative TP53 RE within an Alu of TGFA is marked "p53?". It has two perfect 11-mer matches to the sequence of the corresponding TP53 RE in p21 (sequences are shown for all TP53 REs at bottom). The positions of the Alu elements contribute substantially to F. There are additional significant peaks in the contribution to F corresponding to the position between p53a and p53b within p21 and directly in the cluster of SP1 sites of p21 and TGFA. The area just 3' to the first exon of p21 may contain additional SP1 binding sites.

Figure 12 illustrates F on the upstream regions of two TP53-responsive genes, *p21* (official gene name *CDKN1A*, also known as *WAF1* and *CIP1*) and *TGFA* (transforming growth factor alpha) (Shin, Paterson et al. 1995; Nozell and Chen 2002),

which have high F (6789) given their moderately low *Alu* counts (four and five, respectively). There are two established TP53 response elements (REs) in *TGFA*, with two in the promoter of *p21* and an additional one in the *p21B/C* promoter (which lies in the first intron of *p21*). In this case, F was clearly deriving signal from *Alu* elements, but it also seemed to derive signal from known SP1 response elements. Further, the recently characterized TP53 RE in the promoter of *p21B/C* actually falls *within* an *Alu* element. We speculate that there is an additional TP53 RE within the *Alu* most proximal to the ATG of *TGFA*. In this example, we see high F with a clear contribution from repetitive element structure, but where F also depends on specific transcription factor response elements.

Alu repeats account for roughly 10% of the human genome (Schmid 1998).

Schmid and colleagues have implicated *Alu* elements in heat shock response, genomic imprinting, and direct interaction with PKR, among other cellular processes (Schmid and Jelinek 1982; Hellmann-Blumberg, Hintz et al. 1993; Schmid 1996; Chu, Ballard et al. 1998; Schmid 1998; Kim, Rubin et al. 2001). Indeed, they have even demonstrated a direct role of TP53 in repressing Pol III-directed *Alu* expression (Chesnokov, Chu et al. 1996). Our realization that the recently described TP53 RE in *p21B/C* is in fact within an *Alu* element supports the proposition that TP53 and *Alu* elements functionally interact. Others have shown specific effects of *Alu* elements on transcription of genes such as *ZNF177* (Deininger and Batzer 1999) and *BRCA1* (Sobczak and Krzyzosiak 2002), and have proposed that expression of many genes may be influenced by *Alu* elements. Our results with F on very large sequence set pairs support a broad role for *Alu* elements in particular, and repetitive elements in general, in influencing transcriptional regulation.

There are several potential explanations for this phenomenon. A statistical artifact is unlikely, due to the very low p values obtained coupled to the fact that F was developed on a different data set than the ones on which it was extensively tested. Another possibility is that this is an evolutionary artifact: somehow genes that tend to be co-expressed happen to contain similar repetitive element compositions, although a plausible mechanism for this is not immediately evident. The most interesting possibilities involve some functional role for the composition of these regions in the transcriptional regulation of the specific genes in whose promoter regions they reside. Since it appears that many gene pairs appear to have a component of their transcriptional regulation tied to the presence of *Alu* and other repetitive elements, influence on the structure of DNA, its packaging, or modification by methylation may offer potential mechanisms. Given that *Alu* repeats are known to make frequent jumps (approximately once every two-hundred live births) in the human genome (Deininger and Batzer 1999), our results indicating that these jumps may have effects on gene transcription support the suggestion that the general phenomenon may be a fundamental evolutionary mechanism (Hamdi, Nishio et al. 2000). At minimum, computational methods designed to predict gene transcription based on sequence in humans should begin to consider the possible influence of these repetitive regions, rather than explicitly ignoring them.

4.4. Materials and Methods

4.4.1. *Expression Data and Upstream Sequence*

The Ross, Staunton, and Golub data sets contained expression levels for 6200, 6817, and 7129 genes, across 60 tumor-derived cell lines (first two) and 72 primary

leukemias (third). We determined the upstream regions of the genes as follows: 1) We obtained high quality curated transcripts by finding the equivalent RefSeq NM accession via Unigene and verifying the accuracy of the result by a Blast sequence comparison between the two accessions (Altschul, Madden et al. 1997; Pruitt and Maglott 2001). For the Ross data set, we kept the NM accession only when both 5' and 3' accessions agreed. 2) We mapped the transcript sequences of curated genes onto the August 2001 freeze of the human genome (Haussler 2001) using custom software and kept only those sequences that included the start codon. 3) We retrieved the upstream regions that contained at least 80% valid sequence. After these quality control filters and elimination of duplicates, this yielded a set of 2592 genes for the Ross data set, 3196 genes for the Golub data set, and 3192 genes for the Staunton set. Pearson's correlation was used to define correlated and uncorrelated gene pairs. After these quality control filters and elimination of duplicates, this yielded a set of 2592 genes for the Ross data set, 3192 genes for the Staunton set, and 3196 genes for the Golub data set. Using Pearson's correlation, we then constructed two gene pair sets for each data set, one containing gene pairs whose expression was positively correlated, and the other containing gene pairs whose expression was uncorrelated. For the Ross set, we used a cutoff of 0.4 to define a set of approximately 12,000 positively correlated gene pairs. For the Staunton and Golub sets, which had a much larger number of correlated gene pairs, we used a cutoff of 0.5 to define sets of 44,000 and 34,000 gene pairs, respectively. (Cutoffs were chosen to yield between 10,000 and 50,000 gene pairs. Precise choice of cutoffs did not affect the results presented.) In each case, we created a control set of uncorrelated gene pairs by randomly

UCST LIBRARY

selecting gene pairs with absolute Pearson's correlation less than 0.2, to make an equivalently sized set to the respective positive pairs set.

4.4.2. *Similarity Metrics*

The algorithms share the use of a sequence index to speed execution, introduced in Section 1.5. This index returns all the locations of a particular n -mer in a sequence in constant time. To create this index, we start with an empty table that contains every combination of n -mers for size n . The sequence is broken up into overlapping n -mers, and the position of each particular n -mer is recorded in the table. The advantage of this approach is that while the index-generating step takes linear time, all subsequent accesses to this index are constant time lookups. The two classes of algorithms we pursue follow below. Note: for comparison of two sequences, matches for the reverse complement of the second sequence against the first sequence are included in the score.

Our straightforward approach to determine the similarity of two sequences is to count the number of different n -mers (for a set length n) that are shared between two sequences (i.e. upstream regions of specified length for two genes). By employing the indexing scheme described above, we can compute this in linear time. By iterating through all combinations of n -mers, we multiply the number of appearances of the n -mer in each sequence against each other; this is equivalent to counting the number of dots in a dot plot (Function N).

An alternate approach depends on the observation that a dot plot of the upstream regions of two co-expressed genes tends to have series of n -mer matches along diagonals. If we count only these concordant matches, we may reduce the noise resulting from spurious matches. Function F counts a set of n -mer matches if there are more than a

UCST LIBRARY

threshold of t matches that fall within a specified window w along each diagonal. To improve speed and space efficiency we convert the dot plot (a sparse matrix) into a data structure that records only the n -mer matches for each diagonal in sorted lists. The optimal parameter choices (derived on the Ross data set) were n -mer size of 6, concordance window size of 200, and match threshold of 4.

4.4.3. *Repeat Masked Sequences*

For repeat masked sequences, we used a repeat masked genome downloaded from UCSC's web site, which uses RepeatMasker to erase repeats (Smit, A.F.A. and Green, P. RepeatMasker, <http://ftp.genome.washington.edu/RM/Repeatmasker.html>). To derive *Alu* masked and *Alu* only sequences, we Blast searched for *Alu* sequences using the consensus *Alu* sequences from the major sub-families in Repbase to calculate the start and end points of the *Alu* sequences, and then masked or extracted the corresponding *Alu* sequences accordingly (Jurka 2000).

4.5. Conclusion

This chapter showed that we could use a simple and fast similarity metric utilizing the indexing techniques introduced in Section 1.5 and Chapter 3 and make interesting biological observations even in a higher organism like human (see Section 1.2). In particular, we showed that there is a quantitative relationship between coexpressed pairs of genes and the amount of repetitive element structure found within these genes. Specifically, *Alu* elements, which comprise 10% of the human genome, was contributing a majority of this signal, and as such repetitive elements may play an important part in human transcriptional regulation.

The next chapter looks deeper into transcriptional regulation by focusing on specific binding sites found in promoters. We present MaMF, a motif finder targeted at mammalian organisms that uses indexing extensively.

UCSF LIBRARY

Chapter 5

MaMF: A Deterministic Motif Finding Algorithm with Application to the Human Genome

5.1. Abstract

We present a novel algorithm, MaMF, for identifying transcription factor (TF) binding site motifs. The method is deterministic and depends on an indexing technique to optimize the search process. On common yeast data sets, MaMF performs competitively with other methods. We also present results on a challenging group of eight sets of human genes known to be responsive to a diverse group of TFs. In every case, MaMF finds the annotated motif among the top scoring putative motifs, performing better than other motif finders. We analyzed the remaining high scoring motifs and show that many correspond to other TFs that are known to co-occur with the annotated TF motifs. The significant and frequent presence of co-occurring transcription factor binding sites explains in part the difficulty of human motif finding. MaMF is a very fast algorithm, suitable for application to large numbers of interesting gene sets.

UCSF LIBRARY

5.2. Introduction

Motif finding is one of the core challenges in using bioinformatics to understand transcriptional regulation. By creating a model of the binding site to which a TF is predicted to bind, several questions can be addressed: 1) What is different about the binding site compared with the contextual sequence of the promoter region? 2) What is the nature of the physical interaction between the binding site and its TF? 3) What is the relationship between coexpressed genes that are regulated by the same TF? Because of the centrality of this problem, we use motif finding to address the themes of this thesis, including understanding transcriptional regulation in the context of higher organisms (Sections 1.2 and 1.3), building models of the TF binding site (Section 1.4), and using indexing as a programming paradigm (Section 1.5).

With respect to the first theme, understanding transcriptional regulation in the context of higher organisms, a recent survey by Tompa et al. (2005) comparing currently available motif finding tools has demonstrated the difficulty of addressing the human case and developing methodologies for motif prediction assessment in light of the complexities of mammalian transcriptional machinery. In this survey, the authors presented a methodology in which an algorithm predicts a single motif and is measured by its ability to identify the positions of the annotated binding sites. They found that all 13 algorithms were shown to perform significantly worse on human gene sets than on yeast gene sets. From this assessment, two of their key observations helped motivate this paper: 1) The results suggest that efforts in modeling binding sites in yeast have been more successful than in metazoans, which further suggest that there is opportunity in developing a motif finder that is targeted at higher organisms; and 2) The assessment

UCSF LIBRARY

methodology should consider the top N predicted motifs to increase sensitivity and account for binding sites for multiple transcription factors that may work in concert with the annotated one. To that end, we present a motif finder targeted at higher organisms, and then assess this algorithm using methodologies that address motif assessment challenges in the Tompa et al paper that are inherent in motif finding in higher organisms.

It turns out that the motif prediction problem has a parallel to the problem of predicting the relative conformation and alignment of small-molecule ligands to a protein binding site of unknown structure. The analogy lies in the equivalence between a set of promoters and a set of small molecules: both are ligands of a particular protein (the former are just quite a lot larger). In the former case, we do know neither the relative alignment of the promoters nor the width of the specific footprint of binding. In the latter case, we know neither the relative alignment of bound ligands nor their specific conformations. Our early work in the small-molecule case established a viable approach (Dietterich, Jain et al. 1994; Jain, Dietterich et al. 1994), and our more recent work has yielded practically useful methods for solving the ligand superposition problem in a manner that yields models capable of predicting new ligands (Jain 2000; Jain 2004). The key difference between the two domains, of course, is that we can model the small molecule ligands at atomic scale but must model promoters based on DNA sequence. In both cases, we have a problem of hidden variables (conformation and 3D alignment versus binding footprint and 1D alignment). In neither case is there an *ab initio* solution, since we lack structural information on the binding site of the proteins in question. Hence, heuristic approaches are required. In both cases, our approach is to couple a

UCST LIBRARY

scoring function of empirical design to a fast, deterministic search strategy in order to yield a ranked list of likely solutions.

We present MaMF (Mammalian Motif Finder), whose development was premised on the hypothesis that effective search of the space of possible motifs, coupled with a scoring function that combines local similarity effectively with genomic background information, would yield practically useful results in the metazoan case of motif discovery. The search process is accelerated through the use of an index of the input sequences, which allows MaMF to generate large numbers of aligned motifs quickly, maximizing its search depth without significantly increasing its running time. The fast search procedure is coupled to a very simple scoring function that combines a preference for conservation among input sequences with a preference for under-represented sequences relative to the genome.

We assessed MaMF by answering two questions. The first is a comparative one about algorithm performance relative to other widely used methods, and the second is about the biological significance of high-scoring motifs that are unannotated and may represent false positives. The comparison question addressed both lower and higher organisms. Our baseline results using standard motif assessment techniques demonstrated that MaMF performance on data from yeast and *e. coli* was comparable to other widely used algorithms. To allow systematic analysis of the harder human case, we developed an algorithm assessment method that uses a motif similarity metric to assess the top N motifs predicted by a motif finder (30 in the data presented), in contrast to the method presented by Tompa et al. (but in accordance with their suggestion). We present results on an unbiased set of human promoters, whose TF interactions have been annotated in

UCST LIBRARY

TRANSFAC, and demonstrate that MaMF is able to identify the correct annotated human motifs and is not very sensitive to specific parameter choices. We then use a larger independent benchmark data set obtained from Tompa et al. and show that MaMF performs better than other motif finders.

The second question seeks to determine the biological significance of the high scoring but unannotated motifs. While these motifs are nominal false positives, the transcriptional complexity of mammalian systems suggests that they may in fact be motifs that bind onto other TFs that work in concert with the annotated TF. To test this, we introduce an assessment technique that employs expression microarray data to determine the degree to which a putative motif is found to be enriched among coexpressed genes relative to non-coexpressed genes (the enrichment ratio), which can then be used to develop biological support for high scoring but putative false positive motifs. Using this technique on the TRANSFAC data sets, we show that microarray expression data can be used to support and rank hypothesized motifs, and that many putative false positive motifs predicted by MaMF are probably due to the presence of frequently co-occurring bona fide TF motifs with the annotated motifs. The result is interesting in that it underscores the ubiquity of multiple-TF regulation of gene expression in human biology, but it also offers a way to make use of easily obtained high-throughput biological data to help triage the results of motif-finding exercises.

5.3. MaMF Algorithm Summary

Given a set of N promoters and an input motif width w , MaMF seeks to maximize the value of a scoring function that prefers motifs that are conserved across the different promoters and are under-represented in the target genome (see Figure 1). MaMF yields a

UCST LIBRARY

ranked list of motifs, where each motif contains exactly N sequences of length w (zero or more sites for each input promoter). We will first describe the search algorithm, then the scoring function.

MaMF's search algorithm is deterministic, and it depends on a simple yet effective indexing strategy to optimize performance. Indexing techniques to speed searches have been used widely, most notably in the BLAST algorithm (Altschul, Madden et al. 1997). In the case of MaMF, we can create an index of all n mers (defined as a short sequence of length n , typically 4-6 bp long) found per input sequence, which makes identifying locations within a sequence that have a given n mer a constant time operation. Given indices of two sequences and an n mer, we can identify all alignments between the two sequences that share that particular n mer in constant time. Using this strategy, therefore, we can efficiently generate a lookup table for all sequence pair alignments of width w that share an n mer and meet an identity cutoff t (see Figure 13b). Larger n mer sizes increase speed at the expense of search depth.

Enumerating all sequence pairs from the lookup table and scoring them using the scoring function described below, we keep the top 1000 high scoring sequence pairs to be used as seeds in the motif generation step (see Figure 13c). The motif generation step employs a greedy search strategy that builds motifs from the high scoring sequence alignment seeds, iteratively adding sequences to the growing motif that maximize the motif score (see Figure 13d). At each iteration, we obtain potential sequences that align with any of the sequences already part of the motif via the lookup table, caching those sequences that do not maximize the motif score for reconsideration in subsequent iterations. The motif is complete when it reaches a size threshold equal to N , the number

UCST LIBRARY

of input sequences. The 1000 motifs generated are resorted according to their motif score. To make the results easily viewed, the 1000 motifs are filtered to remove highly similar motifs at the 75% similarity level (see Motif Similarity below), and the top 30 remaining motifs are presented.

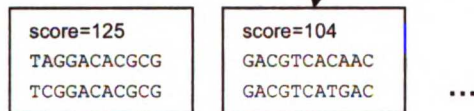
(a) Input Sequences

promoter sequences for POLB, FOS, VIP, ATF2, ADRB2, TGFB2, CCNA2, RPL10, CCND1

(b) Create Lookup Tables Using Indexing Techniques



(c) Generate High Scoring Seeds



(d) Build Motifs Greedily

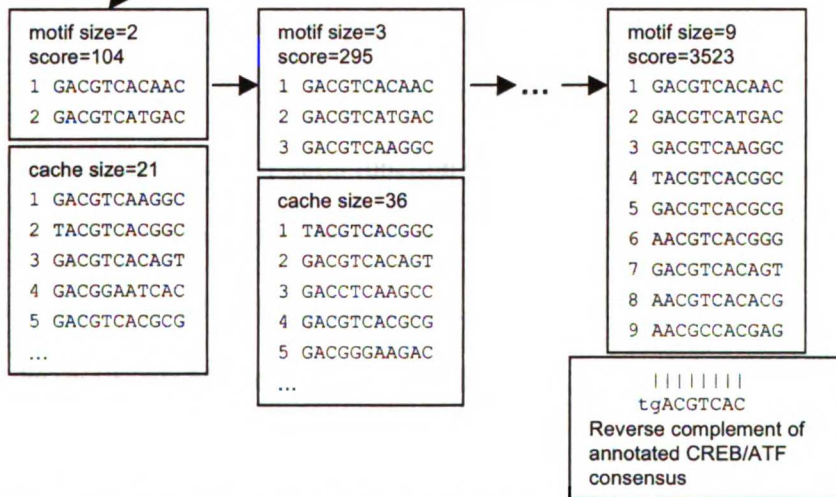


Figure 13. MaMF algorithm walkthrough of the CREB/ATF data set.

The construction of one motif is traced via the arrows from beginning to end, with other examples of intermediate steps shown alongside for comparison.

- a) MaMF starts with a set of input sequences, in this case promoters of N=9 genes from -1000 to +200 of the transcription start site that are responsive to CREB/ATF.
- b) After using the indexing techniques described in the text to index the input sequences, MaMF can quickly create a lookup table for a given sequence of width w that is present in the input

UCST LIBRARY

sequences (boldfaced text in header, $w=11$ in this example), containing all other sequences of width w that share at least one nmer ($n=4$ in this example) and pass an identity cutoff ($t=8$ in this example). Two examples are shown, with matching nmers underlined, and potentially more than one nmer appearing for a given matching sequence.

- c) The seeds are high scoring pairs of sequences used to grow the motifs, generated by enumerating all sequence pairs using the lookup tables, of which two are shown. Of note, the seed with lower score produces one of the highest scoring motifs that corresponds very closely to the annotated CREB/ATF consensus, emphasizing the importance of generating many seeds.
- d) MaMF uses the seeds and the lookup tables to build motifs. In this example, we start with the lower scoring seed in the previous step and retrieve all 21 similar looking sequences from the relevant lookup tables, placing them in the cache. The score of each sequence is the score of the hypothetical motif generated by adding that sequence to the motif. Sorting by this score, the highest scoring sequence (in cache position 1) is added to the motif at position 3. After adding sequences to the cache that are similar to the newly added sequence, this cycle is repeated until the motif size equals the number of input sequence. This approach is greedy because it attempts to maximize the score of the growing motif at every step. The motif shown closely matches the CREB/ATF consensus.

5.3.1. Scoring Function

We define our scoring function to be the following:

$$\text{score} = \left(\sum_{i=1}^w \sum_{j=1}^R \sum_{k=1}^R m(S_{ji}, S_{ki}) \right) \left(-\frac{1}{R} \sum_i \log(p_0(S_i)) \right) \quad (1)$$

where w is the motif width, R is the number of sequences in motif S , $S_1 \dots S_R$ are the individual sequences, $m(S_{ji}, S_{ki})$ is a matching function that returns 1 if sequences S_j and S_k match at position i and 0 otherwise, and $p_0(S_i)$ returns the probability of seeing sequence S_i using a background model. The first term measures motif conservation and is equivalent to the ungapped sum-of-pairs function (Altschul 1989; Gupta, Kececioglu et al. 1995). The second term measures the uniqueness of the motif relative to the relevant genome, obtained by calculating the average background probability of seeing the various sequences in the motif (see Methods for details). The product of the sequence similarity term and the motif frequency term form the scoring function. Motifs that have sequences that are similar to each other and unique relative to the background maximize the score.

UCSF LIBRARY

5.4. Results

We tested MaMF on data from lower organisms as a necessary condition of performance prior to proceeding to the more complicated human case. For both sets of data, we compared performance directly to Weeder (Pavesi, Mauri et al. 2001), Bioprosector (Liu, Brutlag et al. 2001), Consensus (Hertz and Stormo 1999), and AlignAce (Roth, Hughes et al. 1998), all of which are well-established motif finding algorithms. Weeder was shown to perform the most competitively in multiple organisms in the Tompa et al. survey (Tompa, Li et al. 2005). Details of data set preparation, parameter settings for the different algorithms, and methods for evaluation of performance are given in the Materials and Methods section.

5.4.1. Lower Organisms: MaMF Performance





We verified that the MaMF algorithm worked by testing it on common test cases in lower organisms, including *RAP1*, *MCMI*, and *URS1* in yeast, and *CRP* in *e. coli*. These test cases form a common benchmark for motif finding algorithms (Hertz and Stormo 1999; McGuire, Hughes et al. 2000; GuhaThakurta and Stormo 2001; Lieb, Liu et al. 2001; Liu, Brutlag et al. 2001; Liu, Brutlag et al. 2002). The yeast examples contained about 10 genes each, and 1000 bp upstream of each gene was considered. *CRP* contained 33 genes, with 200 bp upstream. MaMF output was defined to be correct if the highest scoring motif matched the reported consensus sequence. In all four cases, the top motif was correct. Table 4 shows the motifs found, the consensus, and gives details of each motif. Performance of the other approaches was comparable, with Bioprosector and Weeder finding the correct motif for all four data sets, and Consensus and AlignAce both

UCSF LIBRARY

finding correct motifs in 3/4 data sets (both failing on *RAP1*), reflecting the fact that these commonly tested cases now form a low bar for evaluation of motif finding algorithms.

Table 4. MaMF run on benchmark yeast and *e. coli* data sets.

For each of the four data sets, the highest scoring motif from MaMF's results (drawn as a sequence logo (Crooks, Hon et al. 2004)) strongly matches the annotated consensus.

TF	Organism	# of Genes	Motif Width	Annotated Consensus	Highest Scoring Motif Found
<i>RAP1</i>	yeast	11	11	CACCCAGACAT	
<i>URS1</i>	yeast	11	11	CGGCGGCTA	
<i>MCM1</i>	yeast	17	11	CCTAAT(A/T)GGG	
<i>CRP</i>	<i>e. coli</i>	33	16	TGTGA(N ₆)TCACA	

5.4.2. Human Data: MaMF Performance

While synthetic data have been used to build larger, more challenging datasets and allow systematic comparisons between algorithms (GuhaThakurta and Stormo 2001; Liu, Brutlag et al. 2002), we favor the use of real biological datasets to measure an algorithm's efficacy. We therefore considered motif finding performance on a set of human example cases. In what follows, we describe two sets of experiments. The first set employs MaMF with unfiltered output, which yields a highly redundant ranked motif list. The second employs a compaction procedure that eliminates redundant motifs, which eases inspection of MaMF output and makes comparison to other algorithms straightforward.

WEST LIBRARY

5.4.2.1. Raw MaMF Output

To minimize the bias of choosing gene sets favorable to motif finding, we exhaustively searched TRANSFAC using strict criteria for human gene sets with sufficient size and verified binding sites, resulting in eight gene sets (see Materials and Methods). Binding site widths for different transcription factors vary, but we chose to search for motifs of width 11 in order to standardize the results. The nmer size was set at 4 and an identity cutoff at 8 ($8/11 = 73\%$ identity). Recall from above that the top scoring motifs matched the correct consensus in the yeast and *e. coli* sets, but of the eight human gene sets, only *E2F* yielded the correct consensus as the top ranked motif. Tompa et al. (2005) noted this issue in their paper, suggesting the use of the top N motifs in algorithm assessment.

To validate whether MaMF was finding the correct motif at all in the remaining gene sets, we assessed the algorithm by counting the number of motifs that matched the correct motif within the top 1000 motifs. We used a motif similarity metric similar to Yona and Levitt (2002), where we define a correct motif to be a putative motif that shares 75% identity with the true consensus motif, considering the best alignment between the two motifs (see Methods). Using this metric, we found that in all eight cases MaMF found at least one motif among the top 1000 motifs that matched the consensus motif, shown in Table 5. In most cases, we found several dozen matching motifs, representing motifs that were very similar to each other, but had a shuffled order of binding sites or a shift in where the core binding site was identified. Generating 1000 motifs of width 11 using ten 1200 bp long sequences took about a minute on a 1.3 GHz computer.

ULST LIBRARY

To assess the probability that the presence of these correct motifs was by chance (given that we considered so many output motifs), we computed the null distribution of correct motifs found by MaMF using 1000 random gene sets. For a given annotated gene set that resulted in some number of correct motifs, the p-value of the result is the percentage of MaMF runs on random gene sets that resulted in more correct motifs than the true gene set. Using this statistical measure, we found that most gene sets returned a number of correct motifs that was statistically significant, shown in Table 5. The remaining gene sets correlated well with the strength of the motif in terms of information content and genome frequency. The *SPI* gene set did poorly because *SPI*'s binding consensus is a GC rich sequence (GGGGCGGGGC) that occurs frequently in the genome. The *ETS* binding motif has a small conserved region with large variability outside of that region, perhaps explaining the weak result in that case.

Table 5. MaMF run on eight TRANSFAC gene sets.

Using a width of 11, a frequency-based background model derived from Refseq gene promoters, nmer size of 4, we generated 1000 motifs. A motif is correct if it shares 75% identity with the TRANSFAC motif using the best possible alignment. The p-value of the result is the percentage of MaMF runs on random gene sets that resulted in more correct motifs than the true gene set. MaMF finds at least one correct motif in all cases, most of which are statistically significant.

Transcription Factor	# Correct Motifs	p value
<i>HNF4A</i>	6	$p = 0.002$
<i>API</i>	15	$p = 0.004$
<i>CREB/ATF</i>	60	$p = 0.002$
<i>E2F</i>	144	$p = 0.001$
<i>SPI</i>	1	$p = 0.78$
<i>ETS</i>	70	$p = 0.26$
<i>CEBP</i>	17	$p = 0.001$
<i>AHR/HIF</i>	115	$p = 0.01$

5.4.2.2. *Filtered MaMF Output*

Since we established that MaMF was finding correct motifs among the 1000 top-ranking motifs, we were interested if MaMF could find the correct motifs in the top 30 results, a more useful metric of measuring success that not only allows direct comparison to other methods but also provides a way for biologists to assess the results by eye.

Having observed that MaMF finds many duplicate motifs, we performed an additional step to remove highly similar motifs (using the motif similarity metric at the 0.75 cutoff), which pruned up to 80% of the motifs. Table 6 shows that MaMF is able to find the correct motif in all eight TFs, in most cases within the top 10, using this compaction procedure. In several cases, the top-ranked correct motif only approximately matched the annotated consensus, but in general a slightly lower ranking motif could be found that more closely matched the annotated consensus.

Table 6. Selected motifs found by MaMF that are highly similar to the annotated motif.

In every gene set, MaMF is able to find motifs that look highly similar to the annotated consensus. Underlined sequence is the 11 bp motif reported by MaMF; additional sequence reported is obtained from the surrounding sequence. Capitalized nucleotides are highly conserved (>50% of nucleotides in the motif at that position have that nucleotide).

TF	Motif rank	Motif similarity to annotated motif	Annotated Consensus/ Motif Result
<i>AHR/HIF</i>	4	0.808	taCGTGcgg <u>GtCACGTcCGg</u>
	6	0.875	<u>GCataCGTGCGc</u>
<i>API</i>	8	0.716	sTGActCA <u>gACgGTGGctCc</u>
	25	0.809	<u>nnGACTCagCctG</u>
<i>CEBP</i>	2	0.701	ATTgCacaAtat <u>ttTGCAAgnGT</u>
	6	0.756	<u>atTGCAAAacTT</u>
<i>CREB/ATF</i>	3	0.941	GTGACGTca <u>scCGTGACGTca</u>
<i>E2F</i>	1	0.873	ttCgCGCCAAac <u>tTTgGCGCcAAat</u>
<i>ETS</i>	6	0.692	aCtTCctg <u>CqCtCCGGtGs</u>
	30	0.826	<u>kCctCCTaCTTc</u>
<i>HNF4A</i>	1	0.652	aGntCAAAGrtcg <u>CaGgGCTCAaGsgtgc</u>
	155	0.830	<u>AGGTCAaAGGTnG</u>
<i>SPI</i>	5	0.697	gGGCGGGgc <u>cAcCCGGGcCGggg</u>
	22	0.742	<u>mGtgCGGGGcCggg</u>

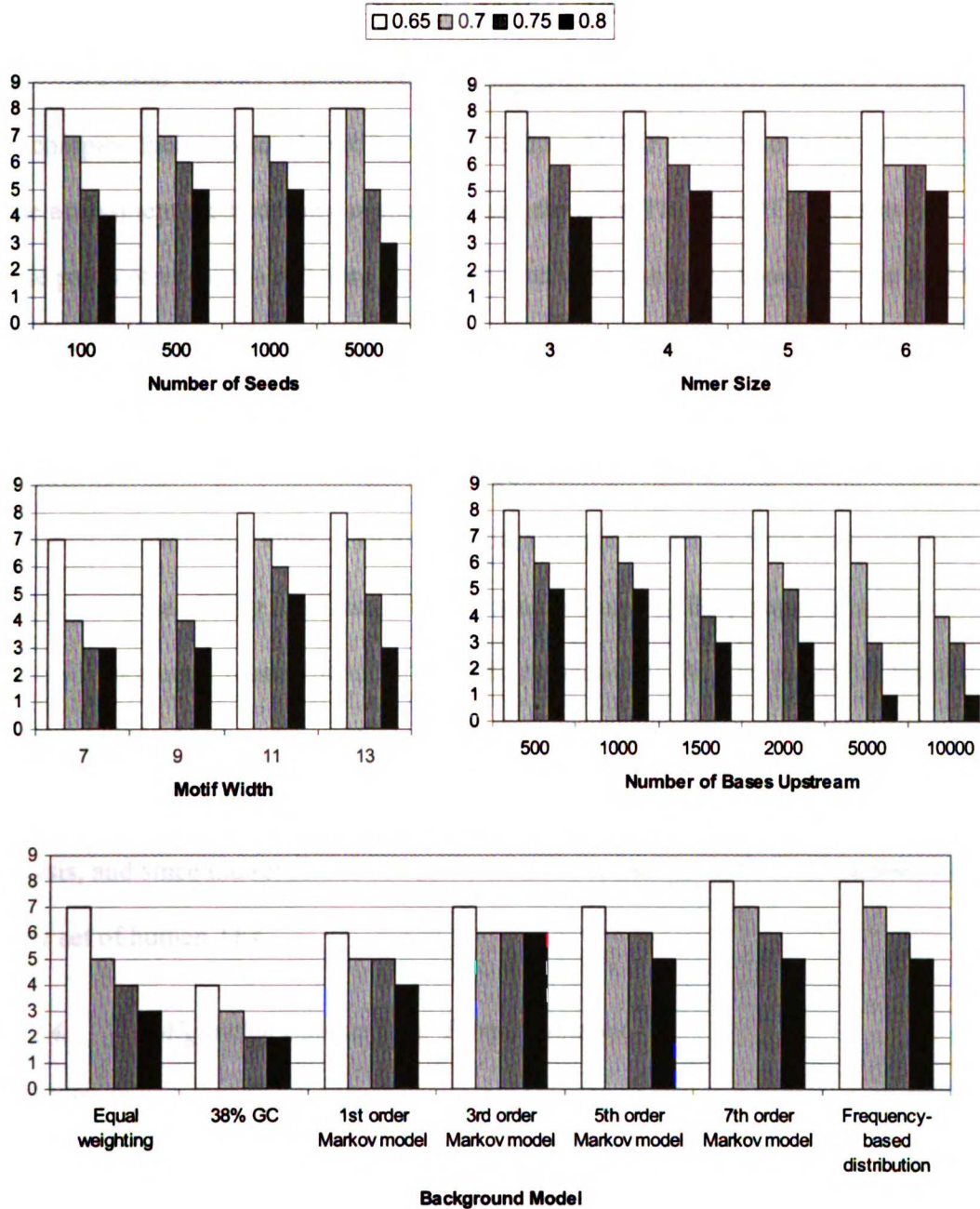
We then tested a variety of parameter settings to evaluate the stability of results. To summarize the results for a given parameter setting, we counted the number of gene sets for which MaMF found a correct motif among the top 30 results, using several motif similarity thresholds to define correctness. Figure 2 shows a series of MaMF runs, varied by the nmer size, motif width, number of seeds, number of bases upstream, and choice of

UWAT LIBRARY

background models. The baseline experiment was based, as above, on an nmer size of 4, a motif width of 11, 1000 seeds, 1000 bases upstream, and the use of the frequency-based background model. Within any given parameter setting, the number of correct gene sets necessarily stays the same or decreases as the motif similarity threshold is increased, reflecting the increasing stringency of categorizing a motif to be correct.

Changing parameters had varying effects on MaMF's performance. MaMF was not very sensitive to the choice of nmer size (Figure 14a) or number of seeds (Figure 14b). With a nmer size of six and 100 seeds, the running time of MaMF can be reduced to less than 10 seconds, suggesting that MaMF can be used in high throughput motif finding studies in human with reasonable sensitivity. A motif width of 11 maximized MaMF's performance (Figure 14c), probably reflecting the average size of the annotated motifs, though other motif widths were also fairly effective. The number of bases upstream used in motif finding greatly affected MaMF's results (Figure 14d). Until about 2000 bases upstream, MaMF was fairly effective in finding motifs, but larger upstream regions made the problem progressively noisier, reducing the number of correct motifs at higher motif similarities. An important determinant of motif finding performance was the choice of genome background models (Figure 14e). The GC weighting scheme performed the worst, and the higher order Markov background models did better than the lower order ones. Interestingly, the equal weighting model (equivalent to having no model) performed better than the GC weighted model. The most sophisticated models utilizing the greatest amount of genomic information, specifically the 7th order Markov background model and the frequency-based background, did the best.

UCL LIBRARY



WEST LIBRARY

Figure 14. Performance of MaMF using various parameter settings.

Unless perturbing that particular parameter, the default parameters are motif width 11, nmer size 4, 1000 seeds, -1000 to +200 bp of the transcription start site, and the count with mutations frequency-based background. a) The number of seeds used did not affect performance. b) Smaller nmer sizes increased sensitivity. c) A motif width of 11 gave the best results, with other motifs giving competitive results. d) MaMF was able to find motifs given up to 2000 bp upstream, and gradually did worse as the amount of upstream was increased. e) The complexity of the background model influenced the MaMF's sensitivity, with the frequency based distribution and higher order Markov models doing the best and the equal weighting (not having a background model at all) doing the worst.

5.4.2.3. *Algorithm Comparison: TRANSFAC Data Set*

We used Weeder, Bioprosector, Consensus, and AlignACE to search for motifs and compare their predicted motifs with those from MaMF (filtered output), using the same assessment methodology as above. Using the eight TRANSFAC gene sets, we chose parameters for the different algorithms that were similar to those used for MaMF but optimized their performance if an identical parameter could not be used (see Methods). Bioprosector performed best when an explicit background model was not provided (background estimated from the input sequences), and we provided results for Bioprosector with and without an explicit background model. In these experiments at the 0.75 motif similarity threshold, MaMF found 6/8 motifs; Bioprosector with and without background found 3/8 and 6/8 motifs, respectively; Weeder found 5/8 motifs; Consensus found 1/8 motifs; and AlignACE found no motifs. Since the performance of MaMF on this set of eight TFs is within its plateau across the parameter sensitivity analysis, and since the set size is small, we made a second test with fixed parameters on a larger set of human TFs.

5.4.2.4. *Algorithm Comparison: Tompa Data Set*

We constructed a non-overlapping data set of 21 human gene sets obtained from the benchmark data in the Tompa et al. paper (see Methods). Using the same metric for success and identical parameters for the algorithms as before, we ran the algorithms on this new data set. Table 7 shows the performance of the different algorithms at the 75% motif similarity threshold, with MaMF finding 12/21 motifs, Weeder finding 8/21 motifs, Bioprosector finding 7/21 (without background) and 4/21 (with background), Consensus finding 1/21, and AlignACE finding 0/21. Under various motif similarity thresholds,

UCL LIBRARY

UWOT LIBRARY

MaMF did consistently best, shown in Figure 15. The rank order of the performance of algorithms generally remained the same compared with the same experiment using the TRANSFAC data set, with Weeder doing better than Bioprosector, but worse than MaMF. The single exception was that Bioprosector used without an explicit background did worse than Weeder in the Tompa data set. Note that we did not evaluate multiple parameter settings for any algorithm on the Tompa data set.

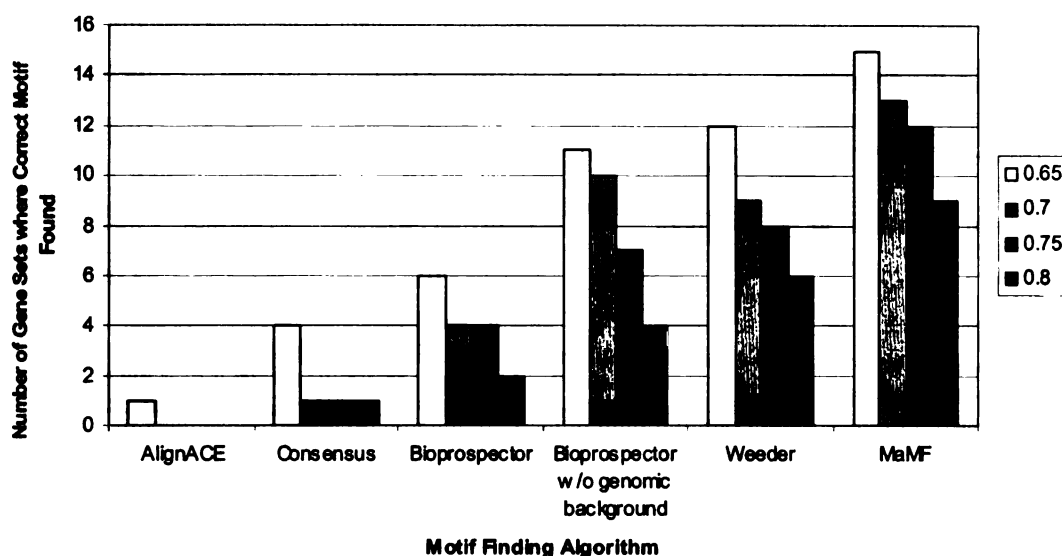


Figure 15. Comparison of motif finding performance between algorithms.

Using 21 human gene sets from the Tompa et al. benchmark, MaMF is the most successful in finding the correct motif for the various motif similarity thresholds shown. A motif is correctly found if the algorithm returns among its top 30 results a motif that matches the annotated motif at the specified motif similarity threshold.

The success rates were *dramatically* higher than levels of correctness reported in the Tompa paper, emphasizing the importance of looking at the top N motifs where N is significantly greater than 1. The results we report in Table 2 can be related to the approach presented in Tompa et al. (2005). At each level of motif similarity, we are defining a different threshold of correctness of a predicted *motif*. Tompa's approach is to consider correctness of predicted *sites* and defines them as correct when a predicted set of nucleotides overlap a true TF binding site by at least one-quarter in length. In our case,

WEST LIBRARY

there are exactly 21 true positives to consider, and each algorithm yielded 30 motifs, most of which constituted nominal false positives. Tompa’s results showed strong correlation among positive predictive value, sensitivity, and the “average site performance.” In our analysis, sensitivity ($TP/(TP+FN)$) is monotonically related to the motif similarity threshold, ranging from a sensitivity of 0.71 down to 0.43 for MaMF. Issues involving quantification of specificity in real data sets are complicated by the likely presence of true functional motifs distinct from the annotated ones, as also noted by the Tompa group, which we discuss below. Nominal sensitivities ($TP/(TP+FP)$) for all methods in our analysis are low, by construction of the assessment, since we considered 30 top-scoring motifs for each promoter set, most of which are false positives by definition if not in reality.

Table 7. Comparison between MaMF, Bioprosector, and Consensus on the Tompa data set.
 The rank of the first motif, if any, that matches the annotated TF motif at the 0.75 motif similarity threshold is reported, looking at the top 30 motifs. AlignACE does not find any correct motifs at this threshold, and is not shown.

	MaMF	Weeder	Bioprosector (without specified background)	Bioprosector (with background)	Consensus
hm02					
hm03			9		
hm04		14			
hm05		7			
hm06	1	1	3	4	
hm07	4				
hm09	1	1	1	8	
hm11					
hm12	13				
hm13					
hm14					
hm15	16				
hm16					
hm17	1	2	2		1
hm18	23			22	
hm21	4	6	8		
hm22	30	1			
hm23	15			26	
hm24	1	6			
hm25	13		15		
hm26			8		

UNIVERSITY OF TORONTO LIBRARY

We believe that MaMF's performance advantage on the larger set of human test cases was not artifactual for three reasons. First, in the case of the motifs for the lower organisms we observed excellent performance for all of the algorithms, reflecting that the algorithms were run properly. Second, the comparison approaches had similar or longer running times (all were similar, with Weeder being approximately 100x slower), so there was no bias toward MaMF with respect to computational burden. Third, the parameters used in the Tompa data set evaluation were fixed without knowledge of that data set. So, with respect to the first question posed in the introduction, we believe that MaMF offers performance advantages over other widely used methods on non-synthetic human data.

5.4.3. *Biological Significance of High Scoring Incorrect Motifs*

Given that only a fraction of the top motifs returned by MaMF were true positives (either for the full 1000 motifs or redundancy-filtered 30), we considered the second question posed in the introduction: what the other motifs might be (referred to below as "incorrect" motifs). There are several possibilities. First, the remaining motifs could really be incorrect and without biological significance, a reflection of a difficult and noise-ridden problem, where the signal from human binding motifs may be too weak to strongly distinguish true positives from false positives. Second, genome-wide effects such as repetitive element distribution or GC composition may encourage composition-based motifs to appear. Indeed we have previously shown a relationship between coregulated genes and the quantity of repetitive elements (Hon and Jain 2003), but in the above experiments, we removed repetitive elements. As for GC composition, it has been previously shown that gene expression variation is related to isochore content (Bernardi 1995; Pesole, Bernardi et al. 1999) which may also yield composition-based motifs. The

UWOF LIBRARY

third possibility is that many of the remaining motifs are biologically active, either as targets of cooperatively acting transcription factors, targets of basal transcriptional machinery, or structural elements involved in DNA packaging and access. This third possibility is the most interesting, and our data supported this interpretation in many cases.

Given a target gene set known to be responsive to some specific TF, MaMF yielded a list of motifs, which included the TF in question, but also included a large number of incorrect motifs. We hypothesized that if an incorrect motif were biologically active, the motif would more likely be found in promoters of genes coexpressed with the target gene set versus the promoters of genes not coexpressed with the target gene set. We defined the *enrichment ratio* of a motif to be the following: If the score for the best alignment of a motif against a gene promoter is called the alignment score, the enrichment ratio of a motif is the average alignment score of coexpressed genes divided by the average alignment score of non-coexpressed genes. We segregated coexpressed and non-coexpressed gene sets using expression microarray data reported in Stuart *et al.* (2003) (see Methods for details). Motifs with an enrichment ratio greater than one are considered interesting, since the expectation value is exactly 1.0 in the case where no enrichment exists. Note, however, that interesting motifs will not have enrichment ratios significantly above 1.0 because it is common to find sequences in non-coexpressed genes by chance that match the motif fairly well (since we are considering large promoter sizes), thus increasing the denominator of the ratio.

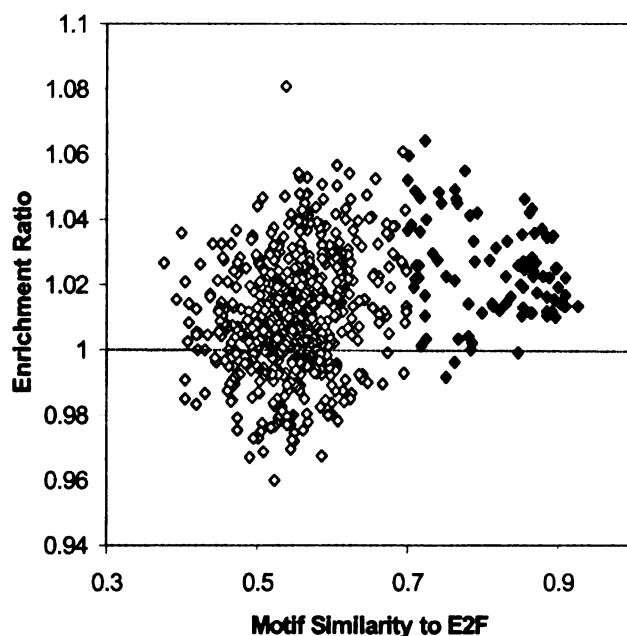


Figure 16. Scatterplot of enrichment ratio to motif similarity for the E2F gene set.

1000 motifs generated by MaMF on the E2F gene set are shown. There are two distinct clusters of motifs, one with black diamonds corresponding to motifs that look highly similar to E2F, and another with clear diamonds corresponding to incorrect motifs. The incorrect motifs are skewed to have an enrichment ratio greater than 1.0. The E2F motifs have an average enrichment ratio greater than the median enrichment ratio for all motifs.

Figure 16 shows a scatterplot of enrichment ratios versus similarity to the correct motif (using the motif similarity metric) for the MaMF generated motifs for *E2F*. There is a separation into two clusters, one representing correct *E2F* motifs (dark diamonds), the other incorrect motifs (hollow diamonds). The *E2F* cluster has enrichment ratios mostly above 1.0, reflecting the expected enrichment of *E2F* motifs within genes that are observed to co-vary their expression with known *E2F* responsive genes. Analyzing the eight TRANSFAC gene sets, we found that correct motifs from *every* gene set had an average enrichment ratio greater than one; they were also greater than the median enrichment ratio of all motifs (see Table 8). More surprising was the large number of *incorrect* motifs that nevertheless had enrichment ratios greater than 1.0. This skew toward ratios greater than 1.0 was statistically significant in 6/8 cases ($p \ll 0.0001$ for

UNIVERSITY OF CALIFORNIA

those six cases), and suggests that that many of those motifs may have real biological function.

Table 8. Enrichment ratio (ER) skew of MaMF motifs and elevated enrichment ratio of correct motifs on the eight TRANSFAC gene sets.

Using scrambled expression data, one would expect enrichment ratios to cluster around 1.0, giving approximately 500 motifs slightly above 1.0. We instead see significant skew of $p \ll 0.0001$ in 6/8 gene sets (those for which the skew is statistically significant have asterisks next to the numbers). We also see the average enrichment ratio of correct motifs to be higher than the median enrichment ratio of all motifs, where a correct motif is one with motif similarity to the annotated motif of at least 0.75.

	# motif with ER >1.0	Average ER of correct motifs	Median ER of all motifs
<i>AHR/HIF</i>	856*	1.032	1.015
<i>API</i>	580*	1.019	1.003
<i>CEBP</i>	502	1.026	0.999
<i>CREB/ATF</i>	920*	1.029	1.005
<i>E2F</i>	804*	1.021	1.013
<i>ETS</i>	587*	1.011	1.004
<i>HNFAA</i>	390	1.010	0.996
<i>SPI</i>	755*	1.011	1.010

We also made the enrichment ratio analysis on the 21 Tompa gene sets (with TFs of unknown identity). The results largely paralleled the analysis for the eight motifs presented in Table 8, with 20/21 showing mean enrichment ratios greater than 1.0 for the top 1000 motifs ($p \ll 0.01$ by exact binomial with respect to proportion of the 21 with ratios > 1.0). Of the 21, 18/21 had statistically significant skew to high enrichment ratios ($p \ll 0.01$ by exact binomial with respect to proportion of the 1000 motifs in a gene set with ratios > 1.0).

Somewhat surprisingly, in 7/21 cases, the annotated motifs of the Tompa data set had enrichment ratios less than 1.0. This stands in some contrast to the results for the TRANSFAC data set. We suspected that this resulted from the greater difficulty of the Tompa data set, in terms of uncertainty of what exactly the annotated motif was (since this was inferred) and lower conservation of the annotated motif. We observed a relationship between enrichment ratios of the annotated motifs and the relative difficulty of identifying the annotated motifs from the promoter sets. We found that for annotated

UNOT LIDRAH I

motifs with enrichment ratio greater than one, 13/14 gene sets had at least one motif finder find the correct motif. Of the gene sets with annotated motifs having an enrichment ratio less than one, only 3/7 gene sets had at least one motif finder find the correct motif. We suspect that in a number of these cases, the nominally incorrect motifs would prove to be biologically meaningful; however, since we could not say with confidence which TFs corresponded to the Tompa gene sets, we did not pursue this issue further.

5.4.4. *Co-occurring Transcription Factor Motifs*

In light of the enrichment ratio analysis, we hypothesized that the incorrect motifs with high enrichment ratios could belong to transcription factors that co-occur with the annotated one in a biologically meaningful manner. To test this, we looked for sets of motifs predicted by MaMF that matched (using the motif similarity metric at the 0.75 threshold) a motif in the TRANSFAC database. For a given set of motifs that matched a TRANSFAC motif, we computed the average enrichment ratio of these motifs. Ordering these sets of motifs by number of matches, we kept the top ten TFs that had an average enrichment ratio greater than the mean enrichment ratio of all motifs from the MaMF output, and had at least four matching motifs. For each case, this resulted in a list of well-studied motifs that were suggested to co-occur with the target TF based on a combination of sequence-based analysis and expression microarray data. We limited our analysis to the TRANSFAC data set since the annotated TFs were known.

UNOT LIDMHI

Table 9. Transcription factor motifs predicted to co-occur with annotated TRANSFAC gene sets.

Over 70% have literature evidence of such co-occurrence. Predicted co-occurring TFs were obtained by finding TFs from TRANSFAC that have many matching motifs in MaMF output, have an average enrichment ratio (ER) greater than the median enrichment ratio of all motifs, and have at least four matching motifs. A matching motif is defined to be a motif with similarity greater than 0.75 to a TRANSFAC motif.

Annotated TF	Predicted Co-occurring TF	average ER of Matching Motifs	# Matching Motifs	PubMed ID of Reference
<i>HNF4A</i>	<i>GR</i>	1.000	122	
	<i>CEBP</i>	1.008	88	7744832, 15388792
	<i>E2A</i>	0.997	38	11994285
	<i>ER</i>	1.002	37	
	<i>CREB</i>	1.002	32	15388792
	<i>GATA</i>	1.012	24	11585914
	<i>PU.1</i>	1.011	24	
	<i>ETS</i>	1.006	23	14678994
	<i>API</i>	1.005	23	1870969
	<i>HMG-IY</i>	1.027	17	
<i>E2F</i>	<i>AHR</i>	1.020	94	10644764
	<i>NF-1</i>	1.022	84	12527763
	<i>EGR</i>	1.020	73	
	<i>SMAD</i>	1.013	24	11689553
	<i>CREB</i>	1.012	22	15123636
	<i>TBP</i>	1.016	17	8255752
	<i>AP-2</i>	1.016	17	12513689
	<i>SPI</i>	1.022	13	11433027, 12513689
	<i>MYC</i>	1.020	11	12004135
	<i>ETS</i>	1.011	8	8290253
<i>SPI</i>	<i>AHR</i>	1.014	51	8647831
	<i>EGR</i>	1.014	48	9698605
	<i>E2F</i>	1.026	4	11433027, 12513689
<i>ETS</i>	<i>CEBP</i>	1.008	165	12594283
	<i>GR</i>	1.006	52	11279115
	<i>AR</i>	1.007	41	8798622
	<i>GATA</i>	1.013	37	8417360, 15537384
	<i>PU.1</i>	1.012	37	12907668
	<i>DBP</i>	1.009	30	
	<i>CREB</i>	1.004	27	9528793
	<i>API</i>	1.020	26	15537384
	<i>SMAD</i>	1.006	23	11590145
	<i>HMG-IY</i>	1.005	21	12907668
<i>CREB/ATF</i>	<i>CEBP</i>	1.008	54	15213229
	<i>ER</i>	1.027	37	11457657
	<i>API</i>	1.031	35	12200429, 9822653
	<i>MYC</i>	1.018	33	12883011
	<i>T3R</i>	1.007	24	
	<i>E2A</i>	1.009	21	
	<i>AR</i>	1.006	17	9822653
	<i>DBP</i>	1.015	13	
	<i>SPI</i>	1.007	11	12200429
	<i>YY1</i>	1.015	7	7769693
<i>CEBP</i>	<i>GR</i>	1.002	46	9442383
	<i>HMG-IY</i>	1.010	37	
	<i>GATA</i>	1.011	32	15632071
	<i>T3R</i>	1.000	31	
	<i>ETS</i>	1.003	30	12594283
	<i>TBP</i>	1.009	28	7556073

UNOT LIDNANI

	<i>API</i>	1.017	24	10942517
	<i>PU.1</i>	1.000	18	8756629
	<i>PEA3</i>	1.004	17	
<i>API</i>	<i>CEBP</i>	1.004	63	10942517
	<i>T3R</i>	1.005	59	
	<i>AHR</i>	1.005	35	
	<i>E2A</i>	1.004	29	
	<i>TCF-4</i>	1.012	25	14552705
	<i>EGR</i>	1.007	25	12457461
	<i>SMAD</i>	1.013	23	10022869,12457461
	<i>GATA</i>	1.006	22	7623817
	<i>ETS</i>	1.008	16	15537384
	<i>LF-A1</i>	1.006	15	
<i>AHR/HIF</i>	<i>MYC</i>	1.033	87	15071503
	<i>ER</i>	1.018	35	15695373
	<i>CREB</i>	1.032	24	15213229
	<i>USF</i>	1.050	22	11313255
	<i>EGR</i>	1.017	21	15047156
	<i>ETS</i>	1.024	19	12464608
	<i>PEA3</i>	1.038	13	
	<i>E2A</i>	1.018	13	
	<i>SP1</i>	1.013	13	8647831
	<i>AP-2</i>	1.012	12	

We validated the predicted co-occurring TFs by looking for literature evidence that both the annotated TF and the predicted co-occurring TF were observed to bind to the same gene promoter. Table 9 shows that 70% of the predicted TFs across all eight gene sets indeed had been shown in the literature to co-occur with the annotated TF. Additionally, the top ten predicted co-occurring TF motifs for a given gene set account for up to 40% of the “incorrect” high scoring motifs generated by MaMF, explaining a large portion of the results. These results suggest that multiple TF regulation of human genes is very common and that databases of TF interactions are probably only scratching the surface at the present time.

Given the MaMF predicted a number of TFs to co-occur with the annotated ones, we present two detailed examples where sufficient literature data exist to show that these predictions held. The first uses the *AHR/HIF* data set, shown in Table 10. Because *MYC* is predicted by MaMF to co-occur in *AHR/HIF* target genes (see Table 9), we checked to see if the 11 *AHR/HIF* target genes taken from TRANSFAC were also responsive to

MYC. We used the *MYC* Target Gene Database (Zeller, Jegga et al. 2003) and found that 7/11 were indeed annotated as *MYC* targets. Since roughly 10% of all human genes are expected to be *MYC* targets (Fernandez, Frank et al. 2003), the likelihood that this is by chance is very low ($p < 2.3 \times 10^{-5}$ by exact binomial). So it appears that *MYC* and *AHR/HIF* indeed do commonly co-occur, which supports the idea that the nominal *MYC* motifs found by MaMF in the *AHR/HIF* genes are biologically relevant.

Table 10. *AHR/HIF* target genes that are also responsive to *MYC*.

Since we predicted that *MYC* co-occurs with *AHR/HIF*, we would expect *MYC* to bind to some *AHR/HIF* target genes. Out of 11 *AHR/HIF* target genes obtained from TRANSFAC, seven are found to be *MYC* responsive, according to the *MYC* Target Gene Database (Zeller, Jegga et al. 2003) ($p < 2.3 \times 10^{-5}$ by exact binomial if 10% of all human genes are *MYC* responsive (Fernandez, Frank et al. 2003)).

Gene Target	Description	References (pubmed ID)
<i>EDNI</i>	endothelin 1	12384550 14522256, 12529326, 11139609, 10737792,
<i>TFRC</i>	transferrin receptor (p90, CD71)	12695333
<i>VEGF</i>	vascular endothelial Growth Factor	10646866, 12368264 12695333, 11353853, 10823814, 11983920,
<i>ENO1</i>	enolase 1, (alpha)	11085504
<i>FOSL1</i>	FOS-like antigen 1	12695333, 11139609
<i>HSPB1</i>	heat shock 27kDa protein 1	11085504
<i>PGK1</i>	Phosphoglycerate kinase 1	10823814

The previous example took a single predicted co-occurring TF (*MYC*) derived from the targets of *AHR/HIF* and established that many of the targets of *AHR/HIF* are also targets of *MYC*. The second example uses the annotated targets of *E2F* and asks whether the 10 different predicted co-occurring TFs from MaMF bind to *MYC*'s promoter, which is an annotated target of *E2F*. Results are shown in Table 11. In 8/10 cases, we were able to find literature evidence that there was direct binding of the predicted TF to the *MYC* promoter. While it is difficult to make a strong statistical statement about this result, it seems improbable that such a high proportion of TFs would be identified in the literature to specifically target *MYC*.

UNIVERSITY OF MICHIGAN

Table 11. Transcription factors predicted to bind onto the *MYC* promoter.

MYC is an *E2F* responsive gene, according to TRANSFAC. Given our predictions for 10 TFs that co-occur with *E2F*, we would expect that some of these TFs would therefore bind to *MYC*. Nine out of eleven have literature evidence of such binding.

TF	Average ER of Correct	Number Correct	<i>MYC</i> promoter references (Pubmed ID)
E2F	1.020644	153	8437848
AhR	1.019511	94	11114727
NF-1	1.021563	84	1945411
EGR	1.020287	73	
SMAD	1.012889	24	11689553
CREB	1.012461	22	
TBP	1.016252	17	11593411
AP-2	1.015884	17	2822255
SP1	1.021808	13	8437848
MYC	1.020263	11	8972190
ETS	1.011082	8	8290253

5.5. Discussion

We have presented a new search algorithm and scoring function for motif finding and have shown competitive performance on a small data set from lower organisms but improved performance on a human TF data set. While the problem is clearly more challenging in the human case, MaMF is able to find correct motifs, measured by quantitative similarity to the annotated motifs. However, the top scoring motif is seldom the correct one, based on annotation, and it is in a large sea of diverse motifs. By employing microarray expression data, we established that a large proportion of the nominally incorrect motifs exhibited enrichment in genes co-expressed with our TF target set.

Further, we have shown that many of these remaining high scoring motifs belong to TFs that co-occur with the annotated TF. Despite this evidence, the relationships and mechanisms between the co-occurring motifs and the annotated TFs remain unclear. It is possible that these co-occurring motifs have no functional relationship with the annotated TF and simply belong to other transcriptional programs that use similar genes. However,

UWU LIDIANI

given the importance of interactions between transcription factors to modulate transcription particularly in humans, we find it more likely that many of these co-occurring TFs interact with the annotated TF in some manner, either in cooperative or inhibitory roles. Some of the difficulty of human motif finding can be attributed to these co-occurring motifs that comprise a large portion of MaMF output. In lower organisms where the transcriptional programs are simpler, the likelihood of finding co-occurring motifs that score higher than the annotated motif is smaller. The successful human motif finder, therefore, needs to be able to search deeper and faster to enumerate both the annotated TF motif as well as co-occurring motifs.

MaMF's performance is largely dependent on two features: the use of a genomic background model and the use of indexing to accelerate sequence comparisons. With respect to the background model, higher order background models have been shown to improve motif detection (Thijs, Lescot et al. 2001), preventing spurious common sequences such as short simple repeats (that remain despite repeat masking) from being the dominant signal. In Figure 2e we validated this by showing that as the complexity of the background model increased, MaMF was increasingly able to find the correct motifs. Furthermore, when comparing algorithms in Figure 3, we observed that algorithm performance corresponds closely with the choice of the background model used, shown in Table 12. The worst performing algorithm, AlignACE, uses no background model, whereas MaMF and Weeder both use frequency-based backgrounds. Interestingly, Bioprospector does best when the background is estimated from the input sequence, suggesting that a local background may be more effective in general than a background

UNIVERSITY OF CALIFORNIA

estimated from the entire genome. We did not examine if such a strategy would also improve MaMF's performance.

Table 12. Background models used in the comparison algorithms.

Algorithms that used more complicated genome backgrounds containing more information about the genome generally did better than those that had simpler genome backgrounds.

Performance Rank	Algorithm	Background Model
1	MaMF	Frequency-based
2	Weeder	Frequency-based
3	Biopro prospector	3 rd order Markov
4	Consensus	GC weighted
5	AlignACE	Equal weighting

The second major feature that MaMF employs is the indexing of sequences to speed searches. The benefit of indexing is most evident when comparing MaMF with Weeder. While both algorithms use frequency-based backgrounds, and Weeder is competitive in performance with MaMF, MaMF runs about 100 times faster than Weeder. A large portion of this speed differential results from the exponential time penalty that Weeder incurs when searching for longer motifs, with running times of greater than one hour for motif widths of 10 and up. MaMF is able to search for longer motifs with no such performance penalty, which makes this algorithm amenable to high throughput experiments as well as longer and more complicated binding motifs.

MaMF builds upon previous algorithms by its use of indexing to optimize performance. The generation of the lookup table is similar to the enumerative approaches used by van Helden (2003) and Sinha and Tompa (2003). Whereas they enumerate all possible motifs, we enumerate all possible nmers to generate sequence pairs that share an exact nmer. Since the nmer size is significantly shorter than the motif width, our approach minimizes the polynomial time penalty that enumeration incurs. The motif generation step uses a greedy approach similar to Consensus (Hertz and Stormo 1999), except that our approach uses the indexed lookup table both to generate sequence pair

UW-MILWAUKEE

seeds and to build motifs. In Consensus, each motif is generated from a seed containing a single sequence and additional sequences to be added to the motif are exhaustively searched from the input sequences. The indexed lookup table in MaMF minimizes the redundancy of generating motifs from similar or identical seeds.

To determine if MaMF's performance stems from its scoring function or the search algorithm, we compared the average scores of both MaMF's and Bioprosector's top 30 motifs for all eight gene sets using scoring functions from both algorithms. We rescored the motifs generated by MaMF and Bioprosector using the scoring function from each algorithm. If the scoring functions were not substantially correlated, we would see in all cases that the motifs produced by algorithm A would score higher using A's scoring function than motifs produced by Algorithm B but rescored by A's scoring function. However, we saw a mixed result. Using MaMF's scoring function to score motifs, we saw that in 7/8 cases MaMF's motifs yielded a MaMF score better than the motifs produced by Bioprosector. In contrast, using Bioprosector's scoring function to score motifs, in just 3/8 cases the motifs produced by Bioprosector scored better than the motifs produced by MaMF. Since MaMF seeks to maximize its own scoring function, the fact that it did a better job of optimizing Bioprosector's scoring function in 5/8 cases suggests that the scoring functions themselves are closely related and that much of the advantage of MaMF comes from its search strategy. In the single case where MaMF found a motif that scored more poorly using its scoring function than the motif found by Bioprosector (*SPI*), we believe that this represented a search failure for MaMF. Potential reasons include a bias against GC rich seeds since they are relatively common

UNIVERSITY OF CALIFORNIA

in the genome, and a difficulty in aligning the middle C in the consensus sequence due to the greedy nature of the search strategy.

We chose an assessment strategy that differed from that used in Tompa et al. in order to better measure performance in the human case. The strategy differed in two ways. Instead of looking at the top motif, we looked at the top 30 motifs predicted by a motif finder. It is important to increase sensitivity in this way because, as we later showed, many of the highest scoring but unannotated motifs are also likely to be biologically relevant. If multiple TFs are involved in a given gene set, the annotated TF motif might not be expected to score the highest.

The second way our strategy differed from Tompa et al was that whereas Tompa measured a motif finder's ability to identify the annotated binding sites (i.e. at the nucleotide and binding site levels), we measured the degree of similarity of the predicted motif to the annotated motif using a quantitative similarity metric (i.e. at the motif level). While measuring nucleotide and site level accuracy is an important metric in identifying biologically active binding sites, we feel that it is more important to analyze motif level accuracy of the motifs predicted by a motif finder. Given confidence of a particular motif, individual binding sites can be predicted from this motif. Furthermore, measuring motif similarity is a natural extension of the traditional metric of comparing a predicted motif to the annotated consensus sequence, only that we are comparing the predicted motif to the annotated consensus *motif*, a richer representational form of the consensus sequence. As a result, our assessment methodology may be more straightforward to understand.

Note that MaMF's use of an empirical scoring function limits our ability to compute and analytical estimate of a p-value given a motif with a score of a certain

UNIVERSITY OF TORONTO

magnitude. We are exploring ways to use empirically computed distributions of scores to provide such estimates. Our use of microarray expression data suggests further inquiry, and may have application in addressing the problem of motif evaluation/validation. We have shown that the enrichment ratio is elevated for the annotated binding motifs in all eight TRANSFAC gene sets, so there may be ways to use the enrichment ratio in motif finding. For instance, one could potentially use the enrichment ratio in a post-processing step to score motifs generated by MaMF and thus yield a biologically motivated ranking, which might ameliorate the need for a statistical model of motif score likelihood. This is attractive because with the proliferation of publicly available expression microarray data (Ball, Awad et al. 2005) it is easy to calculate enrichment ratios for motifs from any gene set. More ambitiously, a motif finding algorithm could be designed around the enrichment ratio, for example by finding motifs that maximize the enrichment ratio. An advantage of this approach is that the genomic background distribution is not required since such information is embedded within the enrichment ratio.

5.6. Materials and Methods

5.6.1. *Background Model*

For the yeast and *e. coli* gene sets, the scoring function used a 3rd order Markov chain background model (similar to Liu et al. (2001)) derived from the relevant organism to assess sequence uniqueness. In human we used a non-probabilistic background distribution that counts actual sequence occurrences for a width w . To account for infrequent sequences that may skew probabilities, for each sequence we defined an exhaustive set of all related sequences that differ from the original sequence by one base

UNIVERSITY OF TORONTO

pair. Because the number of occurrences of these related sequences should be much greater than that of the lone sequence, we created a composite background frequency score for a sequence equal to the total number of occurrences of related sequences divided by the total number of sequences of width w in the data set. Additionally, in the comparison of different background models, we also used an equal weighting model where each nucleotide is equally likely to occur (equivalent to having no model at all), a GC-weighted model where Gs and Cs are less likely to occur (38%), and four Markov chain background models of the 1st, 3rd, 5th, and 7th orders. The human backgrounds were based on the promoters of all Refseq genes.

5.6.2. *Data*

For the TRANSFAC data set, we gathered 8793 gene promoters consisting of sequence -1000 to +200 bp around the transcription start site from the Database of Transcription Start Sites (DBTSS) (Ota, Suzuki et al. 2004), removing repetitive sequence using RepeatMasker (Smit, Hubley et al. 1996-2004). We exhaustively searched the TRANSFAC motif database (Wingender, Chen et al. 2001) (v. 8.1), looking for human transcription factor binding sites that contained the exact sequence at the annotated position in DBTSS. By keeping gene sets of transcription factor motifs that contained at least five genes, we ended up with seven gene sets: *E2F*, *ETS*, *SP1*, *API*, *AHR/HIF*, *CREB/ATF*, and *CEBP*. Additionally we curated a set of human *HNF4A* responsive genes that contained an annotated binding site, using genes and binding sites from <http://www.sladeklab.ucr.edu/info.html>. Upstream regions of 10,000 bp before the transcription start site were obtained from Ensembl (Hubbard, Andrews et al. 2005) to analyze MaMF's performance with larger sequences. This group of well-annotated gene

UNIVERSITY OF CALIFORNIA

sets was used to illustrate parameter sensitivity of MaMF and also for preliminary algorithm comparison (referred to in what follows as the TRANSFAC data set).

To create a data set for the purposes of rigorously comparing algorithm performance, we processed benchmark data from Tompa et al (2005) by using the following procedure:

1. We first downloaded from the Tompa et al website the real human upstream sequences, which contain gene promoters from 1000-3000 bp in length. While the Tompa paper does not make the transcription factors and their annotated motifs associated with each data set directly downloadable, the website displays the true positive binding sites for viewing. We were able to manually extract these and verify that our extracted data matched that which was displayed.
2. Our scoring methodology relied on generation of a position weight matrix from an aligned motif, which was directly available for the TRANSFAC data set. However, the binding sites for many of the Tompa gene sets were highly variable in content and in length (i.e. in many cases not easily condensable into a canonical binding motif), so we used Consensus (using parameters below) to identify a set of 11-mer binding sites to be used as the true binding motifs. We chose Consensus because it is a deterministic algorithm that can find motifs using short input sequences. It might be argued that identification of “true” motifs in this manner might bias results toward Consensus, but it should not bias results toward MaMF or other algorithms.

UWU LINDY INN

3. To remove overlapping gene sets, we identified the transcription factor each gene set was most likely responsive to, by comparing the binding sites with TRANSFAC data. The Tompa paper contained 26 human gene sets, of which five overlapped with the gene sets described above. We eliminated those five, and this resulted in 21 human gene sets. The sequences were then repeat masked using the same procedure as above.

In what follows, this set of 21 human gene sets is referred to as the Tompa data set.

5.6.3. Motif Similarity

We define a motif similarity function that compares two motifs A and B of lengths n and m , respectively, finding the best possible alignment between the two. A motif is an ungapped alignment of a set of sequences, which can be represented as a position weight matrix (PWM). For the purposes of finding the best alignment, we require the second motif B to contain sequences for which surrounding sequence is known, such that we can create an extended motif B' containing this surrounding sequence, of width $m' = 2m$. We calculate the raw similarity score by taking the motif A and aligning it against the extended motif B' , finding the best alignment that maximizes the score, defined to be the following:

$$\max \left\{ \sum_{i=1}^n \sum_{j=i+x}^m \sum_{k=A}^T A_{ik} \cdot B'_{jk} \right\} \text{ for } x = \{0, m' - n\} \quad (2)$$

where motif A and extended motif B' are the PWMs of the two motifs, i and j are the positions within the respective weight matrices, k is the nucleotide, n and m' are the widths of the weight matrices (with the requirement that $m' \geq n$), x is the offset used to align motif A to motif B , and the notation A_{ik} denotes the number of occurrences of nucleotide k at position i . The similarity score is the raw score scaled to between 0 and 1

UNIVERSITY OF TORONTO

by dividing by the maximum possible raw score, obtained by calculating the score of the consensus sequence of motif *A* multiplied by the number of sequences in motif *B*.

5.6.4. *Algorithm Comparison*

We used defaults for AlignACE, Bioprosector, Weeder, and Consensus unless otherwise specified. To make the comparison as similar as possible, we specified a motif width of 11 (if there was the option of choosing a motif width), used the most complex background model available, and used the same repeat masked human sequence as the input. Weeder was parameterized to use the launcher method in the large setting, which tries finding motifs of width 6-12 containing 1-4 mismatches, removes duplicates, and returns the best scoring motifs. We set it to use the provided human background, to search both strands of the input sequences, and to return 30 motifs. Bioprosector was set to use a width of 11, a 3rd order Markov model based on the same Refseq promoters MaMF used, and to report the top 30 motifs; it finds zero or more sites per sequence and the number of sites in a motif is determined internally by the algorithm. Consensus was parameterized to operate in a way similar to MaMF: a width of 11, a background model containing 40% GC, *n* sites per motif for *n* genes (depending on the gene set), zero or more sites per sequence, and the top 30 motifs to be returned. AlignACE used default parameters, and it automatically chooses varying motif lengths and sizes to maximize its motif score. The scoring function in our copy of Bioprosector uses an updated version of the scoring function found in Liu *et al.* (2002) (X. Liu, pers. comm.).

UNIVERSITY OF MICHIGAN

5.6.5. *Enrichment Ratio*

We developed a method to employ expression microarray data in order to provide biological validation for computationally discovered motifs (see Results). The method defines an *enrichment ratio*, which is the degree to which a motif discovered from a target gene set preferentially occurs in the promoters of genes that are co-expressed with the target gene set relative to the presence of the motif in genes that are *not* co-expressed with the target gene set. Formally, the occurrence of a motif A in a single gene promoter is measured by the alignment score z , defined to be the following:

$$\max \left\{ \sum_{i=1}^w A_{i,Q(x+i)} \right\} \text{ for } x = 1 \text{ to } n-w$$

where Q is the gene promoter sequence of length n , with the notation $Q(x+i)$ denoting the $x+i^{\text{th}}$ position in Q ; and motif A is a position weight matrix of width w , with the notation $A_{i,Q(x+i)}$ denoting the number of occurrences of nucleotide $Q(x+i)$ at position i . That is, if $A_{i,Q(x+i)}$ is the score of the motif compared to a specific region of the sequence, the alignment score z is the maximum such score over the entire sequence. Given the alignment score $z(Q)$ for sequence Q , the enrichment ratio can be defined as:

$$\frac{\frac{1}{c} \sum_{i=1}^c z(C_i)}{\frac{1}{u} \sum_{i=1}^u z(U_i)}$$

where C contains the c co-expressed genes and U contains the u non-coexpressed genes. In other words, the enrichment ratio is the quotient of the mean alignment score for co-expressed genes C divided by the mean alignment score for non-coexpressed genes U .

For the calculation of the enrichment ratio, we measured coexpression based on a collection of human expression microarray data reported in Stuart *et al.* (2003). This combined data set contains over 1200 samples, with strong overlap with genes in

UNIVERSITY OF MICHIGAN

DBTSS. The coexpression of a gene against the annotated gene set is calculated if two conditions are met: 1) pairwise Pearson's correlations can be computed between that gene and at least six genes in the annotated gene set, and 2) for each of these six gene pairs, expression values are available for both genes in at least ten samples. Coexpressed genes have a high average Pearson's correlation when compared with the annotated gene set, while non-coexpressed genes have a near zero average Pearson's correlation. For the MaMF output of a given gene set, we computed enrichment ratios for all motifs using the top 20 coexpressed genes and 100 non-coexpressed genes (those with correlations closest to zero).

We computed the null distribution of enrichment factors by generating 10,000 random PWMs of width 11 and computing their enrichment ratios against 10,000 random orderings of the expression data above (to yield different numerator and denominator gene sets). The probability of observing an enrichment ratio greater or equal to 1.035 was 0.05, 1.05 was 0.01, and 1.07 was 0.001. Note, however, that the composition of a motif also affects the distribution of ER given permuted data, so this distribution serves as a guideline for the scale of the observation of ER as opposed to a formal p value. Rather than focusing on absolute enrichment ratios, we have focused on the probability of observing skew in populations of enrichment ratios. Since the populations are large, population skew is a reliable measure and yields p-values by exact binomial computation.

5.6.6. *TRANSFAC TF Motifs*

To find TF motifs that might co-occur with our annotated motifs, we generated a large list of TF motifs curated from TRANSFAC. These motifs were required to have at

UNIVERSITY OF MICHIGAN

least two annotated binding sites obtained from either human, mouse, or rat. This yielded 146 well-described TF motifs that belonged to higher organisms.

5.7. Conclusion

In this chapter, we presented MaMF, a human targeted motif finder, and showed it performs better than other algorithms in human. We also analyzed predicted (but unannotated) high scoring motifs and found that a large percentage belonged to transcription factors that are known to co-occur with the annotated transcription factors, using a combination of sequence analysis and expression microarray data.

The next chapter focuses on the TF binding site exclusively in the binding site recognition problem. Whereas MaMF uses a PWM as its model, ANNFoRM in Chapter 6 uses a neural network to build a more sophisticated binding site model.

UNIVERSITY OF MICHIGAN

Chapter 6

ANNFoRM: Nonlinear Transcription Factor Binding Site Recognition Using Neural Networks

6.1. Abstract

Motivation: The most common techniques used to model a transcription factor binding site are consensus sequences and position weight matrices. While often effective, they cannot represent more subtle features of a binding site, such as interdependencies between nucleotides. Here we present an algorithm, ANNFoRM, which recognizes new binding sites given a corpus of known binding sites. Given a training set comprised of negatives extracted from the genome and positive examples from the corpus, the method employs an artificial neural network that creates a nonlinear model of the binding site using a nonprobabilistic background model.

Results: We have tested ANNFoRM on several sets of yeast binding sites and compared the neural network using test data against a linear model employing a position weight matrix. ANNFoRM's binding site model is better able to differentiate true

ANNFoRM

positives from false positives, particularly in larger data sets. It benefits from the ability to model interdependencies in nucleotide positions as well as the option to embed multiple types of information as input.

6.2. Introduction

In the review of the literature in Section 2.3, the problem of binding site recognition has typically been addressed in two ways: First, some approaches have tried to increase the specificity of existing models by utilizing orthogonal data. Second, other approaches have attempted to build entirely new models that overcome the independence assumption. This chapter approaches binding site recognition using the second method, specifically by using neural networks (Mitchell 1997).

Here we present a novel algorithm, ANNFoRM (Artificial Neural Network For Recognizing Motifs), for recognizing binding sites in a test set given a separate training set of binding sites. The approach directly addresses the independence assumption by employing a non-linear function for the binding site model. Further, rather than embedding a constraint on the background of the target genome by ad hoc means, the method learns the background through use of negative training data. The method models the binding site by using an artificial neural network to learn the differences between the positive training sites and negative sequences drawn from the genome. We have tested ANNFoRM on several smaller yeast data sets drawn from the SCPD (Zhu and Zhang 1999), as well as a larger data set of Rap1 responding genes from results in Lieb et al. (2001). With a small number of training examples, ANNFoRM can outperform the equivalent PWM generated by Bioprosector, though larger data sets are required to maximize its performance.

UNIVERSITY OF CALIFORNIA

1197 1197ADV

6.3. Algorithm and Implementation

6.3.1. *Overview*

We use a fully connected multilayer neural network to derive a model of the binding site. To train the neural network, two sets of data are required. The first is the positive training set of binding sites responsive to a transcription factor, obtained either experimentally or via another prediction algorithm. The second is a negative training set of background sequences from the genome, assumed not to contain positive binding sites. Each training example contains sequence information, contextual information if applicable, and whether it is a binding site. The goal is to fit the network to the data such that it can differentiate positive binding sites from negative non-binding sites. The fitting method uses the backpropagation algorithm to minimize the error between the network output value and the target value.

6.3.2. *Network structure*

The neural network has three layers with weight connections between all input nodes and hidden nodes, and hidden nodes and output nodes. The binding site is represented as $4n$ binary input nodes, where n is the size of the binding site. Described as a matrix, its four rows correspond to the four possible nucleotides, the n columns correspond to each position in the binding site, and each cell refers to one input node (see Figure 17). For each binding site position, one cell in each column is set to one, with the other cells in the column set to zero. If other types of information are to be included, additional input nodes are added (described in methods). There is a customizable number of hidden units, where more units increase the representational power but the likelihood

UWU LLM INN

of overfitting as well. There is a single output node that returns a value between 0 and 1 (one representing a true binding site).

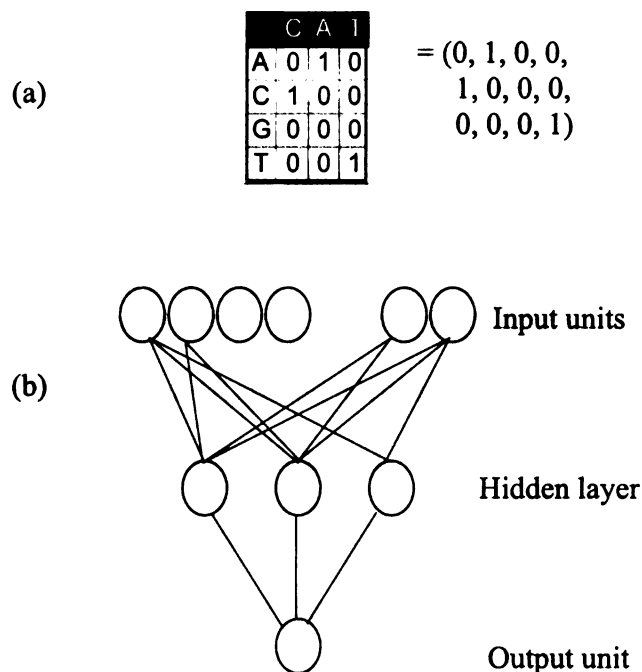


Figure 17. Neural network structure.

(a) Binding site represented as input nodes in a neural network. In this example, the sequence CAT can be formulated as a matrix of values, with the possible nucleotides in rows and the binding site in the columns. For a given nucleotide, a value one is entered in the row to which it corresponds; other cells contain a zero. The resulting values can be converted into a list to be used as the input node values for the neural network. (b) A fully connected three layer neural network. The weights are represented by the edges connecting the nodes. The input units contain binding site values and/or contextual information about the sequence. The hidden layer enables the network to model nonlinear functions. The output unit shows whether the network believes the binding site fits the model.

6.3.3. *Negative Training data*

The network is trained by standard iterative backpropagation on known binding sites and putative non-binding sites (background sequences). The set of background sequences contain many negative examples, vastly more than the number of binding sites in the positive set. To prevent the negative examples from overwhelming the positive ones, we follow a procedure that selects only non-binding sites that will contribute to the training of the network, using the following rules:

UNIVERSITY OF MICHIGAN

- Only high scoring negative binding sites (presumed to be false positives) are chosen for training. Low scoring sites have already been learned as negative sites, and do not provide much additional value.
- Because the search space is large, we choose only several sequences at a time from which to draw negative examples. This is a tradeoff between algorithm running time and completeness of the algorithm. Optimally, every possible negative example is considered every time a new negative sequence is required, but this would not be practical. In practice, a sampling technique allows successful convergence.
- To choose a new sequence to train against, we choose a threshold equal to the lowest output of the top negative sequences used in the previous iteration of training. If the highest scoring sequence in the set of potential examples retrieved from sampling is greater than the threshold, then that sequence is added to the negative list. Otherwise, the sequence is discarded and nothing is added. This is repeated a fixed number of times per iteration.
- All chosen negative examples are cached so that difficult sequences that continue to have a high score can be reintroduced for additional training.
- Sequences that are already present in the positive set of binding sites, as well as negative examples that already are stored in the cache, are ignored. This prevents the network from seeing the same training example in both classes, and prevents common sequences from being over-represented in the network model.

UWU LIDUWU

This complex regime for background sequence selection obviates the need for *any* explicit modeling of the likelihood of observing different sequences.

6.3.4. *Training the Network*

The weights in the network are initially set to random values between -0.01 and 0.01 . The negative examples cache is loaded with high scoring sequences. The following steps are then taken to train the network:

- 1) Take a high scoring negative sequence (that has not been trained on during this iteration) from the cache and modify weights by backpropagation.
- 2) Take a random positive sequence and modify weights. Repeat this up to five times while the average output of all the positive binding sites is less than 0.5 .
- 3) Repeat steps 1 and 2 as many times as twice the number of positive sequences.
- 4) Load the cache with additional negative sequences, considering up to $50,000$ binding sites.
- 5) Repeat the training regime (steps 1-4) until all the possible negative examples have been considered (about 100 iterations).

6.3.5. *Implementation*

ANNFoRM is implemented in C and compiled using GCC under the Cygwin environment on Windows. The code can therefore be compiled on other operating systems like Linux and Mac OS X. On a typical run of 100 iterations, 10 positive training

UWU LILY INN

sequences, a Pentium III-1.4 GHz class PC takes about 10 minutes to execute, using about 35 megabytes of RAM. Times increase when considering contextual information.

6.4. Materials and Methods

6.4.1. Data

Three sets of data were obtained from the SCPD (Zhu and Zhang 1999) to test ANNFoRM: 11 genes responsive to Urs1, 11 genes responsive to Rap1, and 17 genes responsive to Mcm1. Urs1 has a wide variety of targets throughout the yeast genome that are induced under stress conditions. Rap1 is best known to bind to $[C_{(1-3)}A]_n$ repeats at chromosome ends to regulate telomere length, but has a diverse set of functions. Mcm1 is involved in the regulation of cell-type specific and cell-cycle genes.

Lieb et. al (2001) conducted an exhaustive chromatin immunoprecipitation experiment on Rap1, from which we culled a large set of data. They extracted 1 kb chromatin fragments from wildtype yeast and performed immunoprecipitation reactions against polyclonal antibodies derived from Rap1. By affixing enriched DNA fragments from each IP onto a whole-genome yeast microarray, it was determined which portions of the genome were enriched and thereby which genes are targets of Rap1. They determined that 362 genes are responsive to Rap1, of which 122 are ribosomal protein genes (RPG). From this, we generated three sets of data—the 362 total set, 122 RPG set, and 240 non-RPG set.

The negative data set consisted of the 1000 bases upstream of every yeast ORF. This was obtained from SGD (<http://genome-www.stanford.edu/Saccharomyces/>)

UWU
LIIII
INN

(Cherry, Ball et al. 1997) . For convenience, only sequences with the full 1000 bases were considered.

6.4.2. General protocol

With the list of genes responsive to a particular transcription factor, Bioprospector was used to identify the binding sites, using the same parameters used in the Lieb paper (yeast intergenic regions as background model, zero or more binding sites per sequence). We tried a variety of different window sizes, ranging from 7 to 17 base pairs. We decided against manually defining the core binding site at each window size and extracting various training sets from a single Bioprospector run because it was not always obvious what the core binding site might be. The binding sites from the top set of results were extracted and verified against experimentally derived consensus sequences. The discovered binding sites were evenly split by ORFs into training and test sets, using 10 different random splits to support cross-validation estimation of performance of the various methods. We computed PWMs from the alignment of the training set of binding sites. ANNFoRM was trained against the same set of training binding sites plus negative data, using three hidden units. To avoid subtle contamination effects in training, the negative data set was also split into a training and a test set. Each contained 1000 bases of 3165 upstream regions, or approximately 6 million negative examples including the reverse complement of each sequence.

To measure performance, both the PWM and ANNFoRM's trained network were applied to the test positive and negative data sets. The output of each method for a given upstream region was defined to be the highest scoring binding site in the sequence. ROC areas were computed for the sorted output of each method. Comparisons between

methods were accomplished by considering the fraction of training/testing splits where one method outperformed the other method based on ROC area. Statistical significance was assessed by computing the probability of getting m wins out of n trials.

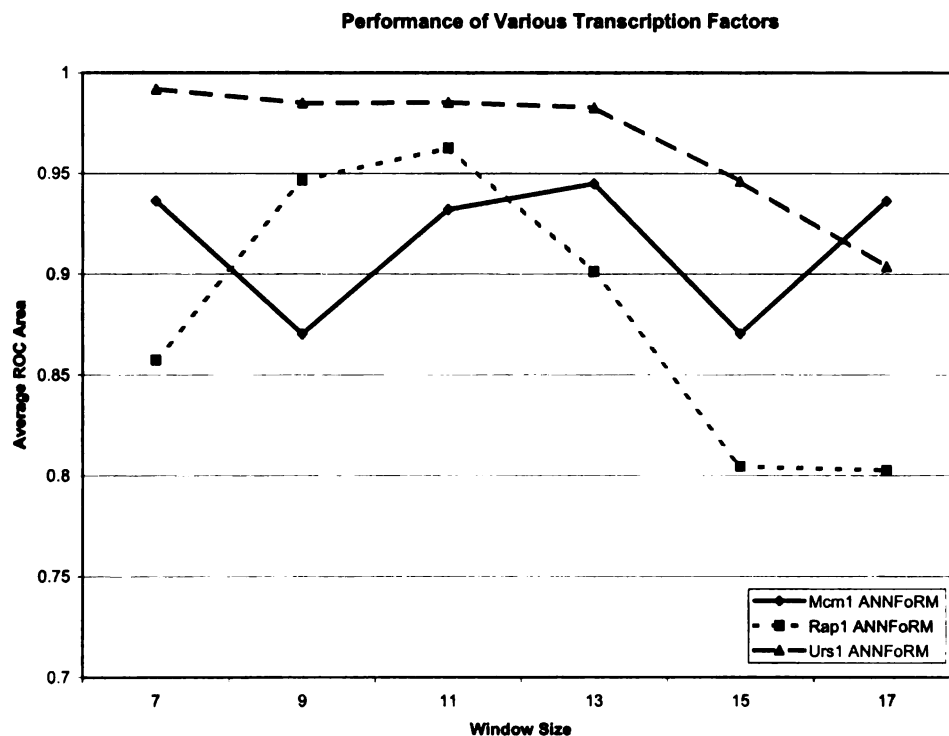


Figure 18. Average ROC areas of ANNFoRM across SCPD data sets. Each data point is an average of 10 train/test data splits. In general, averages are greater than 0.8, with decreasing performance at larger window sizes. Urs1 has on average higher ROC areas probably because of a more strongly conserved binding site.

6.5. Results

Initial development of ANNFoRM used data from the SCPD for testing. ROC areas greater than 0.8 were observed over a broad variety of parameter variations, which suggested robust convergence of the training method. Given successful convergence, we proceeded to observe ANNFoRM's performance over different window sizes for Urs1, Mcm1, and Rap1 (see Figure 18). ANNFoRM performed best at window sizes 11 and 13, approximately the average size of the binding motifs used. At smaller window sizes, the

UNIVERSITY OF CALIFORNIA

performance fluctuated significantly, depending on the transcription factor. This fluctuation seemed to be dependent on the binding site subset chosen to be in the positive data set (the particular split of the data). The Rap1 binding site has a strongly conserved C several bases away from the core sequence CACCCA, resulting in a 9 base pair full consensus sequence. Thus it is not surprising that Rap1 at window size 7 performs poorly. Similarly, the fluctuation of Mcm1 can be explained by the symmetrical nature of the binding site. DNA binding assays have suggested that the second half of the consensus CCTAATTAGG site needs to be conserved, whereas mutations in the first half are tolerated (Passmore, Elble et al. 1989). Window sizes 7 and 11 used binding sites that overlapped the second half of the consensus but window size 9 did not, so window size 9 performed the worst of the three. A shift in binding site locations would likely improve the performance at window size 9.

Table 13. Average ROC areas compared between ANNFoRM (ANN) and the PWM, for Mcm1, Rap1, and Urs1.

Window sizes range between 7 and 17.

w-size	mcm1		rap1		urs1	
	ANN	PWM	ANN	PWM	ANN	PWM
7	0.936	0.952	0.857	0.866	0.992	1.000
9	0.870	0.863	0.947	0.950	0.985	0.990
11	0.932	0.956	0.963	0.955	0.985	0.988
13	0.945	0.983	0.901	0.927	0.983	0.990
15	0.870	0.977	0.805	0.821	0.946	0.990
17	0.936	0.962	0.803	0.892	0.904	0.984

Table 13 compares the performance of ANNFoRM against the PWM approach. In general, ANNFoRM compared favorably with the PWM, often within 0.01 average ROC area. However, as the window size increased, the performance gap between the two increased, favoring the PWM. We speculated that this resulted from overfitting of the neural network. We adjusted the stopping criteria as well as the number of hidden units to

UWI LIBRARY

assess the effects of various network parameters, but did not observe major changes.

Because larger window sizes required additional input nodes, increasing the window size while keeping the positive training set constant appeared to cause the neural network to overfit by focusing on artifacts of the training data that were not representative of the binding site as a whole.

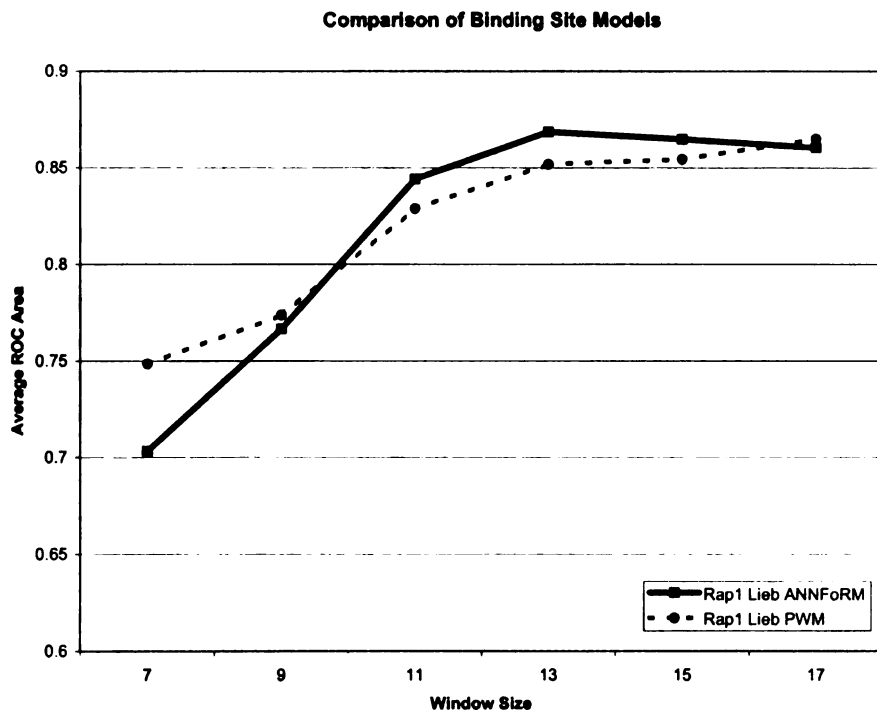


Figure 19. Comparison of ANNFoRM and PWM using Rap1 Lieb data set. Each data point is an average of 10 splits. At the middle window sizes, ANNFoRM separates true positives from false positives better than a position weight matrix.

To test this, we used the collection of Rap1 responsive genes from Lieb et al. (2001) that contained 362 genes (greater than 30 times more positive training data than each of the SCPD derived data sets). Figure 19 shows a comparison of average ROC area between ANNFoRM and the PWM approach across various window sizes. The shape of the curve for ANNFoRM remains similar to the Rap1 data from the SCPD, with peak performance at middle window sizes. However, the drop-off in performance is significantly less at larger window sizes, which is attributable to the increased size of the

UNIVERSITY OF CALIFORNIA

Lieb data set. In this case, a sufficiently large number of training examples reduces overfitting at larger window sizes. More significantly, at window sizes approximating the actual binding site size, ANNFoRM separates true positives from false positives with greater accuracy than the PWM. At a window size of 13, in all ten train/test splits, ANNFoRM had higher ROC areas than the PWM ($p < 0.001$ by t-test comparing ROC areas). A representative comparison of the ROC curve of ANNFoRM versus the PWM using this data set is shown in Figure 20. Over the entire range of true positive rates, ANNFoRM performed better than the PWM. Given a sufficient number of training examples, the neural network can generate a model that more accurately reflects the composition of the binding site than does a position weight matrix.

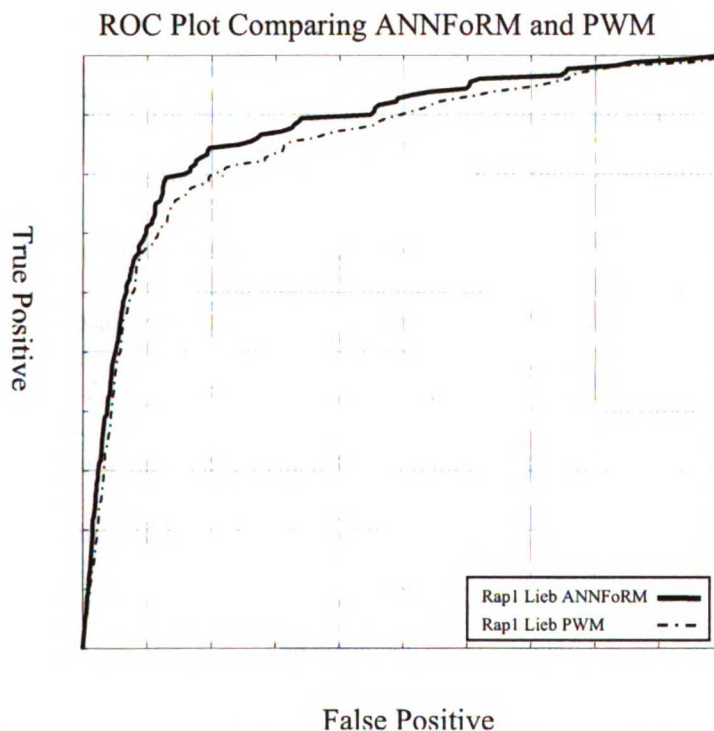


Figure 20. ROC plot comparison of a position weight matrix versus ANNFoRM of a representative split from the Rap1 Lieb data set.

ANNFoRM has an ROC area of 0.876, and the PWM has an ROC area of 0.851.

bioRxiv preprint doi: <https://doi.org/10.1101/201703.001>; this version posted March 1, 2017. The copyright holder for this preprint (which was not certified by peer review) is the author/funder, who has granted bioRxiv a license to display the preprint in perpetuity. It is made available under aCC-BY-NC-ND 4.0 International license.

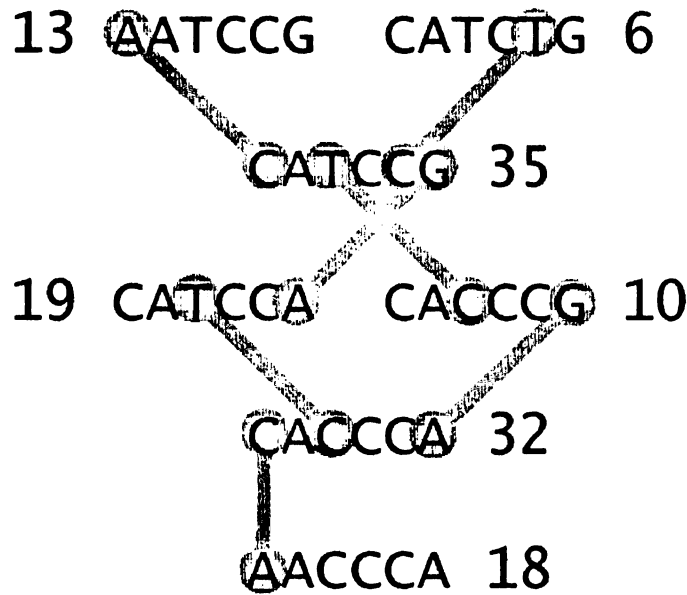


Figure 21. Representation of the core six nucleotides of Rap1 RPG set of binding sites found by Bioprospector (w=11).

The seven most common instantiations of the site are shown with the number of appearances of each 6-mer shown to the side. Sequences that are connected via a gray line differ by one base, shown by the endpoints of the line. The consensus sequence reported in the literature is CACCCA (Pina et al., 2003), but the most common sequence in this data set is CATCCG. The transformation between the two requires two mutations, and the two possible intermediate sequences CATCCA and CACCCG are significantly less common. This suggests that position 3 is interdependent on position 6. Specifically, if position 3 is a C then it is likely that position 6 is an A (and vice versa), and if position 3 is a T then position 6 is a G (and vice versa).

Because a nonlinear model such as the neural network can model sequence interdependencies, we analyzed a subset of the Rap1 target genes that were identified as ribosomal protein genes (RPG), which contains a more conserved set of binding sites. Figure 21 shows the Rap1 core binding site's common sequence variations in the Rap1 RPG data set. The two most common sequences differed by two bases, which suggests a preference that involves a base/base interdependence. With a T in position 3, a G is preferred in position 6, but with a C in position 3, an A is preferred in position 6. We considered whether the ANN approach could effectively model this interdependence and whether there would be a difference between the preferences of the ANN and PWM approaches that hinged on this apparent linkage. To do this, we chose the four most frequently observed Rap1 RPG 11-mers that differed only in positions 3 and 6, and

UNIVERSITY OF CALIFORNIA

obtained their output values from both a PWM and a trained ANN based on this data. The ANN approach learned a preference for T-G or C-A in positions 3 and 6 over T-A or C-G. The linear PWM approach cannot learn this preference, instead reaching a maximum with T-A, which mixes the most popular bases for positions 3 and 6, but ignores context. The sequences and ranks are shown in Table 14.

Table 14. Four sequences that differ in positions 3 and 6.

The boldfaced sequences are preferred by Rap1, having either a T-G or C-A in positions 3 and 6 respectively, which are the more common combinations in the Rap1 RPG data set. ANNFoRM correctly yields the highest scores for the preferred sequences. The PWM, however, is unable to score the boldfaced sequences the highest because it cannot reconcile the interdependence between the nucleotides using its additive model, instead favoring T-A and T-G in positions 3 and 6.

Position	Rank of ANN output	Rank of PWM output
1 1		
12345678901		
CATCCGTACAT	1	2
CACCCATACAT	2	3
CATCCATACAT	3	1
CACCCGTACAT	4	4

Interestingly, this particular preference (T-G and C-A over T-A and C-G, where the popular preference is T in the first position and A in the second) is precisely the exclusive OR (XOR) problem (constructing a model that separates 1-0 and 0-1 from 0-0 and 1-1). This is the classic problem that Minsky and Papert cited as the limiting case of linear models (Minsky & Papert 1969). This effect is ubiquitous in molecular systems, where, for example, making a molecular modification that adds a substituent in either of two places increases activity, but adding the substituent in both places decreases activity. This can arise from a simple volumetric constraint on binding.

We also investigated what advantages additional types of information could provide. We defined a local sequence frequency metric to be a measure of the binding site frequency relative to the local vicinity of the binding site. To generate this value a first-order Markov background model similar to Liu et al. (2001) was used. The Markov background model of a binding site used the 100 bases surrounding the binding site. The

computed metric was used in an additional input node. With this metric, one should expect that a true binding site should be relatively unique in the context of the surrounding sequence. For instance, if the Rap1 binding sequence CACCCA occurred in a long stretch of CA repeats, there would be a higher chance that the sequence occurred randomly and would be potentially less preferred. This is in fact the case for one of our test negatives, YDR545W, which has a high density of CA repeats. When testing against the Lieb Rap1 RPG set (data not shown), the original experiment without the local sequence frequency metric yielded the false positive binding site CACCCACACAC with an average rank of 156 across both positive and negative sequences. On the other hand, the experiment using the local sequence frequency metric penalized the same YDR545W binding site to an average rank of 603, increasing the rank in 9 out of 10 experiments. The local sequence frequency metric reported a -7.3 (log of the probability, larger negative numbers mean greater uniqueness) for the YDR545W site, compared with a typical -13 to -15 in test positives. Indeed, Rap1 is a multifunction transcription factor that not only is a transcriptional activator for Rap1 RPG genes, but also modulates telomeric function via telomeric DNA repeats $(C_{1-3}A)_n$ (Lieb, Liu et al. 2001), both via a similar looking binding site motif. Therefore, the neural network incorporating the local frequency metric and trained on Rap1 RPG data was correctly distinguishing Rap1 RPG sites from telomeric repeats. However, this result should be viewed as preliminary, pending additional validation experiments.

6.6. Conclusion

We have presented a method for constructing nonlinear binding site models of TFs using a neural network, called ANNFoRM, that can separate true binding sites from

false positives. By using sequences drawn from the genome as the negative training examples, the approach does not require any probabilistic genomic background model. ANNFoRM performs competitively with the position weight matrix approach, with improved performance when larger data sets are available for parameter estimation. Larger data sets appear to support extraction of information about interdependencies within a TF binding site. As larger sets of human data become more available, this method should readily extend to human.

UNIVERSITY OF CALIFORNIA

INS. I. P. R. A. R. Y.

Chapter 7

Conclusion

This work presents several approaches to addressing transcription factor binding site modeling in higher organisms. The contribution of this work includes both methodological and algorithmic ideas and biological insight into transcriptional regulation. The results were predicated on two strategies: 1) that fast and efficient computational methods such as indexing are critical to analyzing the complexity of higher organisms, and 2) that the integration of different types of data, specifically sequence and expression data, enables observations about biology that were not otherwise possible. After recapitulating the results of this thesis, I will assess the success and future of these two strategies.

There were several important biological results presented in this thesis. In Chapter 4 we showed a relationship between coexpressed genes and repetitive element structure. Such a relationship points to the possibility that repetitive elements are not “junk” DNA but may play an important part in regulating transcription and DNA packaging and access. In Chapter 5, we presented a novel motif finder that was shown to work in human. Of particular note, among the high scoring but unannotated motifs were a large number of motifs that matched other TF motifs known to co-occur with the annotated

UNIVERSITY OF CALIFORNIA

ICSF LIBRARY

motifs. This points to the complexity not only of human transcription, but also of motif finding in human. In Chapter 6, we affirmed the nonlinearity of TF binding sites.

At the same time, we contributed a number of novel algorithmic methods to facilitate TF modeling in higher organisms. Many of these methods incorporated indexing for optimization. In Chapter 3, we introduced HGS, a disk-based genomic mapping utility that identifies the location of specific sequences. In Chapter 4, we devised an efficient similarity metric for comparing large sequences such as human upstream regions. In Chapter 5, we designed and implemented an index-based greedy search algorithm called MaMF for performing motif finding in mammalian gene sets. MaMF did better in finding motifs on human gene sets than several other motif finders. In Chapter 6, we used neural networks to model TF binding sites in an algorithm called ANNFoRM so that the resulting models could be used to recognize new binding sites. We found that ANNFoRM could recognize binding sites better than its position weight matrix equivalent due to the neural network's ability to model nonlinearity.

All together this thesis offers a view into human transcriptional regulation, showing that computational methods in modeling transcription factor binding sites are not limited to lower organisms but can be used to elucidate the complexities of mammalian systems. As additional experimental and genomic data become available, further work incorporating comparative genomics, ChIP arrays, and other high throughput data will likely continue to advance our understanding of transcriptional regulation.

These results affirm the utility of data integration and fast and efficient computational algorithms as strategies to uncover the complexities of higher organisms.

Without indexing, large scale approaches to sequence comparison, genomic mapping, and motif finding would have been essentially impossible. The developed methods that now take seconds to run would instead take hours, preventing meaningful analysis of large datasets. Without the integration of sequence and expression data, sequence-level features such as repetitive elements and transcription factor binding sites could not be linked with patterns of gene expression, preventing the observations about biological phenomena made in this thesis. These two pillars of research strategy, data integration and fast and deep computational analysis, made the results possible.

In many ways, these two strategies are common to a lot of bioinformatics research and probably represent a direction bioinformatics needs to focus on in order to 1) produce biological discoveries, not just methodological advances, and 2) differentiate itself from other related disciplines such as biophysics and biostatistics. In a small sampling of recent high impact bioinformatics literature, several examples utilize these strategies heavily, particularly with respect to data integration. With respect to motif finding, MDScan (Liu, Brutlag et al. 2002) was unique for its use of ChIP-chip data for its algorithm in order to find sequence motifs. More broadly, high impact papers elucidating regulatory networks regularly utilize combinations of sequence, comparative genomics, expression, gene annotation, and outcomes data (Segal, Shapira et al. 2003; Stuart, Segal et al. 2003; Segal, Friedman et al. 2004). Clearly, bioinformatics has the potential for yielding important insights in biology.

Bioinformatics has been widely heralded as the next age of biology for the last several years, but its promise has yet been fulfilled. Science watchers have eagerly anticipated each major technology advance—first with expression microarrays and then

CONFIDENTIAL

NSF LIBRARY

with the completion of the human genome project—believing that the data from these tools would translate into biological discoveries. As it turned out, and perhaps not surprisingly, the data by themselves could not solve biology. However, given the success of data integration in this thesis and other research, it is likely that the intersection of present and future data sources will yield novel insights and lead to deeper biological insights into biology.

Complementing our current microarray technology and the sequenced human genome will be a slew of new technologies that are already in the pipeline: microfluidics, SNP chips, nanotechnology-based biosensors, etc. As it is, currently available technologies are rapidly improving. DNA sequencing speeds are accelerating at an exponential rate similar to Moore's Law in the semiconductor world, and will soon make sequencing an individual's genome commonplace. Microarrays are approaching 100 thousand measurements per chip. We will come to depend on computationally intensive algorithms that incorporate a great deal of domain-specific knowledge in organizing and interpreting all of it.

Whether we call these efforts bioinformatics, functional genomics, or systems biology, it is clear that such methods will be at the center of the discovery of new drugs and a deeper understanding of human biology and health.

bioinformatics

U.S. LIBRARY

Bibliography

- Altschul, S. F. (1989). "Gap costs for multiple sequence alignment." J Theor Biol **138**(3): 297-309.
- Altschul, S. F., T. L. Madden, et al. (1997). "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs." Nucleic Acids Res **25**(17): 3389-402.
- Anderson, S. (1981). "Shotgun DNA sequencing using cloned DNase I-generated fragments." Nucleic Acids Res **9**(13): 3015-27.
- Bailey, T. L. and C. Elkan (1994). "Fitting a mixture model by expectation maximization to discover motifs in biopolymers." Proc Int Conf Intell Syst Mol Biol **2**: 28-36.
- Ball, C. A., I. A. Awad, et al. (2005). "The Stanford Microarray Database accommodates additional microarray platforms and data formats." Nucleic Acids Res **33 Database Issue**: D580-2.
- Benos, P. V., M. L. Bulyk, et al. (2002). "Additivity in protein-DNA interactions: how good an approximation is it?" Nucleic Acids Res **30**(20): 4442-51.
- Bernardi, G. (1995). "The human genome: organization and evolutionary history." Annu Rev Genet **29**: 445-76.
- Birnbaum, K., P. N. Benfey, et al. (2001). "cis element/transcription factor analysis (cis/TF): a method for discovering transcription factor/cis element relationships." Genome Res **11**(9): 1567-73.
- Blanchette, M. and M. Tompa (2002). "Discovery of regulatory elements by a computational method for phylogenetic footprinting." Genome Res **12**(5): 739-48.
- Boffelli, D., J. McAuliffe, et al. (2003). "Phylogenetic shadowing of primate sequences to find functional regions of the human genome." Science **299**(5611): 1391-4.
- Cherry, J. M., C. Ball, et al. (1997). "Genetic and physical maps of *Saccharomyces cerevisiae*." Nature **387**(6632 Suppl): 67-73.
- Chesnokov, I., W. M. Chu, et al. (1996). "p53 inhibits RNA polymerase III-directed transcription in a promoter-dependent manner." Mol Cell Biol **16**(12): 7084-8.
- Chu, W. M., R. Ballard, et al. (1998). "Potential Alu function: regulation of the activity of double-stranded RNA-activated kinase PKR." Mol Cell Biol **18**(1): 58-68.
- Cliften, P., P. Sudarsanam, et al. (2003). "Finding functional features in *Saccharomyces* genomes by phylogenetic footprinting." Science **301**(5629): 71-6.

U
N
I
V
E
R
S
I
T
Y
O
F
T
E
X
A
S

11027 LIBRARY

- Cliften, P. F., L. W. Hillier, et al. (2001). "Surveying Saccharomyces genomes to identify functional elements by comparative DNA sequence analysis." Genome Res **11**(7): 1175-86.
- Conlon, E. M., X. S. Liu, et al. (2003). "Integrating regulatory motif discovery and genome-wide expression analysis." Proc Natl Acad Sci U S A **100**(6): 3339-44.
- Consortium, I. H. G. S. (2004). "Finishing the euchromatic sequence of the human genome." Nature **431**(7011): 931-45.
- Crooks, G. E., G. Hon, et al. (2004). "WebLogo: a sequence logo generator." Genome Res **14**(6): 1188-90.
- Davis, R., H. Shrobe, et al. (1993). What is Knowledge Representation. AI Magazine. **Spring 1993**: 17-33.
- Deininger, P. L. and M. A. Batzer (1999). "Alu repeats and human disease." Mol Genet Metab **67**(3): 183-93.
- Dietterich, T. G., A. N. Jain, et al. (1994). A Comparison of Dynamic Reposing and Tangent Distance for Drug Activity Prediction. Advances in Neural Information Processing Systems **6**. J. Alspector. San Mateo, CA, Morgan Kaufmann.
- Djordjevic, M., A. M. Sengupta, et al. (2003). "A biophysical approach to transcription factor binding site discovery." Genome Res **13**(11): 2381-90.
- Down, T. A. and T. J. Hubbard (2005). "NestedMICA: sensitive inference of over-represented motifs in nucleic acid sequence." Nucleic Acids Res **33**(5): 1445-53.
- el-Deiry, W. S., S. E. Kern, et al. (1992). "Definition of a consensus binding site for p53." Nat Genet **1**(1): 45-9.
- Ellrott, K., C. Yang, et al. (2002). "Identifying transcription factor binding sites through Markov chain optimization." Bioinformatics **18 Suppl 2**: S100-S109.
- Fernandez, P. C., S. R. Frank, et al. (2003). "Genomic targets of the human c-Myc protein." Genes Dev **17**(9): 1115-29.
- Gasch, A. P. and M. B. Eisen (2002). "Exploring the conditional coregulation of yeast gene expression through fuzzy k-means clustering." Genome Biol **3**(11): RESEARCH0059.
- Goffeau, A., B. G. Barrell, et al. (1996). "Life with 6000 genes." Science **274**(5287): 546, 563-7.
- Golub, T. R., D. K. Slonim, et al. (1999). "Molecular classification of cancer: class discovery and class prediction by gene expression monitoring." Science **286**(5439): 531-7.
- GuhaThakurta, D. and G. D. Stormo (2001). "Identifying target sites for cooperatively binding factors." Bioinformatics **17**(7): 608-21.
- Gupta, S. K., J. D. Kececioglu, et al. (1995). "Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment." J Comput Biol **2**(3): 459-72.
- Hamdi, H. K., H. Nishio, et al. (2000). "Alu-mediated phylogenetic novelties in gene regulation and development." J Mol Biol **299**(4): 931-9.
- Hausler, D. (2001). UCSC Human Genome Project Working Draft, August 2001 Assembly.
- Hellmann-Blumberg, U., M. F. Hintz, et al. (1993). "Developmental differences in methylation of human Alu repeats." Mol Cell Biol **13**(8): 4523-30.

1107 LIBRARY

- Hertz, G. Z. and G. D. Stormo (1999). "Identifying DNA and protein patterns with statistically significant alignments of multiple sequences." Bioinformatics **15**(7-8): 563-77.
- Hon, L. S. and A. N. Jain (2003). "Compositional structure of repetitive elements is quantitatively related to co-expression of gene pairs." J Mol Biol **332**(2): 305-10.
- Hong, P., X. S. Liu, et al. (2005). "A boosting approach for motif modeling using ChIP-chip data." Bioinformatics **21**(11): 2636-43.
- Hubbard, T., D. Andrews, et al. (2005). "Ensembl 2005." Nucleic Acids Res **33**(Database issue): D447-53.
- Jain, A. N. (2000). "Morphological similarity: a 3D molecular similarity method correlated with protein-ligand recognition." J Comput Aided Mol Des **14**(2): 199-213.
- Jain, A. N. (2004). "Ligand-Based Structural Hypotheses for Virtual Screening." J Med Chem **47**(4): 947-961.
- Jain, A. N., T. G. Dietterich, et al. (1994). "A shape-based machine learning tool for drug design." J Comput Aided Mol Des **8**(6): 635-52.
- Jensen, L. J. and S. Knudsen (2000). "Automatic discovery of regulatory patterns in promoter regions based on whole cell expression data and functional annotation." Bioinformatics **16**(4): 326-33.
- Jurka, J. (2000). "Rebase update: a database and an electronic journal of repetitive elements." Trends Genet **16**(9): 418-20.
- Keles, S., M. van der Laan, et al. (2002). "Identification of regulatory elements using a feature selection method." Bioinformatics **18**(9): 1167-75.
- Kent, W. J. (2002). "BLAT--the BLAST-like alignment tool." Genome Res **12**(4): 656-64.
- Kim, C., C. M. Rubin, et al. (2001). "Genome-wide chromatin remodeling modulates the Alu heat shock response." Gene **276**(1-2): 127-33.
- Lander, E. S., L. M. Linton, et al. (2001). "Initial sequencing and analysis of the human genome." Nature **409**(6822): 860-921.
- Lander, E. S., L. M. Linton, et al. (2001). "Initial sequencing and analysis of the human genome." Nature **409**(6822): 860-921.
- Lawrence, C. E., S. F. Altschul, et al. (1993). "Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment." Science **262**(5131): 208-14.
- Levine, M. and R. Tjian (2003). "Transcription regulation and animal diversity." Nature **424**(6945): 147-51.
- Li, H., V. Rhodius, et al. (2002). "Identification of the binding sites of regulatory proteins in bacterial genomes." Proc Natl Acad Sci U S A **99**(18): 11772-7.
- Lieb, J. D., X. Liu, et al. (2001). "Promoter-specific binding of Rap1 revealed by genome-wide maps of protein-DNA association." Nat Genet **28**(4): 327-34.
- Liu, X., D. L. Brutlag, et al. (2001). "BioProspector: discovering conserved DNA motifs in upstream regulatory regions of co-expressed genes." Pac Symp Biocomput: 127-38.
- Liu, X. S., D. L. Brutlag, et al. (2002). "An algorithm for finding protein-DNA binding sites with applications to chromatin-immunoprecipitation microarray experiments." Nat Biotechnol **20**(8): 835-9.

117

ICSF LIBRARY

- Liu, Y., X. S. Liu, et al. (2004). "Eukaryotic regulatory element conservation analysis and identification using comparative genomics." Genome Res **14**(3): 451-8.
- Maxam, A. M. and W. Gilbert (1977). "A new method for sequencing DNA." Proc Natl Acad Sci U S A **74**(2): 560-4.
- McCue, L., W. Thompson, et al. (2001). "Phylogenetic footprinting of transcription factor binding sites in proteobacterial genomes." Nucleic Acids Res **29**(3): 774-82.
- McCue, L. A., W. Thompson, et al. (2002). "Factors influencing the identification of transcription factor binding sites by cross-species comparison." Genome Res **12**(10): 1523-32.
- McGuire, A. M., J. D. Hughes, et al. (2000). "Conservation of DNA regulatory motifs and discovery of new motifs in microbial genomes." Genome Res **10**(6): 744-57.
- Mitchell, T. M. (1997). Artificial Neural Networks. Machine Learning. C. L. Liu, McGraw Hill: 81-101.
- Moretti, P., K. Freeman, et al. (1994). "Evidence that a complex of SIR proteins interacts with the silencer and telomere-binding protein RAP1." Genes Dev **8**(19): 2257-69.
- Nozell, S. and X. Chen (2002). "p21B, a variant of p21(Waf1/Cip1), is induced by the p53 family." Oncogene **21**(8): 1285-94.
- Odom, D. T., N. Zizlsperger, et al. (2004). "Control of pancreas and liver gene expression by HNF transcription factors." Science **303**(5662): 1378-81.
- Oliver, J. L., P. Carpena, et al. (2002). "Isochore chromosome maps of the human genome." Gene **300**(1-2): 117-27.
- Ota, T., Y. Suzuki, et al. (2004). "Complete sequencing and characterization of 21,243 full-length human cDNAs." Nat Genet **36**(1): 40-5.
- Passmore, S., R. Elble, et al. (1989). "A protein involved in minichromosome maintenance in yeast binds a transcriptional enhancer conserved in eukaryotes." Genes Dev **3**(7): 921-35.
- Pavesi, G., G. Mauri, et al. (2001). "An algorithm for finding signals of unknown length in DNA sequences." Bioinformatics **17** Suppl 1: S207-14.
- Pesole, G., G. Bernardi, et al. (1999). "Isochore specificity of AUG initiator context of human genes." FEBS Lett **464**(1-2): 60-2.
- Pruitt, K. D. and D. R. Maglott (2001). "RefSeq and LocusLink: NCBI gene-centered resources." Nucleic Acids Res **29**(1): 137-40.
- Pudimat, R., E. G. Schukat-Talamazzini, et al. (2005). "A multiple-feature framework for modelling and predicting transcription factor binding sites." Bioinformatics.
- Ross, D. T., U. Scherf, et al. (2000). "Systematic variation in gene expression patterns in human cancer cell lines." Nat Genet **24**(3): 227-35.
- Roth, F. P., J. D. Hughes, et al. (1998). "Finding DNA regulatory motifs within unaligned noncoding sequences clustered by whole-genome mRNA quantitation." Nat Biotechnol **16**(10): 939-45.
- Sanger, F. and A. R. Coulson (1975). "A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase." J Mol Biol **94**(3): 441-8.
- Schmid, C. W. (1996). "Alu: structure, origin, evolution, significance and function of one-tenth of human DNA." Prog Nucleic Acid Res Mol Biol **53**: 283-319.

11057 LIBRARY

- Schmid, C. W. (1998). "Does SINE evolution preclude Alu function?" Nucleic Acids Res **26**(20): 4541-50.
- Schmid, C. W. and W. R. Jelinek (1982). "The Alu family of dispersed repetitive sequences." Science **216**(4550): 1065-70.
- Schug, J. and G. C. Overton (1997). TESS: Transcription Element Search Software on the WWW', Computational Biology and Informatics Laboratory
School of Medicine
University of Pennsylvania.
- Segal, E., N. Friedman, et al. (2004). "A module map showing conditional activity of expression modules in cancer." Nat Genet **36**(10): 1090-8.
- Segal, E., M. Shapira, et al. (2003). "Module networks: identifying regulatory modules and their condition-specific regulators from gene expression data." Nat Genet **34**(2): 166-76.
- Shin, T. H., A. J. Paterson, et al. (1995). "p53 stimulates transcription from the human transforming growth factor alpha promoter: a potential growth-stimulatory role for p53." Mol Cell Biol **15**(9): 4694-701.
- Sinha, S. and M. Tompa (2003). "YMF: A program for discovery of novel transcription factor binding sites by statistical overrepresentation." Nucleic Acids Res **31**(13): 3586-8.
- Smit, A. F. A., R. Hubley, et al. (1996-2004). RepeatMasker Open-3.0.
- Smith, T. F. and M. S. Waterman (1981). "Identification of common molecular subsequences." J Mol Biol **147**(1): 195-7.
- Sobczak, K. and W. J. Krzyzosiak (2002). "Structural determinants of BRCA1 translational regulation." J Biol Chem **277**(19): 17349-58.
- Staunton, J. E., D. K. Slonim, et al. (2001). "Chemosensitivity prediction by transcriptional profiling." Proc Natl Acad Sci U S A **98**(19): 10787-92.
- Stormo, G. D. (2000). "DNA binding sites: representation and discovery." Bioinformatics **16**(1): 16-23.
- Stormo, G. D. and G. W. Hartzell, 3rd (1989). "Identifying protein-binding sites from unaligned DNA fragments." Proc Natl Acad Sci U S A **86**(4): 1183-7.
- Stuart, J. M., E. Segal, et al. (2003). "A gene-coexpression network for global discovery of conserved genetic modules." Science **302**(5643): 249-55.
- Thijs, G., M. Lescot, et al. (2001). "A higher-order background model improves the detection of promoter regulatory elements by Gibbs sampling." Bioinformatics **17**(12): 1113-22.
- Tompa, M., N. Li, et al. (2005). "Assessing computational tools for the discovery of transcription factor binding sites." Nat Biotechnol **23**(1): 137-44.
- Urano, T., H. Nishimori, et al. (1997). "Cloning of P2XM, a novel human P2X receptor gene regulated by p53." Cancer Res **57**(15): 3281-7.
- van Helden, J. (2003). "Regulatory sequence analysis tools." Nucleic Acids Res **31**(13): 3593-6.
- van Helden, J., B. Andre, et al. (1998). "Extracting regulatory sites from the upstream region of yeast genes by computational analysis of oligonucleotide frequencies." J Mol Biol **281**(5): 827-42.

UCSF LIBRARY

- Wagner, A. (1999). "Genes regulated cooperatively by one or more transcription factors and their identification in whole eukaryotic genomes." Bioinformatics **15**(10): 776-84.
- Wang, T. and G. D. Stormo (2003). "Combining phylogenetic data with co-regulated genes to identify regulatory motifs." Bioinformatics **19**(18): 2369-80.
- Waterston, R. H., K. Lindblad-Toh, et al. (2002). "Initial sequencing and comparative analysis of the mouse genome." Nature **420**(6915): 520-62.
- White, R. J. (2001). Gene Transcription: Mechanisms and Control. Oxford, Blackwell Science Ltd.
- Wingender, E., X. Chen, et al. (2001). "The TRANSFAC system on gene expression regulation." Nucleic Acids Res **29**(1): 281-3.
- Wolfsberg, T. G., A. E. Gabrielian, et al. (1999). "Candidate regulatory sequence elements for cell cycle-dependent transcription in *Saccharomyces cerevisiae*." Genome Res **9**(8): 775-92.
- Workman, C. T. and G. D. Stormo (2000). "ANN-Spec: a method for discovering transcription factor binding sites with improved specificity." Pac Symp Biocomput: 467-78.
- Xie, X., J. Lu, et al. (2005). "Systematic discovery of regulatory motifs in human promoters and 3' UTRs by comparison of several mammals." Nature **434**(7031): 338-45.
- Yona, G. and M. Levitt (2002). "Within the twilight zone: a sensitive profile-profile comparison tool based on information theory." J Mol Biol **315**(5): 1257-75.
- Zeller, K. I., A. G. Jegga, et al. (2003). "An integrated database of genes responsive to the Myc oncogenic transcription factor: identification of direct genomic targets." Genome Biol **4**(10): R69.
- Zhou, Q. and J. S. Liu (2004). "Modeling within-motif dependence for transcription factor binding site predictions." Bioinformatics **20**(6): 909-16.
- Zhu, J. and M. Q. Zhang (1999). "SCPD: a promoter database of the yeast *Saccharomyces cerevisiae*." Bioinformatics **15**(7-8): 607-11.
- Zhu, Z., Y. Pilpel, et al. (2002). "Computational identification of transcription factor binding sites via a transcription-factor-centric clustering (TFCC) algorithm." J Mol Biol **318**(1): 71-81.

U.S. LIBRARY

Appendix: Documentation of Code and Data

This section describes how the results were generated in Chapter 4 and Chapter 5, which contain the published theoretical contributions of this work. The appendix should allow the reader to rerun the primary experiments and in the case of MaMF understand the software architecture. For specifics such as data files, the reader should consult the accompanying DVD to view files and code.



U.S. LIBRARY

Appendix A Command Summary for Analysis of Functions F and N	123
Appendix B MaMF Usage and Documentation	129
B.1. Sample MaMF Usage.....	129
B.1.1. Sample Experiments	129
B.1.2. MaMF Mode Documentation.....	134
B.1.3. Perl Script Documentation	140
B.2. Documentation of C Files	142
B.2.1. analysis.c	143
B.2.2. ba2d.c	161
B.2.3. background.c	162
B.2.4. dadbl.c	171
B.2.5. daint.c	174
B.2.6. darrv.c	176
B.2.7. dasl.c	177
B.2.8. dastr.c	183
B.2.9. inthash.c	187
B.2.10. matchlist.c	187
B.2.11. microarray.c	191
B.2.12. motif-finder.c	198
B.2.13. oldhash.c	199
B.2.14. pwm.c	203
B.2.15. score.c	211
B.2.16. search.c	215
B.2.17. sequence.c	226
B.2.18. sitelist.c	230
B.2.19. utils.c	236

Appendix A Command Summary for Analysis of Functions F and N

This appendix summarizes the commands used to generate the results in Chapter

3. The original set of commands is stored in a Makefile, from which one can recreate the results given the correct input data.

requires golub-expr.tab, Hs.data

extract the accessions from the expression array table

```
perl get-col.pl golub-expr.tab --col=0 golub-accs.list
```

parse the big unigene file so we can get NMs out of it

```
perl parse-unigene.pl Hs.data Hs-data.tab
```

use parsed unigene file to get the acc->nm mapping

```
perl get-nm-from-unigene.pl golub-accs.list Hs-data.tab golub-nms.tab
```

extract all relevant seqs from table (2 accs and NM)

```
perl get-col.pl golub-nms.tab --nodups --col=0,1 golub-seq.list
```

requires golub-seq.fa

extract the accessions from the expression array table

```
perl get-col.pl golub-expr.tab --col=0 golub-accs.list
```

parse the big unigene file so we can get NMs out of it

```
perl parse-unigene.pl Hs.data Hs-data.tab
```

use parsed unigene file to get the acc->nm mapping

```
perl get-nm-from-unigene.pl golub-accs.list Hs-data.tab golub-nms.tab
```

```
formatdb -i golub-seq.fa -p F -o T
```

```
perl verify-similarity.pl --nm=1 --acc=0 -e=1e-5 golub-nms.tab golub-  
seq.fa golub-nms-blasted.tab
```

extract the NMs from the table

```

perl get-col.pl golub-nms-blasted.tab --nodups --col=1 golub-nms.list
echo ++ Step 2 done. Now manually get genbank entries
echo ++ from golub-nms.list and name as golub-nms.genbank.
echo ++ Then proceed to step3.
perl get-nm-cds.pl golub-nms.genbank golub-cds.fa
echo ++ Step 3 done. Now run hgs on golub-cds.fa and save
echo ++ as golub-cds-hgs.out. Then continue with step 3.

# generated golub-cds-hgs.out from step3

echo >> golub-summary.txt
echo Summary of the golub data set on Mon Jun  6 20:05:59 PST 2005 >>
  golub-summary.txt
echo ----- >> golub-summary.txt
make step4 DATA=golub --warn-undefined-variables --just-print | perl -
  ne 'print $_ unless /summary/' >> golub-summary.txt
make compare -C ../hgs/code
perl parse-hgs-hq.pl golub-cds-hgs.out golub-cds-hgs.tab
perl get-nm-cds.pl golub-nms.genbank golub-cds.fa
formatdb -i golub-cds.fa -p F -o T
perl blast-compare-aba.pl golub-cds.fa 0.002 golub-dups.tab golub-
  dups.list
perl remove-hgs-dups.pl golub-cds-hgs.tab golub-dups.list golub-cds-
  hgs-nr.tab
gcc -o calc-corr calc-corr.c

# extract the accessions from the expression array table
perl get-col.pl golub-expr.tab --col=0 golub-accs.list

# parse the big unigene file so we can get NMs out of it
perl parse-unigene.pl Hs.data Hs-data.tab

# use parsed unigene file to get the acc->nm mapping
perl get-nm-from-unigene.pl golub-accs.list Hs-data.tab golub-nms.tab
perl get-nm-expr-array.pl golub-cds-hgs-nr.tab golub-nms.tab golub-
  expr.tab golub-nm-expr.tab
./calc-corr golub-nm-expr.tab golub-exprcorr.tab

# Get the positive data set (depends on what experiment)
perl get-corr-in-range.pl golub-exprcorr.tab 0.5 1.0 golub-exprcorr-
  pos.tab

# Do the standard positive concordant comparison
compare.exe -indexdir f:/lhon/hgs/index/ -hgs golub-cds-hgs-nr.tab -
  pairs golub-exprcorr-pos.tab -start -10000 -end -1 -nmer 6 -window
  200 -threshold 4 -o golub-f-pos.tab

# Get the control data set (defined as values with -0.2<corr<0.2
perl get-corr-in-range.pl golub-exprcorr.tab -0.2 0.2 golub-zero-ec-
  all.tab

# Get subset of the lines in file, in this case for the large zero corr file
perl get-approx-n-lines.pl golub-zero-ec-all.tab 34000 golub-exprcorr-
  zero.tab

# Do the standard control concordant comparison
compare.exe -indexdir f:/lhon/hgs/index/ -hgs golub-cds-hgs-nr.tab -
  pairs golub-exprcorr-zero.tab -start -10000 -end -1 -nmer 6 -window
  200 -threshold 4 -n 34000 -o golub-f-ctrl.tab

```

```

echo Function F >> golub-summary.txt
./rocd golub-f-pos.tab golub-f-ctrl.tab -a >> golub-summary.txt

# Do the standard positive naive comparison (only 1000 bases upstream)
compare.exe -indexdir f:/lhon/hgs/index/ -method 1 -hgs golub-cds-hgs-
nr.tab -pairs golub-exprcorr-pos.tab -start -1000 -end -1 -nmer 6 -
window 200 -threshold 4 -o golub-n-pos.tab

# Do the standard control naive comparison (only 1000 bases upstream)
compare.exe -indexdir f:/lhon/hgs/index/ -method 1 -hgs golub-cds-hgs-
nr.tab -pairs golub-exprcorr-zero.tab -start -1000 -end -1 -nmer 6 -
window 200 -threshold 4 -n 34000 -o golub-n-ctrl.tab

echo Function N \ (1000 bases upstream) >> golub-summary.txt
./rocd golub-n-pos.tab golub-n-ctrl.tab -a >> golub-summary.txt

# Do the standard positive dotplot comparison
compare.exe -indexdir f:/lhon/hgs/index/ -method 2 -hgs golub-cds-hgs-
nr.tab -pairs golub-exprcorr-pos.tab -start -10000 -end -1 -nmer 6 -
window 200 -threshold 4 -o golub-np-pos.tab

# Do the standard control dotplot comparison
compare.exe -indexdir f:/lhon/hgs/index/ -method 2 -hgs golub-cds-hgs-
nr.tab -pairs golub-exprcorr-zero.tab -start -10000 -end -1 -nmer 6
-window 200 -threshold 4 -n 34000 -o golub-np-ctrl.tab

echo Function N' \ (10,000 bases upstream) >> golub-summary.txt
./rocd golub-np-pos.tab golub-np-ctrl.tab -a >> golub-summary.txt

# Do positive comparison on transcripts using N
compare.exe -indexdir f:/lhon/hgs/index/ -method 1 -fa golub-cds.fa -
hgs golub-cds-hgs-nr.tab -pairs golub-exprcorr-pos.tab -start -10000
-end -1 -nmer 6 -window 200 -threshold 4 -o golub-transcripts-n-
pos.tab

# Do control comparison on transcripts using N
compare.exe -indexdir f:/lhon/hgs/index/ -method 1 -fa golub-cds.fa -
hgs golub-cds-hgs-nr.tab -pairs golub-exprcorr-zero.tab -n 34000 -
start -10000 -end -1 -nmer 6 -window 200 -threshold 4 -o golub-
transcripts-n-ctrl.tab

echo Function N on transcripts >> golub-summary.txt
./rocd golub-transcripts-n-pos.tab golub-transcripts-n-ctrl.tab -a >>
golub-summary.txt

# Do many random comparisons using F
compare.exe -indexdir f:/lhon/hgs/index/ -hgs golub-cds-hgs-nr.tab -n
100000 -start -10000 -end -1 -nmer 6 -window 200 -threshold 4 -o
golub-f-rand.tab

# Extract high and low scores from random comparisons
perl process-converse-results.pl golub-nm-expr.tab golub-f-rand.tab
4500 golub-conv-f-pos.tab golub-conv-f-ctrl.tab

echo Converse on Function F >> golub-summary.txt
./rocd golub-conv-f-pos.tab golub-conv-f-ctrl.tab -a >> golub-
summary.txt

# Get repeat masked upstream regions

```

UCSF LIBRARY

```

compare.exe -indexdir f:/lhon/hgs/rmindex/ -hgs golub-cds-hgs-nr.tab -
  start -10000 -end -1 -checkcomp 0 -getallseq -saveseqfile golub-
  rm.fa

# Calculating repeat masked separation on positive pairs
compare.exe -indexdir f:/lhon/hgs/index/ -fa golub-rm.fa -hgs golub-
  cds-hgs-nr.tab -pairs golub-exprcorr-pos.tab -start -10000 -end -1 -
  nmer 6 -window 200 -threshold 4 -o golub-rm-pos.tab -rc

# Calculating repeat masked separation on non-correlated pairs
compare.exe -indexdir f:/lhon/hgs/index/ -fa golub-rm.fa -hgs golub-
  cds-hgs-nr.tab -pairs golub-exprcorr-zero.tab -n 34000 -start -10000
  -end -1 -nmer 6 -window 200 -threshold 4 -o golub-rm-ctrl.tab -rc

echo Repeat masked using Function F >> golub-summary.txt
./rocd golub-rm-pos.tab golub-rm-ctrl.tab -a >> golub-summary.txt

# Calculating repeat masked separation on positive pairs
compare.exe -method 2 -indexdir f:/lhon/hgs/index/ -fa golub-rm.fa -hgs
  golub-cds-hgs-nr.tab -pairs golub-exprcorr-pos.tab -start -10000 -
  end -1 -nmer 6 -o golub-rm-np-pos.tab

# Calculating repeat masked separation on non-correlated pairs
compare.exe -method 2 -indexdir f:/lhon/hgs/index/ -fa golub-rm.fa -hgs
  golub-cds-hgs-nr.tab -pairs golub-exprcorr-zero.tab -n 34000 -start
  -10000 -end -1 -nmer 6 -o golub-rm-np-ctrl.tab

echo Repeat masked using Function N' >> golub-summary.txt
./rocd golub-rm-np-pos.tab golub-rm-np-ctrl.tab -a >> golub-
  summary.txt

# Get normal upstream regions
compare.exe -indexdir f:/lhon/hgs/index/ -hgs golub-cds-hgs-nr.tab -
  start -10000 -end -1 -checkcomp 0 -getallseq -saveseqfile golub-
  up.fa

# create blast db of the upstream regions of the NMs
formatdb -i golub-up.fa -p F -o T

# create script to run blast on each of the Alu consensus sequences
perl generate-alu-blastall.pl golub-up.fa golub-alus-blastall.bsh
bash -c "source golub-alus-blastall.bsh"

# Look at the blast output and extract the alu hits
perl gen-grab-seq-from-blast.pl golub-up.fa-Alu\*.out golub-alus-
  analyze.bsh
bash -c "source golub-alus-analyze.bsh"

# Consolidate hits that point to the same Alu
perl merge-alu-blast.pl golub-up.fa-Alu\*.tab golub-alus-all.tab

# Generate a masked upstream region with the Alus removed
perl generate-fa-by-nm-file.pl golub-up.fa golub-alus-all.tab golub-
  alu.fa
compare.exe -indexdir f:/lhon/hgs/index/ -hgs golub-cds-hgs-nr.tab -fa
  golub-alu.fa -pairs golub-exprcorr-pos.tab -start -10000 -end -1 -
  nmer 6 -window 200 -threshold 4 -o golub-alus-pos.tab -rc
compare.exe -indexdir f:/lhon/hgs/index/ -hgs golub-cds-hgs-nr.tab -fa
  golub-alu.fa -pairs golub-exprcorr-zero.tab -n 34000 -start -10000 -
  end -1 -nmer 6 -window 200 -threshold 4 -o golub-alus-ctrl.tab -rc

```


UCSF LIBRARY

echo Alus only using Function F >> golub-summary.txt

```
./rocd golub-alus-pos.tab golub-alus-ctrl.tab -a >> golub-summary.txt
compare.exe -indexdir f:/lhon/hgs/index/ -method 2 -hgs golub-cds-hgs-
nr.tab -fa golub-alu.fa -pairs golub-exprcorr-pos.tab -start -10000
-end -1 -nmer 6 -o golub-alus-np-pos.tab
compare.exe -indexdir f:/lhon/hgs/index/ -method 2 -hgs golub-cds-hgs-
nr.tab -fa golub-alu.fa -pairs golub-exprcorr-zero.tab -n 34000 -
start -10000 -end -1 -nmer 6 -o golub-alus-np-ctrl.tab
```

echo Alus only using Function N\ ' >> golub-summary.txt

```
./rocd golub-alus-np-pos.tab golub-alus-np-ctrl.tab -a >> golub-
summary.txt
```

Count number of Alus per NM

```
perl count-freq.pl golub-alus-all.tab golub-up.fa 0 golub-alus-nms-
count.tab
perl compare.pl golub-cds-hgs-nr.tab golub-alus-nms-count.tab golub-
exprcorr-pos.tab golub-alu-product-pos.tab
perl compare.pl golub-cds-hgs-nr.tab golub-alus-nms-count.tab golub-
exprcorr-zero.tab golub-alu-product-ctrl.tab
```

echo Alus product >> golub-summary.txt

```
./rocd golub-alu-product-pos.tab golub-alu-product-ctrl.tab -a >>
golub-summary.txt
```

Get non alus

```
perl inverse-mask.pl golub-up.fa golub-alu.fa > golub-notalus.fa
compare.exe -indexdir f:/lhon/hgs/index/ -fa golub-notalus.fa -hgs
golub-cds-hgs-nr.tab -pairs golub-exprcorr-pos.tab -start -10000 -
end -1 -nmer 6 -window 200 -threshold 4 -o golub-notalus-f-pos.tab -
rc
compare.exe -indexdir f:/lhon/hgs/index/ -fa golub-notalus.fa -hgs
golub-cds-hgs-nr.tab -pairs golub-exprcorr-zero.tab -n 34000 -start
-10000 -end -1 -nmer 6 -window 200 -threshold 4 -o golub-notalus-f-
ctrl.tab -rc
```

echo Not Alus using Function F >> golub-summary.txt

```
./rocd golub-notalus-f-pos.tab golub-notalus-f-ctrl.tab -a >> golub-
summary.txt
```

Get repeats only

```
perl inverse-mask.pl golub-up.fa golub-rm.fa > golub-ronly.fa
compare.exe -indexdir f:/lhon/hgs/index/ -fa golub-ronly.fa -hgs golub-
cds-hgs-nr.tab -pairs golub-exprcorr-pos.tab -start -10000 -end -1 -
nmer 6 -window 200 -threshold 4 -o golub-ronly-f-pos.tab -rc
compare.exe -indexdir f:/lhon/hgs/index/ -fa golub-ronly.fa -hgs golub-
cds-hgs-nr.tab -pairs golub-exprcorr-zero.tab -n 34000 -start -10000
-end -1 -nmer 6 -window 200 -threshold 4 -o golub-ronly-f-ctrl.tab -
rc
```

echo repeats only using Function F >> golub-summary.txt

```
./rocd golub-ronly-f-pos.tab golub-ronly-f-ctrl.tab -a >> golub-
summary.txt
```

Do the standard positive concordant comparison

```
compare.exe -indexdir f:/lhon/hgs/index/ -hgs golub-cds-hgs-nr.tab -
pairs golub-exprcorr-pos.tab -start -10000 -end -1 -nmer 6 -window
200 -threshold 4 -o golub-rc-f-pos.tab -rc
```

Do the standard control concordant comparison

```
compare.exe -indexdir f:/lhon/hgs/index/ -hgs golub-cds-hgs-nr.tab -  
  pairs golub-exprcorr-zero.tab -start -10000 -end -1 -nmer 6 -window  
  200 -threshold 4 -n 34000 -o golub-rc-f-ctrl.tab -rc
```

echo Function F w/ reverse complement >> golub-summary.txt

```
./rocd golub-rc-f-pos.tab golub-rc-f-ctrl.tab -a >> golub-summary.txt
```

UCSF LIBRARY

Appendix B MaMF Usage and Documentation

This appendix shows how to use MaMF (Appendix B.1) and documents the functions that make up MaMF (Appendix B.2). The pdf version of this document contains hyperlinks of important modes, scripts, and functions that jump to the relevant section detailing their use. They are marked by a dotted underline.

B.1. Sample MaMF Usage

B.1.1. *Sample Experiments*

```
#####
```

```
# MaMF permutation analysis
```

```
#
```

```
# Get a random set of sequences
```

```
mf.exe -mode randseqlist -width 11 -i crebatf.list -fa hspromoter-  
masked.fa -o crebatf-permtest0.list
```

```
# Run MaMF
```

```
mf.exe -mode mfseqlist -width 11 -nmersize 4 -distr allhom-human-up10k-  
order11.bgd -distr2 allhom-human-up10k-order11-mut1.bgd -seeds2 1000 -i  
crebatf-permtest0.list -fa hspromoter-masked.fa -o crebatf-  
permtest0.out -log crebatf-permtest0.log
```

```
# Searches input sequences to identify which binding sites are similar to annotated motif,  
to be used in the next step
```

```
mf.exe -mode findhighscoringpwm -i crebatf.sl -seqs crebatf-  
permtest0.list -fa hspromoter-masked.fa -o crebatf-pwmbest-permtest0.sl
```

UCSF LIBRARY

Analyze output

```
analyze-output.pl crebatf-permtest0.log crebatf-pwmbest-permtest0.sl  
crebatf-permtest0.html crebatf-permtest0.txt
```

... do all permutations ...

takes all analyzed output and generates summary statistics of permutation analysis
summarize-permtest.pl

#####

MaMF background models

MaMF can accept a variety of different background models, simply by using different parameters

#

-gcprob 0.3 uses a simple weighted scheme (a la Consensus) with GCs accounting for 30% of the genome

```
mf.exe -mode mfseqlist -width 11 -nmersize 4 -seeds 1000 -seeds2 1000 -  
fa hspromoter-masked.fa -gcprob 0.38 -i crebatf.list -o crebatf-  
width11-gcw.out
```

-bg ../background/allhom-order7.mbm -> 7th order Markov background model (a la Bioprosector, but higher order)

```
mf.exe -mode mfseqlist -width 11 -nmersize 4 -seeds 1000 -seeds2 1000 -  
fa hspromoter-masked.fa -bg allhom-order7.mbm -i crebatf.list -o  
crebatf-width11-bgallhom-order7-markov.out
```

used to generate the Markov background model

```
mf.exe -mode genmarkov -fa allhom-masked.fa -order 3 -o allhom-  
order3.mbm
```

-mt 2 -> combination method using the count method at higher probabilities, and Markov at lower frequencies

```
mf.exe -mode mfseqlist -width 11 -nmersize 4 -seeds 1000 -seeds2 1000 -  
fa hspromoter-masked.fa -bg dbtss-order5.mbm -distr dbtss-width11.bgd -  
mt 2 -i crebatf.list -o crebatf-width11-bgdbtss-order5-mt2-mc.out
```

-distr -> the count method

```
mf.exe -mode mfseqlist -width 11 -nmersize 4 -seeds 1000 -seeds2 1000 -  
fa hspromoter-masked.fa -distr dbtss-width11.bgd -i crebatf.list -o  
crebatf-width11-bgdbtss-count.out
```

-distr2 -> the count w/ mutations method, used in the paper

```
mf.exe -mode mfseqlist -width 11 -nmersize 4 -seeds 1000 -seeds2 1000 -  
fa hspromoter-masked.fa -distr2 dbtss-width11-mut1.bgd -i crebatf.list  
-o crebatf-width11-bgdbtss-cmut.out
```

-psb -> position specific background, using multiple Markov backgrounds, one for each region relative to the transcription start site

```
mf.exe -mode mfseqlist -width 11 -nmersize 4 -seeds 1000 -seeds2 1000 -  
fa hspromoter-masked.fa -psb dbtss-background-order3-bucket100.psb -i  
crebatf.list -o crebatf-width11-order3-psb.out
```

UCSF LIBRARY


```

#####
# Algorithm Comparison
#####
#

#####
# MaMF
#
# Create distribution based on counts of 11mers
mf.exe -mode gendistr -fa allhom-masked.fa -order 11 -o allhom-
width11.bgd -log allhom-width11.log

# Create distribution based on counts of 11mers and their mutations
mf.exe -mode genmutdistr -order 11 -width 11 -distr allhom-width11.bgd
-o allhom-width11-mut1.bgd -log allhom-width11-mut1.log

# Run MaMF
mf.exe -mode mfseqlist -width 11 -nmersize 4 -seeds 1000 -distr2
allhom-width11-mut1.bgd -i crebatf.list -fa hspromoter-masked.fa -o mf-
crebatf-bgallhom-w11.out

# Remove duplicate and highly similar motifs
mf.exe -mode clustersimilarmotifs -width 11 -i mf-crebatf-bgallhom-
w11.out -sl transfac-sites-all-mod.tab -fa hspromoter-masked.fa -o mf-
crebatf-bgallhom-w11-csm.out

# Get summary data on output
mf.exe -mode comparetruth -sl crebatf.sl -fa hspromoter-masked.fa -i
mf-crebatf-bgallhom-w11-csm.out -o mf-crebatf-bgallhom-w11.sum

# Get consensus alignments of motifs that look similar to annotated motif
mf.exe -mode linktotruth -sl crebatf.sl -fa hspromoter-masked.fa -i mf-
crebatf-bgallhom-w11-csm.out -o mf-crebatf-bgallhom-w11.align

# Use both BP and MaMF's scoring functions to score output motifs
mf.exe -mode comparemotifscores -width 11 -sl crebatf.sl -bg allhom-
order3.mbm -distr2 allhom-width11-mut1.bgd -fa hspromoter-masked.fa -i
mf-crebatf-bgallhom-w11-csm.out -o mf-crebatf-bgallhom-w11-compare.out

#####
# Bioprospector
#
# get rid of Ns in allhom data
fa2bp.pl allhom-masked.fa allhom-masked-nons.bp N

# create Bioprospector background model from allhom data
genomebg.exe -i allhom-masked-nons.bp -o bp-allhom.bpb

# get crebatf sequence
subset-fa.pl dbtss-masked-nons.bp crebatf.list 0 bp-crebatf.bp

```

Run Bioprospector

BioProspector.exe -o bp-crebatf-bgallhom-w11-all.out -W 11 -i bp-crebatf.bp -f bp-allhom.bpb -T 30

Process Bioprospector output

bpout2sl.pl bp-crebatf-bgallhom-w11-all.out bp-crebatf-bgallhom-w11.sl bp-crebatf.bp

Get summary data on output

mf.exe -mode comparetruth -sl crebatf.sl -fa dbtss-masked-nons.bp -i bp-crebatf-bgallhom-w11.sl -o bp-crebatf-bgallhom-w11.sum

Use both BP and MaMF's scoring functions to score output motifs

mf.exe -mode comparemotifscores -width 11 -sl crebatf.sl -bg allhom-order3.mbm -distr2 allhom-width11-mut1.bgd -fa hspromoter-masked.fa -i bp-crebatf-bgallhom-w11.sl -o bp-crebatf-bgallhom-w11-compare.out

#####

Consensus

#

mf.exe -mode getcon -i crebatf.list -fa hspromoter-masked.fa -o crebatf.con

Run Consensus

consensus-v6c.exe -f crebatf.con -L 11 -A "a:t 3 c:g 2 n:n 10000" -c1 -pf 1000 -n 10 -m 11 > con-crebatf-w11.out

Process Consensus output

conout2sl.pl con-crebatf-w11.out con-crebatf-w11.sl

Get summary data on output

mf.exe -mode comparetruth -sl crebatf.sl -fa hspromoter-masked.fa -i con-crebatf-w11.sl -o con-crebatf-w11.sum

#####

AlignACE experiment

#

Run AlignACE

AlignACE.exe -i bp-crebatf.bp > aa-crebatf-bgallhom-width11.out

Process AlignACE output

aa2sl.pl aa-crebatf-bgallhom-width11.out aa-crebatf-bgallhom-width11.sl

Get summary data on output

mf.exe -mode comparetruth -sl crebatf.sl -fa dbtss-masked-nons.bp -i aa-crebatf-bgallhom-width11.sl -o aa-crebatf-bgallhom-w11.sum

#####

Enrichment MF

#

UCSF LIBRARY

Run MaMF using enrichmentmf method

```
mf.exe -mode enrichmentmf -width 11 -nmersize 4 -seeds 1000 -fa
hspromoter-masked.fa -distr2 allhom-width11-mut1.bgd -i crebatf.list -
i2 crebatf.sl -sl transfac-sites-all-mod.tab -array stuart-dbtss-human-
subset10.tab -o mfer-crebatf-bgallhom-width11-cmut-enrich.out
```

```
mf.exe -mode clusterscoremotifs -width 11 -i mfer-crebatf-bgallhom-
width11-cmut-enrich.out -sl transfac-sites-all-mod.tab -fa hspromoter-
masked.fa -o mfer-crebatf-bgallhom-width11-cmut-enrich-csm.out
```

Get summary data on output

```
mf.exe -mode comparetruth -sl crebatf.sl -fa hspromoter-masked.fa -i
mfer-crebatf-bgallhom-width11-cmut-enrich-csm.out -o mfer-crebatf-
bgallhom-w11.sum
```

#####

ERMF

#

Run MaMF using ERMF method

```
mf.exe -mode ermfm -width 11 -nmersize 4 -seeds 1000 -ranking crebatf-
ordered.tab -fa hsfr-promoter-masked.fa -distr2 allhom-width11-mut1.bgd
-i crebatf.list -i2 correlation_table.txt -sl transfac-sites-all-
mod.tab -o mfercg-crebatf-bgallhom-width11-cmut-enrich-fugu.out
```

Rerank motifs based on composite enrichment ratio and remove duplicates

```
mf.exe -mode clusterscoremotifs -width 11 -i mfercg-crebatf-bgallhom-
width11-cmut-enrich-fugu.out -sl transfac-sites-all-mod.tab -fa
hspromoter-masked.fa -o mfercg-crebatf-bgallhom-width11-cmut-enrich-
fugu-csm.out
```

Get summary data on output

```
mf.exe -mode comparetruth -sl crebatf.sl -fa hspromoter-masked.fa -i
mfercg-crebatf-bgallhom-width11-cmut-enrich-fugu-csm.out -o mfercg-
crebatf-bgallhom-w11.sum
```

#####

PerturbRandomMotifs experiment

#

```
mf.exe -mode perturbrandommotifs -width 11 -i crebatf-ordered.tab -fa
hspromoter-masked.fa -sl transfac-sites-all-mod.tab -o prm-crebatf-
width11.out
```

Rerank motifs based on composite enrichment ratio and remove duplicates

```
mf.exe -mode clusterscoremotifs -width 11 -i prm-crebatf-width11.out -
sl transfac-sites-all-mod.tab -fa hspromoter-masked.fa -o prm-crebatf-
width11-cluster.out
```

Get summary data on output

```
mf.exe -mode comparetruth -sl crebatf.sl -fa hspromoter-masked.fa -i
prm-crebatf-width11-cluster.out -o prm-crebatf-w11.sum
```

UCSF LIBRARY

Get consensus alignments of motifs that look similar to annotated motif
 mf.exe -mode linktotruth -sl crebatf.sl -fa hspromoter-masked.fa -i
 prm-crebatf-width11-cluster.out -o prm-crebatf-w11.align

B.1.2. MaMF Mode Documentation

MaMF Mode Summary	
mf.exe -mode <u>clusterscoremotifs</u> [-width] [-i] [-sl] [-fa] [-o]	not used in paper, but essentially grouping common motifs and getting a composite enrichment ratio (due to noise) and using that as the score
mf.exe -mode <u>clustersimilarmotifs</u> [-width] [-i] [-sl] [-fa] [-o]	Removes highly similar motifs
mf.exe -mode <u>comparemotifscores</u> [-width] [-sl] [-bg] [-distr2] [-fa] [-i] [-o]	Compare motif scores using MaMF's scoring function and Bioprospector's scoring function
mf.exe -mode <u>comparetruth</u> [-sl] [-fa] [-i] [-o]	Summarizes comparison of true motif against output motifs
mf.exe -mode <u>enrichmentmf</u> [-width] [-nmersize] [-seeds] [-fa] [-distr2] [-i] [-i2] [-sl] [-array] [-o]	MaMF followed by a post processing step that uses Enrichment Ratios to rescore motifs
mf.exe -mode <u>ornmf</u> [-width] [-nmersize] [-seeds] [-ranking] [-fa] [-distr2] [-i] [-i2] [-sl] [-o]	similar to enrichmentmf but combines ChIP array data and expression microarray, and throws in some comparative genomics as well
mf.exe -mode <u>findhighscoringpwm</u> [-i] [-seqs] [-fa] [-o]	finds binding sites in input sequence similar to annotated motif
mf.exe -mode <u>gendistr</u> [-fa] [-order] [-o] [-log]	Creates a count background model for use with MaMF
mf.exe -mode <u>genmarkov</u> [-fa] [-order] [-o]	Creates a Markov background model for use with MaMF
mf.exe -mode <u>genmutdistr</u> [-order] [-width] [-distr] [-o] [-log]	Creates a count with mutations background model for use with MaMF
mf.exe -mode <u>getcon</u> [-i] [-fa] [-o]	Gets Consensus-style sequence
mf.exe -mode <u>linktotruth</u> [-sl] [-fa] [-i] [-o]	Displays and aligns consensus sequences of motifs similar to truth
mf.exe -mode <u>mfseqlist</u> [-width] [-nmersize] [-seeds] [-distr2] [-i] [-fa] [-o]	Runs MaMF using a series of accessions as input
mf.exe -mode <u>parturbrandommotifs</u> [-width] [-i] [-fa] [-sl] [-o]	New motif finding method, starting with a random motif and optimizing based on enrichment ratio
mf.exe -mode <u>randseqlist</u> [-width] [-i] [-fa] [-o]	Get a random set of sequences

UCSF LIBRARY

clusterscoremotifs

```
mf.exe -mode clusterscoremotifs [-width] [-i] [-sl] [-fa] [-o]
```

not used in paper, but essentially grouping common motifs and getting a composite enrichment ratio (due to noise) and using that as the score

Parameters:

-width - width of motif to be found
-i - input motifs
-sl - transfac motifs
-fa - fasta sequence to get surrounding sequence
-o - output

clustersimilarmotifs

```
mf.exe -mode clustersimilarmotifs [-width] [-i] [-sl] [-fa] [-o]
```

Removes highly similar motifs

See Methods in MaMF paper for details

Parameters:

-width - width of motifs in input
-i - input motifs
-sl - transfac motifs to allow annotation of resulting motifs
-fa - fasta sequence to allow execution of similarity metric for similar motifs
-o - reduced set of motifs

comparemotifscores

```
mf.exe -mode comparemotifscores [-width] [-sl] [-bg] [-distr2] [-fa] [-i] [-o]
```

Compare motif scores using MaMF's scoring function and Bioprospector's scoring function

Parameters:

UCSF LIBRARY

- width - width of motif to be found
- sl - annotated motif used as the "truth"
- bg - a Markov model based background distribution
- distr2 - background distribution
- fa - fasta sequence to get surrounding sequence
- i - input motifs
- o - scores

comparetruth

```
mf.exe -mode comparetruth [-sl] [-fa] [-i] [-o]
```

Summarizes comparison of true motif against output motifs

Parameters:

- sl - true motif
- fa - fasta sequence to get surrounding sequence
- i - motifs obtained from a motif finder
- o - summary output

enrichmentmf

```
mf.exe -mode enrichmentmf [-width] [-nmersize] [-seeds] [-fa] [-distr2] [-i] [-i2] [-sl] [-array] [-o]
```

MaMF followed by a post processing step that uses Enrichment Ratios to rescore motifs

Parameters:

- width - width of motif to be found
- nmersize - nmer size for indexing
- seeds - number of seeds to use for greedy search
- fa - fasta file containing actual sequence
- distr2 - background distribution
- i - accession numbers, one per line
- i2 - annotated motif shared in input
- sl - transfac motifs to diagnosis each motif
- array - microarray to calculate enrichment ratios
- o - output

UCSF LIBRARY

ermf

```
mf.exe -mode ermf [-width] [-nmersize] [-seeds] [-ranking] [-fa]
[-distr2] [-i] [-i2] [-s1] [-o]
```

similar to enrichmentmf but combines ChIP array data and expression microarray, and throws in some comparative genomics as well

for ChIP data, uses a ranking of related genes, equivalent to coexpression, but instead it's strength of binding

Parameters:

- width - width of motif to be found
- nmersize - nmer size for indexing
- seeds - number of seeds to be used in the greedy search
- ranking - ranking of genes as obtained from ChIP data (though possibly expression data too?)
- fa - fasta sequence to get surrounding sequence
- distr2 - background distribution
- i - input accession numbers
- i2 - correlation table of genes from one organism to another
- s1 - transfac motifs
- o - output motifs, scored using above data

findhighscoringpwm

```
mf.exe -mode findhighscoringpwm [-i] [-seqs] [-fa] [-o]
```

finds binding sites in input sequence similar to annotated motif

Parameters:

- i - annotated motif
- seqs - input sequence ids
- fa - fasta sequences
- o - output of similar binding sites

gendistr

```
mf.exe -mode gendistr [-fa] [-order] [-o] [-log]
```

UCSF LIBRARY

Creates a count background model for use with MaMF

Parameters:

- fa - input sequence
 - order - actually the width of the sequence to calculate a probability for
 - o - output background model
 - log - statistics
-

genmarkov

```
mf.exe -mode genmarkov [-fa] [-order] [-o]
```

Creates a Markov background model for use with MaMF

Parameters:

- fa - input sequence
 - order - nth order background model
 - o - output background model
-

genmutdistr

```
mf.exe -mode genmutdistr [-order] [-width] [-distr] [-o] [-log]
```

Creates a count with mutations background model for use with MaMF

Parameters:

- order - actually the width of the sequence to calculate a probability for
 - width - needs to be same as order
 - distr - output from a gendistr run
 - o - output background model
 - log - statistics
-

getcon

```
mf.exe -mode getcon [-i] [-fa] [-o]
```

Gets Consensus-style sequence

Parameters:

UCSF LIBRARY

- i - input accessions, one per line
- fa - fasta file containing sequence
- o - output Consensus-style sequence

linktotruth

```
mf.exe -mode linktotruth [-sl] [-fa] [-i] [-o]
```

Displays and aligns consensus sequences of motifs similar to truth

Parameters:

- sl - annotated motif (truth)
- fa - fasta sequence to get surrounding sequence
- i - input motifs
- o - alignment of the various similar motifs

mfseqlist

```
mf.exe -mode mfseqlist [-width] [-nmersize] [-seeds] [-distr2] [-i] [-fa] [-o]
```

Runs MaMF using a series of accessions as input

Parameters:

- width - width of motif to be found
- nmersize - nmer size for indexing, typically 4
- seeds - number of seeds to use for greedy search
- distr2 - background distribution
- i - accession numbers, one per line
- fa - fasta file containing actual sequence
- o - output motifs

perturbrandommotifs

```
mf.exe -mode perturbrandommotifs [-width] [-i] [-fa] [-sl] [-o]
```

New motif finding method, starting with a random motif and optimizing based on enrichment ratio

enrichment ratio is too noisy by itself

Parameters:

- width - width of motif to be found
- i - input accession numbers
- fa - fasta sequence to get surrounding sequence
- sl - transfac motifs
- o - output motifs

randseqlist

```
mf.exe -mode randseqlist [-width] [-i] [-fa] [-o]
```

Get a random set of sequences

Parameters:

- width - not used
- i - Number of ids to get
- fa - fasta sequences

B.1.3. Perl Script Documentation

Script Summary
aa2sl.pl [\$fin] [\$fout] convert AlignACE output to SiteList formatted output
analyze-output.pl [\$fin] [\$fpwmbest] [\$fout] [\$fsl] visualize MaMF output
bpout2sl.pl [\$fin] [\$fout] [\$ffa] converts Bioprospector output into the SiteList format
conout2sl.pl [\$fin] [\$fout] converts Consensus output into SiteList format
subset-fa.pl [\$fin] [\$list] [\$col] [\$fout] given a fasta file, outputs a new fasta file with a subset of the sequences

aa2sl.pl

```
aa2sl.pl [$fin] [$fout]
```

convert AlignACE output to SiteList formatted output

UCSF LIBRARY

Parameters:

`$fin` - AliceACE output

`$fout` - SiteList formatted output

analyze-output.pl

```
analyze-output.pl [$fin] [$fpwmbest] [$fout] [$fsl]
```

visualize MaMF output

Parameters:

`$fin` - input motifs from MaMF

`$fpwmbest` - Sequences in output considered to be highly similar to annotated motif

`$fout` - process html output

`$fsl` - annotated binding sites

bpout2sl.pl

```
bpout2sl.pl [$fin] [$fout] [$ffa]
```

converts Bioprospector output into the SiteList format

fasta file is required to calculate sequence lengths

Parameters:

`$fin` - bioprospector output

`$fout` - SiteList formatted output

`$ffa` - genomic fasta file

conout2sl.pl

```
conout2sl.pl [$fin] [$fout]
```

converts Consensus output into SiteList format

Parameters:

`$fin` - Consensus output

`$fout` - Sitelist formatted output

subset-fa.pl

```
subset-fa.pl [$fin] [$list] [$col] [$fout]
```

given a fasta file, outputs a new fasta file with a subset of the sequences

Parameters:

`$fin` - input fasta file

`$list` - list of desired sequences

`$col` - column that gene ids are found in the list (if in a tab delimited file)

`$fout` - output fasta file

B.2. Documentation of C Files

`motif_finder.c` is the entry point into MaMF. The meat of the algorithm is found in `score.c` and `search.c`, which implement the scoring function and search algorithms, respectively. Most of the experiments I performed to analyze the results of MaMF, explore the enrichment ratio concept, and other miscellaneous experiments are found in `analysis.c`.

<code>analysis.c</code>	this is the dumpyard of all my experiments.
<code>ba2d.c</code>	binary 2d array
<code>background.c</code>	utility functions to calculate background probabilities
<code>dadbl.c</code>	pseudo-class of a dynamic array of doubles
<code>daint.c</code>	pseudo-class of a dynamic array of ints
<code>darrv.c</code>	dynamic array of void*
<code>dasl.c</code>	pseudo-class of a dynamic array of SiteLists
<code>dastr.c</code>	pseudo-class of dynamic array of strings
<code>inthash.c</code>	hash table using int as the key
<code>matchlist.c</code>	pseudo-class to handle pairs of sequence locations in a space efficient manner
<code>microarray.c</code>	pseudo-class to handle microarray data
<code>motif-finder.c</code>	mainly to coordinate what is to be done in this run via arguments from command line

UCSF LIBRARY

<u>oldhash.c</u>	hash table library
<u>pwm.c</u>	pseudo-class for a position weight matrix (PWM) object
<u>score.c</u>	functions to calculate scores of motifs
<u>search.c</u>	greedy motif finding using indexing to accelerate the search process
<u>sequence.c</u>	utility functions to handle the Sequence object
<u>sitelist.c</u>	pseudo-class of a list of binding sites
<u>utils.c</u>	library of utility functions

B.2.1. *analysis.c*

this is the dumpyard of all my experiments.

The Enrichment Ratio code, optimizing motif using ER, and other experiments are all here

Function Summary	
void	<u>AllSingleGeneER</u> (HashTable* ma, dasl* transfac) Compute enrichment ratios for all genes in gSeqList using the different transfac motifs
void	<u>ApplyBackgroundProb</u> (SiteList* sl, int width) calculates the background probability of each SiteWin using CalculateFrequency
double	<u>CalcGCCContent</u> (dastr* list) returns average GC content of a set of genes
void	<u>CalcGCEnrichMotifRatio</u> (dasl* result, dastr* geneList, HashTable* ma) plot GC content of geneList, proxyList, nullList, and average of results
double	<u>CalcMotifGCCContent</u> (SiteList* sl) returns the GC content percentage of the motif passed in
void	<u>ChipPlotMotifRatio</u> (dastr* orderedGeneList, int width) Uses PlotRatio to calculate ER of all subseqs of genes in geneList, using chip data (an ordered list of genes)
double	<u>ClusterOptimizeMotif</u> (PWM** motif, dasl* result) failed idea, optimizes a given motif by maximizing the enrichment ratio?
void	<u>ClusterScoreMotifs</u> (dasl* result, int width, dasl* transfac) Removes highly similar motifs from results and prints them
dasl*	<u>ClusterSimilarMotifs</u> (dasl* result, int width) This merges motifs that are highly similar to each other
void	<u>ClusterWithTransfac</u> (dasl* result, int width, dasl* transfac) a method of judging significance of motifs similar to transfac motifs found in results

void	<u>ClusterWithTransfac2</u> (dasl* result, int width, dasl* transfac, dastr* geneList, char* microarray) a method of judging significance of motifs similar to transfac motifs found in results
void	<u>ClusterWithTransfac3</u> (dasl* result, int width, dasl* transfac, dastr* geneList, dastr* orderedGeneList) a method of judging significance of motifs similar to transfac motifs found in results
int	<u>CompareDbList</u> (const void* a, const void* b) compares list of doubles
int	<u>CompareDbListHighFirst</u> (const void* a, const void* b) compares doubles in first positions of array
int	<u>CompareDouble1</u> (const void* a, const void* b) compare doubles, lower first
int	<u>CompareDouble1R</u> (const void* a, const void* b) compare doubles, higher first
PWM*	<u>ComputeExtendedPWM</u> (SiteList* sl, int len) extends a PWM both directions using real sequence data
void	<u>Experiment050310</u> (dastr* geneList, SiteList* rankingsl, int width, dasl* (*searchfn) (dastr* seqList, int width, double (*ScoringMetric) (SiteList*, int)), double (*scorefn) (SiteList* list, int width), char* array) don't think it worked
double**	<u>GenerateGCEnrichmentFactor</u> (dastr* orderedGeneList, int width) calculates the relationship between GC content and enrichment ratio?
void	<u>GenerateMotifsER</u> (dastr* orderedGeneList, int width) generate enrichment ratios from purely randomly generated motifs
void	<u>GenerateMotifsERDistribution</u> (dastr* orderedGeneList, int width) generated enrichment ratios of random motifs with a specific range of GC content
void	<u>GetAverageGeneRank</u> (HashTable* ma) using random geneLists, get the average rank of gene in resulting genesByCorrelation
SiteList*	<u>GetBestSiteWins</u> (dasl* sll) keep the best scoring SiteWin from each SiteList
PWM*	<u>GetGCRandomMotif</u> (int i, int numIter, int width) generates a motif with specific GC content
PWM*	<u>GetGCRandomMotif2</u> (int num, int width) this version guarantees a level of gcs
double	<u>GetMotifStrengthRatio</u> (dastr* goodList, dastr* badList, PWM* motif) looks like the predecessor to the enrichment ratio?
int	<u>GetNumSubseqAlignment</u> (char* subseq, Sequence* seq, int width) Gets the number of alignments of subseq within seq
dasl*	<u>GetTransfacMotifs</u> (dasl* transfac, char* s) Gets the SiteList of a particular Transfac motif
void	<u>JiggleMotif</u> (PWM* motif, int width, double* score, dasl* slla,

	<p>dasl* s1b, int subset1, int subset2)</p> <p>randomly jiggles motif but only keeps change if new motif is better</p>
void	<p><u>LinkResultsToTransfac</u>(dasl* all, dasl* transfac)</p> <p>plots results with similarity to motifs in the transfac list</p>
void	<p><u>LinkResultsToTransfac3</u>(dasl* all, dasl* transfac, double threshold)</p> <p>plots similarity of results to motifs in the transfac list</p>
void	<p><u>LinkResultsToTruth</u>(dasl* all, SiteList* truth, double threshold)</p> <p>Prints consensus of motifs that are highly similar to the true SiteList</p>
SiteList*	<p><u>OptimizeMotif</u>(PWM* motif, int width, dastr* a, dastr* b)</p> <p>randomly perturbs motif to maximize enrichment ratio</p>
void	<p><u>PerturbRandomMotifs</u>(dastr* orderedGeneList, int numMotifs, int width, dasl* transfac)</p> <p>algorithm that starts with random motif and optimizes based on maximizing enrichment ratio</p>
void	<p><u>PlotMotifRatio</u>(dastr* geneList, int width, HashTable* ma)</p> <p>Uses PlotRatio to calculate ER of all subseqs of genes in geneList, using microarray data</p>
void	<p><u>PlotMotifRatioTrend</u>(dasl* result, dastr* orderedGeneList, int width)</p> <p>plots the scores the best alignments of genes against some of the results, ordered by gene coexpression</p>
void	<p><u>PlotNumberNs</u>(dastr* geneList, HashTable* ma)</p> <p>Prints the number of Ns for each sequence in the microarray, ordered by coexpression</p>
void	<p><u>PlotRatio</u>(dastr* geneList, dastr* proxyList, dastr* nullList, int width)</p> <p>plots ER of all subseqs of genes in geneList</p>
void	<p><u>PlotSecondaryMotifs</u>(dasl* result, dastr* geneList, HashTable* ma)</p> <p>plot the relative positions of the best alignments of two genes for a given motif, and its enrichment ratio</p>
void	<p><u>RankGenesWithClusteredMotifs</u>(dasl* result, int width, dasl* transfac, char* tf, SiteList* ordered)</p> <p>Reuse clustering from above, and see if top 50 motifs can be used to more accurately predict</p>
SiteList*	<p><u>ReadTransfacData</u>(char* f)</p> <p>reads transfac data for one TF</p>
dasl*	<p><u>ReadTransfacMatrixData</u>(char* f)</p> <p>creates an array of SiteLists from transfac data</p>
void	<p><u>SeqFreq</u>(char* s)</p> <p>frequency of s within all sequence in gSeqs</p>
void	<p><u>SeqFreqER</u>(dastr* orderedGeneList, int width)</p> <p>computes frequency of all nmers of width for all genes in proxyList and nullList</p>
int	<p><u>SeqFreqSubset</u>(char* s, dastr* seqList)</p> <p>counts the number of times s occurs in all promoters of seqList</p>

UCSF LIBRARY

void	ShiftPerturbMotifs (dasl* input, dastr* orderedGeneList, int numMotifs, int width, dasl* transfac) takes input motifs, calculates shifted motifs, and optimizes those motifs
void	SimpleJiggleMotif (PWM* motif) randomly jiggles motif without optimization in mind, unlike JiggleMotif
void	SingleGeneER (char* id, HashTable* ma, dasl* transfac) Take a single gene and compute enrichment ratios for all transfac motifs against that gene
void	SingleGeneMF (char* id, int width, dasl* (*searchfn) (dastr* seqList, int width, double (*ScoringMetric)(SiteList*, int)), double (*scorefn) (SiteList* list, int width), HashTable* ma, dasl* transfac) experimental method to run MaMF on one gene
void	SuperClusterMotifs (dasl* result, int width, dasl* transfac) cluster motifs together, ranking by highest average ER
void	TransfacClusterScoreMotifs (dasl* result, int width, dasl* transfac) Computes a score for each transfac motif based on how common that motif is found in the results

AllSingleGeneER

void **AllSingleGeneER**(HashTable* ma, dasl* transfac)

Compute enrichment ratios for all genes in gSeqList using the different transfac motifs

Parameters:

ma - microarray expression data

transfac - transfac motifs

ApplyBackgroundProb

void **ApplyBackgroundProb**(SiteList* sl, int width)

calculates the background probability of each SiteWin using CalculateFrequency

Parameters:

sl - input SiteList

width - motif width

CalcGCCContent

double **CalcGCCContent**(dastr* list)

returns average GC content of a set of genes

Parameters:

list - list of gene names

CalcGCEnrichMotifRatio

```
void CalcGCEnrichMotifRatio(dasl* result, dastr* geneList, HashTable*  
ma)
```

plot GC content of geneList, proxyList, nullList, and average of results

Parameters:

result - motifs output from motif finder

geneList - genes of interest

ma - microarray data

CalcMotifGCContent

```
double CalcMotifGCContent(SiteList* sl)
```

returns the GC content percentage of the motif passed in

Parameters:

sl - SiteList of interest

ChipPlotMotifRatio

```
void ChipPlotMotifRatio(dastr* orderedGeneList, int width)
```

Uses PlotRatio to calculate ER of all subseqs of genes in geneList, using chip data (an ordered list of genes)

Parameters:

orderedGeneList - ordered list of genes calculated from coexpression using microarray data

width - motif width

ClusterOptimizeMotif

```
double ClusterOptimizeMotif(PWM** motif, dasl* result)
```

failed idea, optimizes a given motif by maximizing the enrichment ratio?

assumption: each result has extended motif as well as ER in score

Parameters:

result - motifs output from motif finder

ClusterScoreMotifs

```
void ClusterScoreMotifs(dasl* result, int width, dasl* transfac)
```

Removes highly similar motifs from results and prints them

This function does the scoring and sorting, but then the merging happens in ClusterSimilarMotifs.

The sorting uses a composite score where for a given motif, we take all motifs similar at a given threshold and take the nth highest (based on minMotifs) motif score to be used as the composite score. This is useful for the noisiness of calculating enrichment ratios

Parameters:

result - motifs output from motif finder

width - motif width

transfac - transfac motifs

ClusterSimilarMotifs

```
dasl* ClusterSimilarMotifs(dasl* result, int width)
```

This merges motifs that are highly similar to each other

the order of result matters, because the clustering starts from the top of the list. Once it's been processed, it's not looked at again

Parameters:

result - motifs output from motif finder

width - motif width

UCSF LIBRARY

ClusterWithTransfac

void **ClusterWithTransfac**(dasl* result, int width, dasl* transfac)

a method of judging significance of motifs similar to transfac motifs found in results

Parameters:

result - motifs output from motif finder

width - motif width

transfac - transfac motifs

ClusterWithTransfac2

void **ClusterWithTransfac2**(dasl* result, int width, dasl* transfac, dastr* geneList, char* microarray)

a method of judging significance of motifs similar to transfac motifs found in results

this version will randomize the microarray instead of the motifs

Parameters:

result - motifs output from motif finder

width - motif width

transfac - transfac motifs

geneList - genes used in motif finding (not explicitly reported in result)

microarray - filename of microarray

ClusterWithTransfac3

void **ClusterWithTransfac3**(dasl* result, int width, dasl* transfac, dastr* geneList, dastr* orderedGeneList)

a method of judging significance of motifs similar to transfac motifs found in results

this version will randomize the gene rankings instead of the motifs

Parameters:

result - motifs output from motif finder

width - motif width

transfac - transfac motifs

geneList - genes used in motif finding (not explicitly reported in result)

orderedGeneList - genes ordered by coexpression

UCSF LIBRARY

CompareDbList

int **CompareDbList**(const void* a, const void* b)

compares list of doubles

CompareDbListHighFirst

int **CompareDbListHighFirst**(const void* a, const void* b)

compares doubles in first positions of array

Parameters:

a - first array
b - second array

CompareDouble1

int **CompareDouble1**(const void* a, const void* b)

compare doubles, lower first

Parameters:

a - first double
b - second double

CompareDouble1R

int **CompareDouble1R**(const void* a, const void* b)

compare doubles, higher first

Parameters:

a - first double
b - ssecond double

ComputeExtendedPWM

PWM* **ComputeExtendedPWM**(SiteList* sl, int len)

extends a PWM both directions using real sequence data

UCSF LIBRARY

Parameters:

s1 - SiteList
len - amount to extend by each direction

Experiment050310

```
void Experiment050310(dastr* geneList, SiteList* rankingsl, int width,  
dasl* (*searchfn) (dastr* seqList, int width, double  
(*ScoringMetric)(SiteList*, int)), double (*scorefn) (SiteList* list,  
int width), char* array)
```

don't think it worked

Ajay's experiment

Parameters:

geneList - genes of interest
width - motif width
width - motif width

GenerateGCEnrichmentFactor

```
double** GenerateGCEnrichmentFactor(dastr* orderedGeneList, int width)
```

calculates the relationship between GC content and enrichment ratio?

Parameters:

orderedGeneList - ordered list of genes calculated from coexpression using microarray data
width - motif width

GenerateMotifsER

```
void GenerateMotifsER(dastr* orderedGeneList, int width)
```

generate enrichment ratios from purely randomly generated motifs

potential for biased motifs wrt GC content, so therefore GenerateMotifsERDistribution

Parameters:

UCSF LIBRARY

orderedGeneList - ordered list of genes calculated from coexpression using microarray data
width - motif width

GenerateMotifsERDistribution

void **GenerateMotifsERDistribution**(dastr* orderedGeneList, int width)

generated enrichment ratios of random motifs with a specific range of GC content

Parameters:

orderedGeneList - ordered list of genes calculated from coexpression using microarray data
width - motif width

GetAverageGeneRank

void **GetAverageGeneRank**(HashTable* ma)

using random geneLists, get the average rank of gene in resulting genesByCorrelation

Parameters:

ma - microarray data

GetBestSiteWins

SiteList* **GetBestSiteWins**(dasl* sll)

keep the best scoring SiteWin from each SiteList

Parameters:

sll - a list of SiteLists of interesting as input

GetGCRandomMotif

PWM* **GetGCRandomMotif**(int i, int numIter, int width)

generates a motif with specific GC content

GC% = $i/\text{numIter}$

UCSF LIBRARY

Parameters:

i - amount of GC
numIter - out of total
width - motif width

GetGCRandomMotif2

PWM* **GetGCRandomMotif2**(int num, int width)

this version guarantees a level of gcs

Parameters:

width - motif width

GetMotifStrengthRatio

double **GetMotifStrengthRatio**(dastr* goodList, dastr* badList, PWM* motif)

looks like the predecessor to the enrichment ratio?

GetNumSubseqAlignment

int **GetNumSubseqAlignment**(char* subseq, Sequence* seq, int width)

Gets the number of alignments of subseq within seq

Parameters:

subseq - short sequence
seq - sequence analyzed
width - subseq width

GetTransfacMotifs

daSl* **GetTransfacMotifs**(daSl* transfac, char* s)

Gets the SiteList of a particular Transfac motif

Parameters:

UCSF LIBRARY

transfac - transfac motifs
s - string name of the transfac motif of interest

JiggleMotif

```
void JiggleMotif(PWM* motif, int width, double* score, dasl* slla,  
dasl* sllb, int subset1, int subset2)
```

randomly jiggles motif but only keeps change if new motif is better

where better is a higher resulting enrichment ratio, the data of which is kept in slla and sllb

GetAverageMax gets the averages for the coexpressed and noncoexpressed genes

Parameters:

motif - input motif
width - motif width
score - current score of motif
slla - Short sequences from the coexpressed genes
sllb - Short sequences from the noncoexpressed genes
subset1 - number of top sequences to consider in slla
subset2 - number of top sequences to consider in sllb

LinkResultsToTransfac

```
void LinkResultsToTransfac(dasl* all, dasl* transfac)
```

plots results with similarity to motifs in the transfac list

Parameters:

all - all results
transfac - transfac motifs

LinkResultsToTransfac3

```
void LinkResultsToTransfac3(dasl* all, dasl* transfac, double  
threshold)
```

plots similarity of results to motifs in the transfac list

similar to LinkResultsToTruth but compares against all transfac motifs

UCSF LIBRARY

Parameters:

all - all the results
transfac - transfac motifs to be compared against
threshold - similarity threshold to be considered similar

LinkResultsToTruth

void **LinkResultsToTruth**(dasl* all, SiteList* truth, double threshold)

Prints consensus of motifs that are highly similar to the true SiteList

Variation of the LinkResults* functions

Parameters:

all - all the results (from the motif finder)
truth - to be compared against
threshold - the similarity threshold of calling a motif a hit

OptimizeMotif

SiteList* **OptimizeMotif**(PWM* motif, int width, dastr* a, dastr* b)

randomly perturbs motif to maximize enrichment ratio

first all the short sequences are generated from a and b, so that they are cached for use with enrichment ratio, and ranked by similarity to the motif

JiggleMotif uses the first 100 highly ranked sequences from a and b; if a jiggled motif yields a higher ER than the original motif, then that new motif is kept

Occasionally, we rescore all the sequences so that the rank more closely matches the similarity to the motif; but I do it only sometimes for a performance enhancement

Parameters:

width - motif width
a - positive sequences for computation of enrichment ratio
b - negative sequences for computation of enrichment ratio

PerturbRandomMotifs

void **PerturbRandomMotifs**(dastr* orderedGeneList, int numMotifs, int width, dasl* transfac)

UCSF LIBRARY

algorithm that starts with random motif and optimizes based on maximizing enrichment ratio

the heavy lifting is in `OptimizeMotif`

Parameters:

`orderedGeneList` - ordered list of genes calculated from coexpression using microarray data

`numMotifs` - number of motifs to generate

`width` - motif width

`transfac` - transfac motifs

PlotMotifRatio

```
void PlotMotifRatio(dastr* geneList, int width, HashTable* ma)
```

Uses `PlotRatio` to calculate ER of all subseqs of genes in `geneList`, using microarray data

Parameters:

`geneList` - genes of interest

`width` - motif width

`ma` - microarray data

PlotMotifRatioTrend

```
void PlotMotifRatioTrend(dasl* result, dastr* orderedGeneList, int width)
```

plots the scores the best alignments of genes against some of the results, ordered by gene coexpression

probably to examine the behavior of the components of ER

Parameters:

`result` - motifs output from motif finder

`orderedGeneList` - ordered list of genes calculated from coexpression using microarray data

`width` - motif width

PlotNumberNs

```
void PlotNumberNs(dastr* geneList, HashTable* ma)
```

UCSF LIBRARY

Prints the number of Ns for each sequence in the microarray, ordered by coexpression

Parameters:

geneList - list of genes of interest

ma - microarray data

PlotRatio

```
void PlotRatio(dastr* geneList, dastr* proxyList, dastr* nullList, int width)
```

plots ER of all subseqs of genes in geneList

Parameters:

geneList - gene of interest

proxyList - genes with high correlation

nullList - genes with zero correlation

width - motif width

PlotSecondaryMotifs

```
void PlotSecondaryMotifs(dasl* result, dastr* geneList, HashTable* ma)
```

plot the relative positions of the best alignments of two genes for a given motif, and its enrichment ratio

Parameters:

result - motifs output from motif finder

geneList - genes of interest

ma - microarray data

RankGenesWithClusteredMotifs

```
void RankGenesWithClusteredMotifs(dasl* result, int width, dasl* transfac, char* tf, SiteList* ordered)
```

Reuse clustering from above, and see if top 50 motifs can be used to more accurately predict

Parameters:

UCSF LIBRARY

result - motifs output from motif finder
width - motif width
transfac - transfac motifs
tf - TF to be tested

ReadTransfacData

SiteList* **ReadTransfacData**(char* f)

reads transfac data for one TF

Parameters:

f - filename

ReadTransfacMatrixData

dasl* **ReadTransfacMatrixData**(char* f)

creates an array of SiteLists from transfac data

Parameters:

f - filename

SeqFreq

void **SeqFreq**(char* s)

frequency of s within all sequence in gSeqs

Parameters:

s - short sequence in question

SeqFreqER

void **SeqFreqER**(dastr* orderedGeneList, int width)

computes frequency of all nmers of width for all genes in proxyList and nullList

Parameters:

UCSF LIBRARY

orderedGeneList - ordered list of genes calculated from coexpression using microarray data

width - motif width

SeqFreqSubset

```
int SeqFreqSubset(char* s, dastr* seqList)
```

counts the number of times s occurs in all promoters of seqList

Parameters:

s - short sequence in question

seqList - list of genes to do computation

ShiftPerturbMotifs

```
void ShiftPerturbMotifs(dasl* input, dastr* orderedGeneList, int numMotifs, int width, dasl* transfac)
```

takes input motifs, calculates shifted motifs, and optimizes those motifs

uses [OptimizeMotif](#)

Parameters:

input - a set of different motifs

orderedGeneList - ordered list of genes calculated from coexpression using microarray data

numMotifs - not used

width - motif width

transfac - transfac motifs

SimpleJiggleMotif

```
void SimpleJiggleMotif(PWM* motif)
```

randomly jiggles motif without optimization in mind, unlike JiggleMotif

Parameters:

motif - input motif

UCSF LIBRARY

SingleGeneER

```
void SingleGeneER(char* id, HashTable* ma, dasl* transfac)
```

Take a single gene and compute enrichment ratios for all transfac motifs against that gene

Parameters:

id - gene id
ma - microarray expression data
transfac - transfac motifstransfac data

SingleGeneMF

```
void SingleGeneMF(char* id, int width, dasl* (*searchfn) (dastr*  
seqList, int width, double (*ScoringMetric)(SiteList*, int)), double  
(*scorefn) (SiteList* list, int width), HashTable* ma, dasl* transfac)
```

experimental method to run MaMF on one gene

get coexpressed genes using microarray, then run MaMF on these genes

Parameters:

id - gene id
width - width of motifs to be found
searchfn - search function pointer
scorefn - scoring function pointer
ma - microarray data
transfac - transfac motifstransfac data

SuperClusterMotifs

```
void SuperClusterMotifs(dasl* result, int width, dasl* transfac)
```

cluster motifs together, ranking by highest average ER

Parameters:

result - motifs output from motif finder
width - motif width
transfac - transfac motifs

UCSF LIBRARY

TransfacClusterScoreMotifs

```
void TransfacClusterScoreMotifs (das1* result, int width, das1*  
transfac)
```

Computes a score for each transfac motif based on how common that motif is found in the results

Parameters:

result - resulting motifs from MaMF

width - motif width

transfac - transfac motifs

B.2.2. *ba2d.c*

binary 2d array

space efficient implementation of a binary 2d array

Function Summary	
void	ba2d_Free (ba2d* arr) free the ba2d array
int	ba2d_IsBitSet (ba2d* arr, int x, int y) test if the bit is set
ba2d*	ba2d_New (int xdim, int ydim) creates a new ba2d array
int	ba2d_SetBit (ba2d* arr, int x, int y) set a bit in the array

ba2d_Free

```
void ba2d_Free (ba2d* arr)
```

free the ba2d array

Parameters:

arr - array to be operated on

ba2d_IsBitSet

```
int ba2d_IsBitSet (ba2d* arr, int x, int y)
```

UCSF LIBRARY

test if the bit is set

Parameters:

arr - array to be operated on
x - x dimension
y - y dimension

ba2d_New

ba2d* **ba2d_New**(int xdim, int ydim)

creates a new ba2d array

Parameters:

xdim - max x dimension
ydim - max y dimension

ba2d_SetBit

int **ba2d_SetBit**(ba2d* arr, int x, int y)

set a bit in the array

Parameters:

arr - array to be operated on
x - x dimension
y - y dimension

B.2.3. background.c

utility functions to calculate background probabilities

also embeds the pseudo-class distr, which uses actual frequency of sequences from the genome

Function Summary	
double	CalcAverageProbability (Distr* d, char* seq, int len) calculates average probability of sequences similar to seq
int	CalcMostSimilar (char* s, char* id) finds the best alignment to s and return the number of shared nucleotides
double	CalculateFrequency (char* seq, int len)

UCSF LIBRARY

	calculate probability of seq depending on background model
double	<u>ComputeEnrichmentRatio</u> (char* s, HashTable* erHash, dastr* posList, dastr* negList) computes enrichment ratio of a sequence (not motif)
double	<u>ComputeMDScanProbability</u> (HashTable* markov, SiteWin* sw, int len) calculates markov probability of sequence using method described in MDScan
double	<u>ComputeMarkovCount</u> (HashTable* markov, SiteWin* sw, int len) computes expected frequency (count) from markov probability
double	<u>ComputeMarkovProbability</u> (HashTable* markov, char* seq, int len) computes my version of the markov probability
double	<u>ComputePositionSpecificMarkovCount</u> (HashTable** markov, SiteWin* sw, int len) computes expected frequency (count) from position specific markov probability
double	<u>ComputePositionSpecificMarkovProbability</u> (HashTable** markov, SiteWin* sw, int len) computes position specific markov probability
double	<u>ComputeSimpleProb</u> (double gcProb, SiteWin* sw, int len) calculate a simple probability using just the GC content of the genome
Distr*	<u>CountBackgroundDistribution</u> (HashTable* seqs, int nmersize, FILE* fout) creates the counting method background distribution
void	<u>CountBackgroundWithMutations</u> (Distr* bg, int nmersize, FILE* fout) creates a secondary distribution considering similar sequences
HashTable*	<u>GenerateMarkovDependency</u> (HashTable* seqs, int order) generate markov background model
void	<u>GeneratePositionSpecificMarkovBackground</u> (int order, int bucketSize) create markov background models for different regions of sequence
double	<u>GetBackgroundProbability</u> (Distr* d, char* seq, int len, HashTable* markov) calculate background probability using markov background model
double	<u>GetBackgroundProbability2</u> (Distr* d, char* seq, int len) calculate background probability using distr (count method)
double	<u>GetCountFrequency</u> (Distr* d, char* seq, int len) get frequency of seq (in genome)
char*	<u>GetSubSequence2</u> (char* id, int posStart, int posEnd, int strand) same as GetSubSequence
HashTable**	<u>LoadPositionSpecificMarkovBackground</u> (char* filename) loads a position specific markov background
void	<u>distr_Free</u> (Distr* d) free object
unsigned int	<u>distr_Get</u> (Distr* d, int idx)

UCSF LIBRARY

	gets value of index
void	distr_Incr (Distr* d, int idx) increments count of that particular index
Distr*	distr_Load (char* filename) load distribution data from a file
void	distr_Save (Distr* d, FILE* fout) save distribution data to a file
void	distr_Set (Distr* d, int idx, unsigned int val) sets index to a particular value

CalcAverageProbability

double **CalcAverageProbability**(Distr* d, char* seq, int len)

calculates average probability of sequences similar to seq

Parameters:

d - distr object
seq - input sequence
len - length of sequence

CalcMostSimilar

int **CalcMostSimilar**(char* s, char* id)

finds the best alignment to s and return the number of shared nucleotides

Parameters:

s - input sequence
id - gene id

CalculateFrequency

double **CalculateFrequency**(char* seq, int len)

calculate probability of seq depending on background model

Parameters:

seq - input sequence
len - length of input sequence

UCSF LIBRARY

ComputeEnrichmentRatio

double **ComputeEnrichmentRatio**(char* s, HashTable* erHash, dastr* posList, dastr* negList)

computes enrichment ratio of a sequence (not motif)

assume all sequences exist

Parameters:

s - input sequence

erHash - hash of enrichment ratios?

posList - proxy list of sequences

negList - null list of sequences

ComputeMDScanProbability

double **ComputeMDScanProbability**(HashTable* markov, SiteWin* sw, int len)

calculates markov probability of sequence using method described in MDScan

it uses the prior probabilities of the preceding sequence before the input sequence for the first couple of probabilities

Parameters:

markov - stored markov probabilities

sw - input sequence

len - length of sequence

ComputeMarkovCount

double **ComputeMarkovCount**(HashTable* markov, SiteWin* sw, int len)

computes expected frequency (count) from markov probability

similar to ComputeMarkovProbability

Parameters:

markov - markov background model

sw - input sequence

len - length of input sequence

UCSF LIBRARY

ComputeMarkovProbability

double **ComputeMarkovProbability**(HashTable* markov, char* seq, int len)

computes my version of the markov probability

for ACGT, $P(A) \cdot P(C|A) \cdot P(G|AC) \cdot P(T|ACG)$

Parameters:

markov - stored markov probabilities

seq - input sequence

len - length of sequence

ComputePositionSpecificMarkovCount

double **ComputePositionSpecificMarkovCount**(HashTable** markov, SiteWin* sw, int len)

computes expected frequency (count) from position specific markov probability

similar to ComputePositionSpecificMarkovProbability

Parameters:

len - length of input sequence

ComputePositionSpecificMarkovProbability

double **ComputePositionSpecificMarkovProbability**(HashTable** markov, SiteWin* sw, int len)

computes position specific markov probability

Parameters:

markov - a position specific markov model

sw - input sequence

len - length of input sequence

ComputeSimpleProb

double **ComputeSimpleProb**(double gcProb, SiteWin* sw, int len)

calculate a simple probability using just the GC content of the genome

Parameters:

gcProb - gc genome background
sw - input sequence
len - length of input sequence

CountBackgroundDistribution

Distr* **CountBackgroundDistribution**(HashTable* seqs, int nmersize, FILE* fout)

creates the counting method background distribution

Parameters:

seqs - sequences to be considered
nmersize - nmer size of the distribution
fout - file pointer of where to save data

CountBackgroundWithMutations

void **CountBackgroundWithMutations**(Distr* bg, int nmersize, FILE* fout)

creates a secondary distribution considering similar sequences

for each sequence, obtain all other sequence resulting from one mutation, and add up frequencies of all these sequences

this reduces the inaccuracy in the case that a particular sequence is infrequent

Parameters:

bg - input distribution
nmersize - size of nmers in distribution
fout - file pointer to where data will be saved

GenerateMarkovDependency

HashTable* **GenerateMarkovDependency**(HashTable* seqs, int order)

generate markov background model

Parameters:

UCSF LIBRARY

seqs - input sequence
order - specify nth order model

GeneratePositionSpecificMarkovBackground

```
void GeneratePositionSpecificMarkovBackground(int order, int  
bucketSize)
```

create markov background models for different regions of sequence

based on GenerateMarkovDependency but now you look at -100:0, -200:-100, etc and create a background model for each

Parameters:

order - create nth order model
bucketSize - size of bucket for each mini-background

GetBackgroundProbability

```
double GetBackgroundProbability(Distr* d, char* seq, int len,  
HashTable* markov)
```

calculate background probability using markov background model

Parameters:

d - distr object
seq - input sequence
len - length of input sequence
markov - markov background model

GetBackgroundProbability2

```
double GetBackgroundProbability2(Distr* d, char* seq, int len)
```

calculate background probability using distr (count method)

Parameters:

d - distr object
seq - input sequence
len - length of input sequence

UCSF LIBRARY

GetCountFrequency

double **GetCountFrequency**(Distr* d, char* seq, int len)

get frequency of seq (in genome)

Parameters:

d - distr object

seq - input sequence

len - length of input sequence

GetSubSequence2

char* **GetSubSequence2**(char* id, int posStart, int posEnd, int strand)

same as GetSubSequence

copied from search. (so this is a second copy)

Parameters:

id - gene id

posStart - position start

posEnd - position end

strand - strand

LoadPositionSpecificMarkovBackground

HashTable** **LoadPositionSpecificMarkovBackground**(char* filename)

loads a position specific markov background

created using GeneratePositionSpecificMarkovBackground

Parameters:

filename - input filename

distr_Free

void **distr_Free**(Distr* d)

free object

Parameters:

d - distr object

distr_Get

unsigned int **distr_Get**(Distr* d, int idx)

gets value of index

Parameters:

d - distr object

idx - index corresponding to a sequence

distr_Incr

void **distr_Incr**(Distr* d, int idx)

increments count of that particular index

Parameters:

d - distr object

idx - index corresponding to a sequence

distr_Load

Distr* **distr_Load**(char* filename)

load distribution data from a file

Parameters:

filename - filename from which distribution data will be loaded

distr_Save

void **distr_Save**(Distr* d, FILE* fout)

save distribution data to a file

Parameters:

UCSF LIBRARY

d - distr object

fout - file pointer where data will be saved

distr_Set

```
void distr_Set(Distr* d, int idx, unsigned int val)
```

sets index to a particular value

Parameters:

d - distr object

idx - index corresponding to a sequence

val - value to be set

B.2.4. *dadbl.c*

pseudo-class of a dynamic array of doubles

Function Summary	
void	dadbl_Add (dadbl* d, double val) adds val to the end of tharray
dadbl*	dadbl_Free (dadbl* d, int deep) frees the dadbl object
double	dadbl_Get (dadbl* d, int index) returns a value from the array
int	dadbl_KeepBest (dadbl* d, int num, int (*func) (const void*, const void*), int deep) first optionally sorts, then keeps num elements
dadbl*	dadbl_New (int incrSize) create a dynamic array of doubles
void	dadbl_Save (dadbl* d, int (*printfn) (const char* format, ...)) saves to printfn
void	dadbl_Sort (dadbl* d, int (*func) (const void*, const void*)) sorts the array
void	dadbl_Trunc (dadbl* d, int num, int deep) truncates the array by num

dadbl_Add

```
void dadbl_Add(dadbl* d, double val)
```

adds val to the end of tharray

Parameters:

UCSF LIBRARY

d - the input dadbl object
val - the value to be added

dadbl_Free

```
dadbl* dadbl_Free(dadbl* d, int deep)
```

frees the dadbl object

Parameters:

d - the input dadbl object
deep - not used

dadbl_Get

```
double dadbl_Get(dadbl* d, int index)
```

returns a value from the array

Parameters:

d - the input dadbl object
index - index to be obtained

dadbl_KeepBest

```
int dadbl_KeepBest(dadbl* d, int num, int (*func) (const void*, const void*), int deep)
```

first optionally sorts, then keeps num elements

similar to [dadbl_Trunc](#)

Parameters:

d - the input dadbl object
num - number of elements to keep
func - sort function, optional
deep - not used

dadbl_New

```
dadbl* dadbl_New(int incrSize)
```

UCSF LIBRARY

create a dynamic array of doubles

dynamic as in the size of the array changes as necessary

Parameters:

`incrSize` - increment size of internal dynamic array

`dadbl_Save`

```
void dadbl_Save(dadbl* d, int (*printfn)(const char* format, ...))
```

saves to printfn

printfn examples include `OutF` and `LogF`

Parameters:

`d` - the input dadbl object

`printfn` - print function pointer

`dadbl_Sort`

```
void dadbl_Sort(dadbl* d, int (*func)(const void*, const void*))
```

sorts the array

Parameters:

`d` - the input dadbl object

`func` - sort function

`dadbl_Trunc`

```
void dadbl_Trunc(dadbl* d, int num, int deep)
```

truncates the array by num

similar to `dadbl_KeepBest`

Parameters:

`d` - the input dadbl object

`num` - number of elements to delete from end of array

`deep` - not used

UCSF LIBRARY

B.2.5. *daint.c*

pseudo-class of a dynamic array of ints

Function Summary	
void	daint_Add (daint* d, int val) adds val to array
daint*	daint_Free (daint* d, int deep) free array
int	daint_Get (daint* d, int index) gets value at index
int	daint_KeepBest (daint* d, int num, int (*func) (const void*, const void*), int deep) keeps a set number of elements in array
daint*	daint_New (int incrSize) create a dynamic array of ints
void	daint_Save (daint* d, int (*printfn) (const char* format, ...)) saves array to file
void	daint_Sort (daint* d, int (*func) (const void*, const void*)) sorts array
void	daint_Trunc (daint* d, int num, int deep) truncates array

daint_Add

void **daint_Add**(daint* d, int val)

adds val to array

Parameters:

d - the input daint object

val - value to add

daint_Free

daint* **daint_Free**(daint* d, int deep)

free array

Parameters:

d - the input daint object

deep - not used

daint_Get

```
int daint_Get(daint* d, int index)
```

gets value at index

Parameters:

d - the input daint object

index - index of interest

daint_KeepBest

```
int daint_KeepBest(daint* d, int num, int (*func) (const void*, const void*), int deep)
```

keeps a set number of elements in array

Parameters:

d - the input daint object

num - number of element to keep

func - optionally sort array

deep - not used

daint_New

```
daint* daint_New(int incrSize)
```

create a dynamic array of ints

dynamic as in the size of the array changes as necessary

Parameters:

incrSize - increment size of internal dynamic array

daint_Save

```
void daint_Save(daint* d, int (*printfn) (const char* format, ...))
```

saves array to file

printfn may contain something like OutF or LogF.

UCSF LIBRARY

Parameters:

d - the input daint object
printfn - print function pointer

daint_Sort

void **daint_Sort**(daint* d, int (*func) (const void*, const void*))

sorts array

Parameters:

d - the input daint object
func - sort function pointer

daint_Trunc

void **daint_Trunc**(daint* d, int num, int deep)

truncates array

Parameters:

d - the input daint object
num - number of elements to truncate
deep - not used

B.2.6. darrv.c

dynamic array of void*

Function Summary	
void	DArrV Add (DArrV* d, void* val) adds value to array
DArrV*	DArrV Free (DArrV* d, int deep) frees array
DArrV*	DArrV New (int incrSize) creates a new dynamic array

DArrV_Add

void **DArrV_Add**(DArrV* d, void* val)

UCSF LIBRARY

adds value to array

Parameters:

d - dynamic array object
val - value to add

DArrV_Free

DArrV* **DArrV_Free**(DArrV* d, int deep)

frees array

note that the deep flag only performs a free() on each void*; it might be necessary to do a custom free for each object

Parameters:

d - object to free
deep - boolean to free each object

DArrV_New

DArrV* **DArrV_New**(int incrSize)

creates a new dynamic array

Parameters:

incrSize - increment size

B.2.7. dasl.c

pseudo-class of a dynamic array of SiteLists

Function Summary	
void	dasl_Add (dasl* d, SiteList* sl) adds SiteList to array
void	dasl_Append (dasl* a, dasl* b) appends b to a
dasl*	dasl_Copy (dasl* d) shallow copy of dasl
dasl*	dasl_DeepCopy (dasl* d) deep copy of dasl
void	dasl_DelPos (dasl* d, int pos, int deep)

UCSF LIBRARY

	delete a specific SiteList
int	dasl_Exists (dasl* d, SiteList* sl) does a shallow check to see if sl exists
dasl*	dasl_Free (dasl* d, int deep) free array
SiteList*	dasl_Get (dasl* d, int index) gets SiteList at index
dasl*	dasl_GetRandom (dasl* d, int count, int start, int end) randomly choose some SiteList within a range
int	dasl_KeepBest (dasl* d, int num, int (*func) (const void*, const void*), int deep) keep portion of dasl
int	dasl_KeepBestThreshold (dasl* d, double min, int deep) keeps only SiteLists with minimum score
dasl*	dasl_Load (char* filename) loads dasl from file
SiteList*	dasl_Merge (dasl* d) flattens a dasl into a single SiteList
dasl*	dasl_New (int incrSize) create a dynamic array of SiteLists
void	dasl_PartialSort (dasl* d, int num, int (*func) (const void*, const void*)) sorts portion of dasl
double	dasl_Pearson (dasl* sllist) calculates Pearson's coefficient for pairs of scores within a dasl
void	dasl_Print (dasl* d, int num) print to OutF
void	dasl_Save (dasl* d, int (*printfn) (const char* format, ...)) saves to file
void	dasl_Sort (dasl* d, int (*func) (const void*, const void*)) sorts dasl
void	dasl_Trunc (dasl* d, int num, int deep) truncate dasl

dasl_Add

```
void dasl_Add(dasl* d, SiteList* sl)
```

adds SiteList to array

Parameters:

d - the input dasl object

sl - SiteList to add

UCSF LIBRARY

dasl_Append

void **dasl_Append**(dasl* a, dasl* b)

appends b to a

Parameters:

a - initial dasl

b - dasl to append

dasl_Copy

dasl* **dasl_Copy**(dasl* d)

shallow copy of dasl

Parameters:

d - the input dasl object

dasl_DeepCopy

dasl* **dasl_DeepCopy**(dasl* d)

deep copy of dasl

Parameters:

d - the input dasl object

dasl_DelPos

void **dasl_DelPos**(dasl* d, int pos, int deep)

delete a specific SiteList

Parameters:

d - the input dasl object

pos - position to delete

deep - flag of whether to free SiteList

UCSF LIBRARY

dasl_Exists

```
int dasl_Exists(dasl* d, SiteList* sl)
```

does a shallow check to see if sl exists

Parameters:

d - the input dasl object

sl - SiteList of interest

dasl_Free

```
dasl* dasl_Free(dasl* d, int deep)
```

free array

Parameters:

d - the input dasl object

deep - flag to free each SiteList object

dasl_Get

```
SiteList* dasl_Get(dasl* d, int index)
```

gets SiteList at index

Parameters:

d - the input dasl object

index - index of interest

dasl_GetRandom

```
dasl* dasl_GetRandom(dasl* d, int count, int start, int end)
```

randomly choose some SiteList within a range

Parameters:

d - the input dasl object

count - number to choose

UCSF LIBRARY

start - start range
end - end range

dasl_KeepBest

int **dasl_KeepBest**(dasl* d, int num, int (*func) (const void*, const void*), int deep)

keep portion of dasl

Parameters:

d - the input dasl object
num - number to keep
func - sort function pointer
deep - flag to do deep freeing

dasl_KeepBestThreshold

int **dasl_KeepBestThreshold**(dasl* d, double min, int deep)

keeps only SiteLists with minimum score

Parameters:

d - the input dasl object
min - minimum threshold
deep - flag to do deep freeing

dasl_Load

dasl* **dasl_Load**(char* filename)

loads dasl from file

format of file uses SiteList_PrettyLoad

Parameters:

filename - file to load

dasl_Merge

SiteList* **dasl_Merge**(dasl* d)

UCSF LIBRARY

flattens a dasl into a single SiteList

Parameters:

d - the input dasl object

dasl_New

dasl* **dasl_New**(int incrSize)

create a dynamic array of SiteLists

Parameters:

incrSize - increment size of internal dynamic array

dasl_PartialSort

void **dasl_PartialSort**(dasl* d, int num, int (*func) (const void*, const void*))

sorts portion of dasl

Parameters:

d - the input dasl object

num - number to sort

func - sort function pointer

dasl_Pearson

double **dasl_Pearson**(dasl* sllist)

calculates Pearson's coefficient for pairs of scores within a dasl

Parameters:

sllist - array of SiteLists

dasl_Print

void **dasl_Print**(dasl* d, int num)

print to OutF

UCSF LIBRARY

Parameters:

d - the input dasl object
num - number to print

dasl_Save

void **dasl_Save**(dasl* d, int (*printfn)(const char* format, ...))

saves to file

Parameters:

d - the input dasl object
printfn - print function pointer

dasl_Sort

void **dasl_Sort**(dasl* d, int (*func)(const void*, const void*))

sorts dasl

Parameters:

d - the input dasl object
func - sort function pointer

dasl_Trunc

void **dasl_Trunc**(dasl* d, int num, int deep)

truncate dasl

Parameters:

d - the input dasl object
num - number to truncate
deep - flag to do deep freeing

B.2.8. *dastr.c*

pseudo-class of dynamic array of strings

Function Summary

UCSF LIBRARY

void	dastr_Add (dastr* d, char* val) adds string to array
dastr*	dastr_Combine (dastr* a, dastr* b) creates a new dastr and adds two dastrs to it
char*	dastr_Exists (dastr* d, char* str) Check if string already exists in array
dastr*	dastr_Free (dastr* d, int deep) free object
char*	dastr_Get (dastr* d, int index) returns string at index
dastr*	dastr_GetFirstNStr (dastr* d, int num) gets the first n strings in array
dastr*	dastr_GetRandom (dastr* d, int count, dastr* disallowed) gets random strings, disallowing some of them
dastr*	dastr_Load (char* file) loads a set of strings from a file
dastr*	dastr_New (int incrSize) creates a new dynamic array of strings
void	dastr_Print (dastr* d) prints to LogF
void	dastr_Randomize (dastr* list) randomizes the order of the strings
void	dastr_SaveStream (dastr* seqList, int (*printfn)(const char* format, ...)) strings to print to stream
dastr*	dastr_Split (char* str) splits string at tabs and creates a new dastr, allocating memory for the split strings

dastr_Add

void **dastr_Add**(dastr* d, char* val)

adds string to array

Parameters:

d - dynamic array object

val - string to add

dastr_Combine

dastr* **dastr_Combine**(dastr* a, dastr* b)

creates a new dastr and adds two dastrs to it

Parameters:

a - dastr 1
b - dastr 2

dastr_Exists

char* **dastr_Exists**(dastr* d, char* str)

Check if string already exists in array

Parameters:

d - dynamic array object
str - string to check

dastr_Free

dastr* **dastr_Free**(dastr* d, int deep)

free object

Parameters:

d - dynamic array object
deep - flag to free string memory too

dastr_Get

char* **dastr_Get**(dastr* d, int index)

returns string at index

Parameters:

d - dynamic array object
index - index of string

dastr_GetFirstNStr

dastr* **dastr_GetFirstNStr**(dastr* d, int num)

gets the first n strings in array

Parameters:

d - dynamic array object
num - number of strings to get

dastr_GetRandom

dastr* **dastr_GetRandom**(dastr* d, int count, dastr* disallowed)

gets random strings, disallowing some of them

Parameters:

d - dynamic array object
count - number of strings
disallowed - disallowed strings

dastr_Load

dastr* **dastr_Load**(char* file)

loads a set of strings from a file

Parameters:

file - filename

dastr_New

dastr* **dastr_New**(int incrSize)

creates a new dynamic array of strings

Parameters:

incrSize - increment size of dynamic array

dastr_Print

void **dastr_Print**(dastr* d)

prints to LogF

Parameters:

d - dynamic array object

dastr_Randomize

```
void dastr_Randomize(dastr* list)
```

randomizes the order of the strings

Parameters:

list - list of strings

dastr_SaveStream

```
void dastr_SaveStream(dastr* seqList, int (*printfn)(const char*  
format, ...))
```

strings to print to stream

Parameters:

seqList - list of strings

printfn - print function pointer

dastr_Split

```
dastr* dastr_Split(char* str)
```

splits string at tabs and creates a new dastr, allocating memory for the split strings

Parameters:

str - input string

B.2.9. *inhash.c*

hash table using int as the key

please see oldhash.c for equivalent comments

B.2.10. *matchlist.c*

pseudo-class to handle pairs of sequence locations in a space efficient manner

UCSF LIBRARY

a SiteMatch is the location of a site

SiteMatchList is an array of SiteMatches

a MatchList contains the actual array of pairs of SiteMatches

Function Summary	
void	MatchList Add (MatchList* ml, SiteMatch* sm1, SiteMatch* sm2) Adds a pair of SiteMatches to the MatchList object
void	MatchList Free (MatchList* ml, int deep) Frees MatchList
char*	MatchList GetSeqID (MatchList* ml, int site) Gets gene id of one side of the matches
SiteMatch*	MatchList GetSiteMatch (MatchList* ml, int match, int site) gets a SiteMatch
MatchList*	MatchList New (int incrSize, char* id1, char* id2) creates a new MatchList
MatchList*	MatchList ReorderMatches (MatchList* ml, char* id) puts the specified gene id in the first position
void	SML Add (SiteMatchList* sml, SiteMatch* sm) adds SiteMatch to SiteMatchList
SiteMatchList*	SML Create (int incrSize, char* id) creates a dynamic array of SiteMatches
void	SML Free (SiteMatchList* sml) frees object
void	SiteMatch Set (SiteMatch* sm, short start, short end, char compl) sets values in a SiteMatch

MatchList_Add

```
void MatchList_Add(MatchList* ml, SiteMatch* sm1, SiteMatch* sm2)
```

Adds a pair of SiteMatches to the MatchList object

Parameters:

ml - MatchList object

sm1 - SiteMatch 1

sm2 - SiteMatch 2

MatchList_Free

```
void MatchList_Free(MatchList* ml, int deep)
```

Frees MatchList

Parameters:

ml - MatchList object

deep - whether to free the internal SiteListMatch objects

MatchList_GetSeqID

char* **MatchList_GetSeqID**(MatchList* ml, int site)

Gets gene id of one side of the matches

Parameters:

ml - MatchList object

site - (0 or 1)

MatchList_GetSiteMatch

SiteMatch* **MatchList_GetSiteMatch**(MatchList* ml, int match, int site)

gets a SiteMatch

Parameters:

ml - MatchList object

match - match index

site - which side of the matches (0 or 1)

MatchList_New

MatchList* **MatchList_New**(int incrSize, char* id1, char* id2)

creates a new MatchList

intentially it's two arrays of SiteMatchLists

Parameters:

incrSize - increment size of dynamic array

id1 - gene id of one side of the matches

id2 - gene id of the other side of the matches

MatchList_ReorderMatches

MatchList* **MatchList_ReorderMatches**(MatchList* ml, char* id)

puts the specified gene id in the first position

Parameters:

ml - MatchList object

id - gene id

SML_Add

void **SML_Add**(SiteMatchList* sml, SiteMatch* sm)

adds SiteMatch to SiteMatchList

Parameters:

sml - SML object

sm - SiteMatch to add

SML_Create

SiteMatchList* **SML_Create**(int incrSize, char* id)

creates a dynamic array of SiteMatches

Parameters:

incrSize - increment size for dynamic array

id - designate a gene id for the SML, since all SML will refer to the same gene

SML_Free

void **SML_Free**(SiteMatchList* sml)

frees object

Parameters:

sml - SML object

UCSF LIBRARY

SiteMatch_Set

void **SiteMatch_Set**(SiteMatch* sm, short start, short end, char compl)

sets values in a SiteMatch

Parameters:

sm - input sitematch

start - start position

end - end position

compl - strand (0 for forward, 1 for reverse)

B.2.11. *microarray.c*

pseudo-class to handle microarray data

data is stored in a tab delimited format, with the first row containing the sample names (optionally) and the first column containing gene ids

several functions are devoted to subsetting samples, which was not used in the MaMF paper

Function Summary	
double	ma_CalcPearson (HashTable* ma, char* gene1, char* gene2) Calculates Pearson's correlation of expression data between two genes
dastr*	ma_CalcPearsonBestGenes (HashTable* ma, dastr* geneList, int numGenes, int* subset, int numSubset) calculates genes that get the best average pairwise Pearson's correlation with a specified list of genes
SiteList*	ma_CalcPearsonGeneVsAll (HashTable* ma, char* gene1, int* subset, int numSubset) sorts genes by those that have the best Pearson's correlation to the specified gene1
double	ma_CalcPearsonGeneVsGeneSet (HashTable* ma, char* geneID, dastr* geneList, int* subset, int numSubset) calculates average Pearson's between a gene and a list of genes, with optional subsetting
double	ma_CalcPearsonScore (HashTable* ma, dastr* geneList) returns the average pearson's correlation using all the pairwise comparisons between genes
double	ma_CalcPearsonScore2Subset (HashTable* ma, dastr* origGenes, dastr* foundGenes, int* subset, int numSubset) calculates average pairwise pearson's between genes in first gene set versus genes in second gene set, allowing subsets

double	ma CalcPearsonScoreSubset (HashTable* ma, dastr* geneList, int* subset, int numSubset) calculates average pairwise pearson's using subset of samples
double	ma CalcPearsonSubset (HashTable* ma, char* gene1, char* gene2, int* subset, int numSubset) calculates pearson's using subset of samples
SiteList*	ma CalcPearsonVsGenes (HashTable* ma, dastr* geneList, int* subset, int numSubset) sorts genes by average pairwise Pearson's correlation with a specified list of genes
void	ma CompareSamples (HashTable* ma) calculates Pearson's correlation between samples (instead of genes)
void	ma Free (HashTable* ma) frees the microarray
HashTable*	ma LoadArray (char* arrayFile, int hasHeader) loads microarray into memory
int*	ma MaximizePearsonSubset (HashTable* ma, dastr* geneList, int numSubset, int numIter, double sampleThreshold) stochastic hillclimbing to maximize average pairwise pearson's within a gene set by choosing the optimal subset
HashTable*	ma New (int expectedNumGenes) creates a new microarray object
void	ma PlotPearsonScore (HashTable* ma, dastr* geneList) calculates Pearson's correlation between genes in gene list and prints them
void	ma RandomizeValues (HashTable* ma) randomize non null cells in microarray to [0..1]
void	ma RemoveSamples (HashTable* ma, char* filename) removes samples from a microarray and outputs edited microarray
void	ma ScrambleValuesWithinGenes (HashTable* ma) randomizes the order of expression values per gene
void	ma ViewSubset (HashTable* ma, char* filename) retrieves the sample data for all genes for pairs of samples

HashIterateNext

```
for (iter=HashNewIterator(seqs); iter->val != NULL;
HashIterateNext(iter))
```

returns the next value in iterator

hi->key contains the key associated with it

Parameters:

hi - hash iterator object

ma_CalcPearson

double **ma_CalcPearson**(HashTable* ma, char* gene1, char* gene2)

Calculates Pearson's correlation of expression data between two genes

expression data is obtained for each sample between the genes; we require at least 10 sample shared between the two genes for the PC to be calculated

the PC measures the level of coexpression between two genes

Parameters:

ma - microarray object

gene1 - gene id 1

gene2 - gene id 2

ma_CalcPearsonBestGenes

dastr* **ma_CalcPearsonBestGenes**(HashTable* ma, dastr* geneList, int numGenes, int* subset, int numSubset)

calculates genes that get the best average pairwise Pearson's correlation with a specified list of genes

see [ma_CalcPearsonVsGenes](#)

Parameters:

ma - microarray object

geneList - list of genes

numGenes - number of best genes to return

subset - sample subset to use

numSubset - number of samples in subset, 0 to use all

ma_CalcPearsonGeneVsAll

SiteList* **ma_CalcPearsonGeneVsAll**(HashTable* ma, char* gene1, int* subset, int numSubset)

sorts genes by those that have the best Pearson's correlation to the specified gene1

I don't handle subsets...

Parameters:

UCSF LIBRARY

ma - microarray object
gene1 - gene id 1
subset - sample subset to use
numSubset - number of samples in subset, 0 to use all

ma_CalcPearsonGeneVsGeneSet

double **ma_CalcPearsonGeneVsGeneSet**(HashTable* ma, char* geneID, dastr* geneList, int* subset, int numSubset)

calculates average Pearson's between a gene and a list of genes, with optional subsetting

Parameters:

ma - microarray object
geneID - gene ID
geneList - list of genes
subset - sample subset to use
numSubset - number of samples in subset, 0 to use all

ma_CalcPearsonScore

double **ma_CalcPearsonScore**(HashTable* ma, dastr* geneList)

returns the average pearson's correlation using all the pairwise comparisons between genes

Parameters:

ma - microarray object
geneList - list of genes

ma_CalcPearsonScore2Subset

double **ma_CalcPearsonScore2Subset**(HashTable* ma, dastr* origGenes, dastr* foundGenes, int* subset, int numSubset)

calculates average pairwise pearson's between genes in first gene set versus genes in second gene set, allowing subsets

Parameters:

ma - microarray object
origGenes - first gene set
foundGenes - second gene set

UCSF LIBRARY

subset - sample subset to use
numSubset - number of samples in subset, 0 to use all

ma_CalcPearsonScoreSubset

double **ma_CalcPearsonScoreSubset**(HashTable* ma, dastr* geneList, int* subset, int numSubset)

calculates average pairwise pearson's using subset of samples

Parameters:

ma - microarray object
geneList -
subset - sample subset to use
numSubset - number of samples in subset, 0 to use all

ma_CalcPearsonSubset

double **ma_CalcPearsonSubset**(HashTable* ma, char* gene1, char* gene2, int* subset, int numSubset)

calculates pearson's using subset of samples

Parameters:

ma - microarray object
gene1 - gene id 1
gene2 - gene id 2
subset - sample subset to use
numSubset - number of samples in subset, 0 to use all

ma_CalcPearsonVsGenes

SiteList* **ma_CalcPearsonVsGenes**(HashTable* ma, dastr* geneList, int* subset, int numSubset)

sorts genes by average pairwise Pearson's correlation with a specified list of genes

Parameters:

ma - microarray object
geneList - specified list of genes
subset - sample subset to use
numSubset - number of samples in subset, 0 to use all

UCSF LIBRARY

ma_CompareSamples

void **ma_CompareSamples**(HashTable* ma)

calculates Pearson's correlation between samples (instead of genes)

basically a measure of similarity between samples

uses PearsonsCoeff

I used this to toss out some highly similar (in fact identical) samples in the Stuart data set

Parameters:

ma - microarray object

ma_Free

void **ma_Free**(HashTable* ma)

frees the microarray

Parameters:

ma - microarray object

ma_LoadArray

HashTable* **ma_LoadArray**(char* arrayFile, int hasHeader)

loads microarray into memory

Parameters:

arrayFile - filename of microarray

hasHeader - flag to specify whether a header exists

ma_MaximizePearsonSubset

int* **ma_MaximizePearsonSubset**(HashTable* ma, dastr* geneList, int numSubset, int numIter, double sampleThreshold)

stochastic hillclimbing to maximize average pairwise pearson's within a gene set by choosing the optimal subset

UCSF LIBRARY

also has a simulated annealing feature

Parameters:

ma - microarray object

geneList - list of genes

numSubset - number of samples in subset, 0 to use all

numIter - number of iterations to do hillclimbing

sampleThreshold - required percentage of genes that use particular sample in order to be considered in subset

ma_New

HashTable* **ma_New**(int expectedNumGenes)

creates a new microarray object

stored as a hash table

Parameters:

expectedNumGenes - approximate number of genes in microarray

ma_PlotPearsonScore

void **ma_PlotPearsonScore**(HashTable* ma, dastr* geneList)

calculates Pearson's correlation between genes in gene list and prints them

Parameters:

ma - microarray object

geneList - list of genes

ma_RandomizeValues

void **ma_RandomizeValues**(HashTable* ma)

randomize non null cells in microarray to [0..1]

Parameters:

ma - microarray object

UCSF LIBRARY

ma_RemoveSamples

void **ma_RemoveSamples**(HashTable* ma, char* filename)

removes samples from a microarray and outputs edited microarray

after deciding that there were duplicates in the Stuart data in [ma_CompareSamples](#), I used this to edit the [microarray](#)

Parameters:

ma - microarray object

filename - file containing a list of sample index positions

ma_ScrambleValuesWithinGenes

void **ma_ScrambleValuesWithinGenes**(HashTable* ma)

randomizes the order of expression values per gene

Parameters:

ma - microarray object

ma_ViewSubset

void **ma_ViewSubset**(HashTable* ma, char* filename)

retrieves the sample data for all genes for pairs of samples

probably to see what was going on in [ma_CompareSamples](#)

Parameters:

ma - microarray object

filename - file containing pearson's correlation, sample1 and sample2 per line

B.2.12. *motif-finder.c*

mainly to coordinate what is to be done in this run via arguments from command

line

Function Summary

UCSF LIBRARY

void	ProcessCommandLine (int argc, void* argv[]) process input from command line
int	main (int argc, void* argv[]) First function called from command line, controls which functions are called as a result of the arguments on the command line

ProcessCommandLine

void **ProcessCommandLine**(int argc, void* argv[])

process input from command line

two formats, either as a two valued argument, e.g. -[argument] [value

or a single valued argument, e.g. -[argument]

Parameters:

argc - number of arguments from command line

argv[] - command line arguments

main

int **main**(int argc, void* argv[])

First function called from command line, controls which functions are called as a result of the arguments on the command line

Parameters:

argc - number of arguments from command line

argv[] - arguments from command line

B.2.13. *oldhash.c*

hash table library

Function Summary	
	HashConstructTable () Creates a new hash table
	HashDel () Delete a key from the hash table and return associated data
double	HashDoubleMinusMinus (HashTable* ht, char* key)

	<p>treats the value of a key as a double and decrements</p>
double	<p>HashDoublePlusPlus (HashTable* ht, char* key) treats the value of a key as a double and increments</p>
void	<p>HashFreeTable (HashTable *table, void (*func) (void *)) frees hash table</p>
	<p>HashInsert () inserts a key into the hash table</p>
for (iter=HashNewIterator(seqs); iter->val != NULL;	<p>HashIterateNext (iter)) returns the next value in iterator</p>
HashTable*	<p>HashLoadString (char* filename) Loads a hash table of strings</p>
HashIter*	<p>HashNewIterator (HashTable* ht) creates a hash iterator</p>
int	<p>HashPlusPlus (HashTable* ht, char* key) treats the value of a key as an int and decrements</p>
HashTable*	<p>LoadHashDouble (char* filename) Loads a hash table of doubles</p>
static unsigned	<p>hash (const char* ptr) Generates a hash value for a string</p>

HashConstructTable

HashConstructTable ()

Creates a new hash table

Parameters:

table - generally set to NULL
size - expected size of hash table

HashDel

HashDel ()

Delete a key from the hash table and return associated data

returns NULL if not present

Parameters:

UCSF LIBRARY

key - key to delete
table - hash table

HashDoubleMinusMinus

double **HashDoubleMinusMinus**(HashTable* ht, char* key)

treats the value of a key as a double and decrements

used for markov stuff, a little hackish

Parameters:

ht - hash table
key - key

HashDoublePlusPlus

double **HashDoublePlusPlus**(HashTable* ht, char* key)

treats the value of a key as a double and increments

used for markov stuff, a little hackish

Parameters:

ht - hash table
key - key

HashFreeTable

void **HashFreeTable**(HashTable *table, void (*func)(void *))

frees hash table

Parameters:

table - table to be freed
func - function pointer to be applied to each element in table

HashInsert

HashInsert ()

inserts a key into the hash table

returns NULL if it failed to be inserted, like if it already exists

Parameters:

key - key of the key-value pair

data - value of the key-value pair, can be anything

table - hash table

HashIterateNext

```
for (iter=HashNewIterator(seqs); iter->val != NULL;
HashIterateNext(iter))
```

returns the next value in iterator

hi->key contains the key associated with it

Parameters:

hi - hash iterator object

HashLoadString

```
HashTable* HashLoadString(char* filename)
```

Loads a hash table of strings

Parameters:

filename - filename containing key-value pairs

HashNewIterator

```
HashIter* HashNewIterator(HashTable* ht)
```

creates a hash iterator

needs to be freed when done

Parameters:

ht - hash table

HashPlusPlus

int **HashPlusPlus**(HashTable* ht, char* key)

treats the value of a key as an int and decrements

Parameters:

ht - hash table

key - key

LoadHashDouble

HashTable* **LoadHashDouble**(char* filename)

Loads a hash table of doubles

used for markov stuff

key is a string, value is a double

Parameters:

filename -

filename - filename containing key-value pairs

hash

static unsigned **hash**(const char* ptr)

Generates a hash value for a string

An excellent string hashing function. Adapted from glib's `g_str_hash()`. Investigation by Karl Nelson . Do a web search for "g_str_hash X31_HASH" if you want to know more.

Parameters:

ptr - string to be hashed

B.2.14. *pwm.c*

pseudo-class for a position weight matrix (PWM) object

UCSF LIBRARY

each cell in the matrix contains an integer

Function Summary	
double	PWM_AddSite (PWM* motif, SiteWin* sw) adds frequency values of SiteWin to the PWM
double	PWM_CalcGCContent (PWM* motif) Calculate GC content of motif
double	PWM_CalcInformationContent (PWM* motif) calculate information content of motif
PWM*	PWM_Copy (PWM* pwm) copies PWM
PWM*	PWM_CreateFromOffsetSiteList (SiteList* sl, int offset, int width) Creates a PWM from the SiteList, starting from an offset
PWM*	PWM_CreateFromSiteList (SiteList* sl, int width) creates a PWM from the SiteList
PWM*	PWM_CreateFromSiteWin (SiteWin* sw, int width) Creates a PWM based on a SiteWin
PWM*	PWM_CreateRandom (int numSites, int width) creates a random PWM
void	PWM_DetermineConsensus (PWM* motif, char* consensus) generates the consensus sequence from the PWM using a small number of IUPAC symbols
void	PWM_DetermineConsensusStranded (PWM* motif, char* consensus, int strand) generates consensus from PWM or its reverse complement
void	PWM_DetermineMaxes (PWM* motif) determines the max score of each position in PWM
void	PWM_ExpandSites (PWM* motif, int multiplier) Multiplies values in PWM by set amount
void	PWM_Free (PWM* motif) free PWM
PWM*	PWM_GenRandom (int width, int sum) a more efficient way to randomly generate a motif
SiteWin*	PWM_GetBestSubseqAlignment (PWM* motif, Sequence* seq, int width) finds best alignment of the PWM to the sequence
double	PWM_GetWorstScoreSiteList (PWM* motif, SiteList* sl, int num) returns the score of the worst aligning site against the PWM
PWM*	PWM_Load (char* filename) Loads a PWM from a file
PWM*	PWM_Load2 (FILE* fin) Loads a PWM from a file, enhanced
PWM*	PWM_New (int width) creates a new PWM
void	PWM_Print (PWM* motif) Prints internals of PWM

double	PWM_ScoreDNA (PWM* motif, unsigned int dna) calculates PWM score of DNA-type sequence
double	PWM_ScoreSite (PWM* motif, char* s) calculate PWM score of site
double	PWM_ScoreSiteList (PWM* motif, SiteList* sl, int num) scores all sites against PWM and returns the sum of scores of the top num sites
double	PWM_ScoreSlidingPWM (PWM* extendedPWM, PWM* motif, int* offset, int* strand) finds the score of the best alignment of motif against the extendedPWM
void	PWM_Stream (PWM* motif, int (*printfn)(const char* format, ...)) Prints PWM internals to stream

PWM_AddSite

double **PWM_AddSite**(PWM* motif, SiteWin* sw)

adds frequency values of SiteWin to the PWM

Parameters:

motif - PWM object

sw - SiteWin to add

PWM_CalcGCCContent

double **PWM_CalcGCCContent**(PWM* motif)

Calculate GC content of motif

Parameters:

motif - PWM object

PWM_CalcInformationContent

double **PWM_CalcInformationContent**(PWM* motif)

calculate information content of motif

Parameters:

motif - PWM object

PWM_Copy

PWM* **PWM_Copy**(PWM* pwm)

copies PWM

Parameters:

pwm - input PWM

PWM_CreateFromOffsetSiteList

PWM* **PWM_CreateFromOffsetSiteList**(SiteList* sl, int offset, int width)

Creates a PWM from the SiteList, starting from an offset

Parameters:

sl - input SiteList

offset - offset index

width - width of PWM

PWM_CreateFromSiteList

PWM* **PWM_CreateFromSiteList**(SiteList* sl, int width)

creates a PWM from the SiteList

Parameters:

sl - input sitelist

width - width of sitelist

PWM_CreateFromSiteWin

PWM* **PWM_CreateFromSiteWin**(SiteWin* sw, int width)

Creates a PWM based on a SiteWin

Parameters:

sw - input SiteWin

width - width of SiteWin

UCSF LIBRARY

PWM_CreateRandom

PWM* **PWM_CreateRandom**(int numSites, int width)

creates a random PWM

Parameters:

numSites - number of sites in the PWM

width - width of PWM

PWM_DetermineConsensus

void **PWM_DetermineConsensus**(PWM* motif, char* consensus)

generates the consensus sequence from the PWM using a small number of IUPAC symbols

Parameters:

motif - PWM object

consensus - pre-allocated string that will contain consensus

PWM_DetermineConsensusStranded

void **PWM_DetermineConsensusStranded**(PWM* motif, char* consensus, int strand)

generates consensus from PWM or its reverse complement

if strand==0 then same as PWM_DetermineConsensus

Parameters:

motif - PWM object

consensus - pre-allocated string that will contain consensus

strand - whether to look at reverse complement of PWM

PWM_DetermineMaxes

void **PWM_DetermineMaxes**(PWM* motif)

determines the max score of each position in PWM

UCSF LIBRARY

calls `PWM_RunningMax` also

Parameters:

motif - PWM object

PWM_ExpandSites

void **PWM_ExpandSites**(PWM* motif, int multiplier)

Multiplies values in PWM by set amount

Parameters:

motif - PWM object

multiplier - multiply values in PWM by this amount

PWM_Free

void **PWM_Free**(PWM* motif)

free PWM

Parameters:

motif - PWM object

PWM_GenRandom

PWM* **PWM_GenRandom**(int width, int sum)

a more efficient way to randomly generate a motif

Parameters:

width - width of motif to create

sum - sum of each column

PWM_GetBestSubseqAlignment

SiteWin* **PWM_GetBestSubseqAlignment**(PWM* motif, Sequence* seq, int width)

finds best alignment of the PWM to the sequence

UCSF LIBRARY

Parameters:

motif - PWM object
seq - input sequence
width - width of alignment, asserted to be that of motif

PWM_GetWorstScoreSiteList

double **PWM_GetWorstScoreSiteList**(PWM* motif, SiteList* sl, int num)

returns the score of the worst aligning site against the PWM

Parameters:

motif - PWM object
sl - list of binding sites
num - not used

PWM_Load

PWM* **PWM_Load**(char* filename)

Loads a PWM from a file

Parameters:

filename - filename containing PWM

PWM_Load2

PWM* **PWM_Load2**(FILE* fin)

Loads a PWM from a file, enhanced

this is for loading PWMs with the #PWM format, which allows the data to be part of a larger file

Parameters:

fin - file pointer containing PWM data

PWM_New

PWM* **PWM_New**(int width)

creates a new PWM

Parameters:

width - width of new PWM

PWM_Print

void **PWM_Print**(PWM* motif)

Prints internals of PWM

Parameters:

motif - PWM object

PWM_ScoreDNA

double **PWM_ScoreDNA**(PWM* motif, unsigned int dna)

calculates PWM score of DNA-type sequence

Parameters:

motif - PWM object

dna - DNA number

PWM_ScoreSite

double **PWM_ScoreSite**(PWM* motif, char* s)

calculate PWM score of site

Parameters:

motif - PWM object

s - sequence that aligns to motif

PWM_ScoreSiteList

double **PWM_ScoreSiteList**(PWM* motif, SiteList* sl, int num)

scores all sites against PWM and returns the sum of scores of the top num sites

Parameters:

motif - PWM object
 sl - list of binding sites
 num - number of sites to contribute to score

PWM_ScoreSlidingPWM

```
double PWM_ScoreSlidingPWM(PWM* extendedPWM, PWM* motif, int* offset,
int* strand)
```

finds the score of the best alignment of motif against the extendedPWM

Parameters:

extendedPWM - a PWM that has been extended, should be larger than motif
 motif - PWM object
 offset - returns the offset of the best alignment of motif to the extendedPWM
 strand - returns the strand of the best alignment of motif to the extendedPWM

PWM_Stream

```
void PWM_Stream(PWM* motif, int (*printfn)(const char* format, ...))
```

Prints PWM internals to stream

Parameters:

motif - PWM object
 printfn - print function pointer

B.2.15. score.c

functions to calculate scores of motifs

The main function is GetTurboScore

There's a lot of other scoring functions I played with that didn't pan out, which I didn't document, but they follow a similar format as GetTurboScore

Function Summary	
double	AvLogMarkovSeqProb (SiteList* list, int width) calculates average log markov probability of a set of sequences
double	GetBPMotifScore (SiteList* list, int width)

	calculates the Bioprospector score of a motif
double	<u>GetInformationContent</u> (SiteList* list, int width) calculates information content of motif
int	<u>GetLinearScore</u> (SiteList* list, int width) calculates number of pairwise matches between all sequences
int	<u>GetLinearScoreWindow</u> (SiteList* list, int width, int start) calculates number of pairwise matches within window in motif
double	<u>GetLinearScoreWithBackground</u> (SiteList* list, int width) calculates score incorporating linear score and markov background probability
double	<u>GetTurboScore</u> (SiteList* list, int width) calculate the score used in MaMF paper
double	<u>GetTurboScorePlusSiteWin</u> (SiteList* list, SiteWin* sw, int width) calculates hypothetical score of adding a particular SiteWin to the motif
int	<u>LinearCompare</u> (char* a, char* b, int len) calculates number of matches between sequences

AvLogMarkovSeqProb

double **AvLogMarkovSeqProb**(SiteList* list, int width)

calculates average log markov probability of a set of sequences

Parameters:

list - motif

width - width of sequences

GetBPMotifScore

double **GetBPMotifScore**(SiteList* list, int width)

calculates the Bioprospector score of a motif

uses recent scoring function in MDScan (also used in BP now)

Parameters:

list - motif

width - width of sequences

GetInformationContent

double **GetInformationContent**(SiteList* list, int width)

calculates information content of motif

Parameters:

list - input motif
width - width of sequences

GetLinearScore

```
int GetLinearScore(SiteList* list, int width)
```

calculates number of pairwise matches between all sequences

similar to LinearCompare, but expanded to more than two sequences

Parameters:

list - motif
width - width of sequences

GetLinearScoreWindow

```
int GetLinearScoreWindow(SiteList* list, int width, int start)
```

calculates number of pairwise matches within window in motif

similar to GetLinearScore

Parameters:

list - motif
width - width of window
start - index offset

GetLinearScoreWithBackground

```
double GetLinearScoreWithBackground(SiteList* list, int width)
```

calculates score incorporating linear score and markov background probability

see GetLinearScore, ComputeMarkovProbability

Parameters:

list - input motif
width - width of sequences

GetTurboScore

double **GetTurboScore**(SiteList* list, int width)

calculate the score used in MaMF paper

there is caching of the scores to make it faster as the motif is greedily built up, so you don't have to recount all the pairwise comparisons, or the background probabilities

see also GetTurboScorePlusSitewin, which calculates hypothetical score of adding a particular SiteWin to the motif

assume list has sequences of equal length, length == width

Parameters:

list - motif

width - width of sequences

GetTurboScorePlusSitewin

double **GetTurboScorePlusSitewin**(SiteList* list, SiteWin* sw, int width)

calculates hypothetical score of adding a particular SiteWin to the motif

see also GetTurboScore

Parameters:

list - motif

sw - SiteWin to add

width - width of sequences

LinearCompare

int **LinearCompare**(char* a, char* b, int len)

calculates number of matches between sequences

Parameters:

a - sequence 1

b - sequence 2

len - length of sequences

B.2.16. search.c

greedy motif finding using indexing to accelerate the search process

start with TurboGreedyMotifSearch to understand what's going on

sadly, as I look at this code more I also found that I ended up not allowing any mismatches in the nmers at all (see GetScoringTable) since in motif-finder.c cNmerScoreThreshold is set to cNmerSize*2 , so all this clever programming wasn't actually used, probably because of performance reasons

there's some other ideas using different methods (like Gibbs sampling), but I didn't bother documenting them since they don't work

Function Summary	
void	AddIndexToCache (HashTable* cache, char* nmer, int idx) Adds an index to be associated with an nmer
void	AddOverlappingSiteWin (SiteList* osl, SiteList* sl, SiteWin* sw, MatchList*** matchesGrid, HashTable* seqHash, int width, double (*ScoringMetric)(SiteList*, int)) uses pre-built matches to find similar sequences to that of a SiteWin and adds them to osl
void	CacheNmerCombinations (HashTable* cache, int idx, char* nmer, int nmerSize, int nmerScoreThreshold, scoret* st) caches nmer and IUPAC synonyms for given position idx
void	CompareCaches (iHashTable* matches, int minScore, int width, daint* cache1, daint* cache2, char* id1, char* id2, ba2d* comparedArray) does simple sequence comparison using the index caches
int	CompareNucleotides (char nucl, char nuc2, int strand1, int strand2) checks if nucleotides are identical, considering reverse complement if necessary
MatchList***	CreateMatches (dastr* seqList, int width, double (*ScoringMetric)(SiteList*, int), HashTable* seqHash) creates matches Grid
HashTable*	CreateTurboCache (Sequence* seq, int width, scoret* scoringTable) generates a cache (index table) from a sequence
void	EnumerateMatchWindows (dasl* bestmatches, MatchList* ml, int width, double (*ScoringMetric)(SiteList*, int), double* min, double* max) enumerate sequence alignments and keep the best to be used as seeds
SiteList*	ExtendSiteList (SiteList* sl, int extendleft, int extendright) extends the SiteList by specified amount, filling with genomic sequence
void	FreeMatchesGrid (MatchList*** matchesGrid, dastr* seqList)

	freed the matches grid
dasl*	GenerateSeeds (dastr* seqList, int width, double (*ScoringMetric)(SiteList*, int), MatchList*** matchesGrid) generates seeds to be used in step 3
int	GetIndexFromPosition (int pos, int strand) converts position and strand into an index
int	GetIndexFromSiteList (SiteList* sl) calculates the super index from the SiteList
int	GetPositionFromIndex (int idx, int* strand) converts index into position and strand
int	GetPositionFromSiteWin (SiteWin* sw, int* strand) retrieves strand and position from SiteWin
void	GetPositionsFromSuperIndex (unsigned int idx, int* pos1, int* strand1, int* pos2, int* strand2) gets two sets of position from an index
double	GetScoringMetricPlusSiteWin (double (*ScoringMetric)(SiteList*, int), SiteList* sl, SiteWin* sw, int width) Calculates the score of the SiteList with additional SiteWin
scoret*	GetScoringTable (int type) returns a scoring table to judge how nucleotide alignments are scored
SiteWin*	GetSiteWin (char* id, int posStart, int posEnd, int strand, int getSeq) constructs a SiteWin to annotate sequence
char*	GetSubSequence (char* id, int posStart, int posEnd, int strand) gets sub sequence from gene promoter
unsigned int	GetSuperIndex (int pos1, int strand1, int pos2, int strand2) gets an index from two sets of positions
dasl*	GreedyMerge (dasl* sllist, MatchList*** matchesGrid, HashTable* seqHash, dastr* seqList, int width, double (*ScoringMetric)(SiteList*, int)) Takes all seeds and greedily builds motifs from them
MatchList*	MergeMatches (ihashTable* matches, char* id1, char* id2, int width) merges matches into the space efficient MatchList structure
SiteList*	PartialGreedyMergeMatches (SiteList* slorig, MatchList*** matchesGrid, HashTable* seqHash, int width, double (*ScoringMetric)(SiteList*, int), SiteList* matchCache, int maxMatches) builds motif up greedily
void	RecordNmerFrequency (dasl* d, int width) annotate each of the SiteWins for their background probability
void	ScoreSiteListWindows (dasl* list, int width, double (*ScoringMetric)(SiteList*, int)) score all SiteList Windows
dasl*	TurboGreedyMotifSearch (dastr* seqList, int width, double (*ScoringMetric)(SiteList*, int)) the Papa function to do the greedy search

AddIndexToCache

```
void AddIndexToCache(HashTable* cache, char* nmer, int idx)
```

Adds an index to be associated with an nmer

Parameters:

cache - cache object
nmer - nmer index to which to add
idx - index to add to nmer hash

AddOverlappingSiteWin

```
void AddOverlappingSiteWin(SiteList* osl, SiteList* sl, SiteWin* sw,  
MatchList*** matchesGrid, HashTable* seqHash, int width, double  
(*ScoringMetric)(SiteList*, int))
```

uses pre-built matches to find similar sequences to that of a SiteWin and adds them to osl

Parameters:

osl - the running SiteList of overlapping SiteWins
sl - original SiteList (current motif in greedy search to be built up)
matchesGrid - all the matches
seqHash - hash of sequences to get correct index for matchesGrid
width - width
ScoringMetric - scoring function pointer

CacheNmerCombinations

```
void CacheNmerCombinations(HashTable* cache, int idx, char* nmer, int  
nmerSize, int nmerScoreThreshold, score_t* st)
```

caches nmer and IUPAC synonyms for given position idx

takes an nmer, calculates combinations, and then calls [AddIndexToCache](#) for each nmer derivative

Parameters:

cache - cache object
idx - position index of nmer
nmer - nmer to be considered
nmerSize - length of nmer

nmerScoreThreshold - threshold to consider nmer
st - score table used

CompareCaches

```
void CompareCaches(ihashTable* matches, int minScore, int width, daint*  
cache1, daint* cache2, char* id1, char* id2, ba2d* comparedArray)
```

does simple sequence comparison using the index caches

This is the beef of the accelerated sequence comparison using indexing.

We use the caches of two sequences, iterate through the caches and generate matches. These matches are stored in the matches variable, which is a hash of ints.

The matches variable is stored as a hash for space efficiency (instead of a 1000x1000 array for two 1000 bp sequences)

Parameters:

matches - results of matches are stored in a hash table
minScore - minimum score to be counted as a match
width - width of sequences
cache1 - sequence 1 stored as a cache
cache2 - sequence 2 stored as a cache
id1 - gene id 1
id2 - gene id 2
comparedArray - a log of whether a particular region in the comparison has already been examined

CompareNucleotides

```
int CompareNucleotides(char nucl, char nuc2, int strand1, int strand2)
```

checks if nucleotides are identical, considering reverse complement if necessary

Parameters:

so1 -
so2 -
nucl - nucleotide 1
nuc2 - nucleotide 2
strand1 - strand of nucl
strand2 - strand of nuc2

CreateMatches

```
MatchList*** CreateMatches(dastr* seqList, int width, double  
(*ScoringMetric)(SiteList*, int), HashTable* seqHash)
```

creates matches Grid

Parameters:

seqList - list of gene ids
width - width of sites
ScoringMetric - scoring function pointer
seqHash - id-to-index conversion

CreateTurboCache

```
HashTable* CreateTurboCache(Sequence* seq, int width, score*  
scoringTable)
```

generates a cache (index table) from a sequence

similar to cache.c from previous projects

Parameters:

seq - input sequence
width - width of sub sequences
scoringTable - how to deal with mismatches (not really used in MaMF paper)

EnumerateMatchWindows

```
void EnumerateMatchWindows(dasl* bestmatches, MatchList* ml, int width,  
double (*ScoringMetric)(SiteList*, int), double* min, double* max)
```

enumerate sequence alignments and keep the best to be used as seeds

Parameters:

bestmatches - output matches, a running total
ml - match list of a particular pair of sequences
width - width
ScoringMetric - scoring function pointer
min - min score of match to keep
max - max score of matches so far

ExtendSiteList

SiteList* **ExtendSiteList**(SiteList* sl, int extendleft, int extendright)

extends the SiteList by specified amount, filling with genomic sequence

obviously this requires the SiteLists to be already based on real sequence

Parameters:

sl - input SiteList

extendleft - extend to the left this many nucleotides

extendright - extend to the right this many nucleotides

FreeMatchesGrid

void **FreeMatchesGrid**(MatchList*** matchesGrid, dastr* seqList)

frees the matches grid

Parameters:

matchesGrid -

seqList - sequence ids

GenerateSeeds

dasl* **GenerateSeeds**(dastr* seqList, int width, double
(*ScoringMetric)(SiteList*, int), MatchList*** matchesGrid)

generates seeds to be used in step 3

EnumerateMatchWindows does the real work

Parameters:

seqList - sequence ids

width - width

ScoringMetric - scoring function pointer

matchesGrid - 2d array of matches

GetIndexFromPosition

int **GetIndexFromPosition**(int pos, int strand)

converts position and strand into an index

this limits input sequence to

see also GetPositionFromIndex

Parameters:

pos - position

strand - strand

GetIndexFromSiteList

```
int GetIndexFromSiteList(SiteList* sl)
```

calculates the super index from the SiteList

uses GetSuperIndex

Parameters:

sl - input SiteList containing two SiteWins

GetPositionFromIndex

```
int GetPositionFromIndex(int idx, int* strand)
```

converts index into position and strand

this limits input sequence to

see also GetIndexFromPosition

Parameters:

idx - input index

strand - output strand

GetPositionFromSiteWin

```
int GetPositionFromSiteWin(SiteWin* sw, int* strand)
```

retrieves strand and position from SiteWin

Parameters:

sw - input SiteWin
strand - strand to be returned

GetPositionsFromSuperIndex

void **GetPositionsFromSuperIndex**(unsigned int idx, int* pos1, int* strand1, int* pos2, int* strand2)

gets two sets of position from an index

see also [GetSuperIndex](#)

Parameters:

idx - input index
pos1 - output
strand1 - output
pos2 - output
strand2 - output

GetScoringMetricPlusSiteWin

double **GetScoringMetricPlusSiteWin**(double (*ScoringMetric)(SiteList*, int), SiteList* sl, SiteWin* sw, int width)

Calculates the score of the SiteList with additional SiteWin

grouping function to call the appropriate function given the scoring metric

Parameters:

ScoringMetric - scoring function pointer
sl - input sitelist
sw - input SiteWin
width - width of sw

GetScoringTable

score_t* **GetScoringTable**(int type)

returns a scoring table to judge how nucleotide alignments are scored

you can theoretically define arbitrary scoring tables to allow for types of mismatches

e.g. in type 1, IUPAC symbols R and Y get a score of 1, for instance if A and G are aligned which share an R, then you would get 1 point. But a complete match A and A would get 2 points

in the end I stuck with type 0 because the partial matches slowed down the algorithm a lot, though as I write this I wonder how much "a lot" actually is....

sadly, as I look at this code more I also found that I ended up not allowing any mismatches at all since in motif-finder. `cNmerScoreThreshold` is set to `cNmerSize*2` so all this clever programming wasn't actually used, probably because of performance reasons

Parameters:

type - type of scoring table, 0 is mismatches allowed, 1 is fancy IUPAC type matching

GetSiteWin

```
SiteWin* GetSiteWin(char* id, int posStart, int posEnd, int strand, int  
getSeq)
```

constructs a SiteWin to annotate sequence

calls GetSubSequence

Parameters:

id - gene id

posStart - position start

posEnd - position end

strand - strand

getSeq - flag to conditionally get sequence

GetSubSequence

```
char* GetSubSequence(char* id, int posStart, int posEnd, int strand)
```

gets sub sequence from gene promoter

Parameters:

id - gene id

posStart - position start

posEnd - position end

strand - strand

GetSuperIndex

unsigned int **GetSuperIndex**(int pos1, int strand1, int pos2, int strand2)

gets an index from two sets of positions

see also [GetPositionsFromSuperIndex](#)

Parameters:

pos1 - position 1
strand1 - strand 1
pos2 - position 2
strand2 - strand 2

GreedyMerge

dasl* **GreedyMerge**(dasl* sllist, MatchList*** matchesGrid, HashTable* seqHash, dastr* seqList, int width, double (*ScoringMetric)(SiteList*, int))

Takes all seeds and greedily builds motifs from them

calls [PartialGreedyMergeMatches](#) to do the real work

Parameters:

sllist - all seeds
matchesGrid - all the matches
seqHash - id-to-index conversion
width - width
ScoringMetric - scoring function pointer

MergeMatches

MatchList* **MergeMatches**(iHashTable* matches, char* id1, char* id2, int width)

merges matches into the space efficient MatchList structure

not only for space efficiency, but I think it makes it easier later on to enumerate all potential pairwise alignments

posX on negative strand is the "end" on forward strand...

for all matches, first seq is id1, second seq is id2

Parameters:

matches - matches as a hash table
id1 - first gene id
id2 - second gene id
width - width of sites

PartialGreedyMergeMatches

```
SiteList* PartialGreedyMergeMatches(SiteList* slorig, MatchList***  
matchesGrid, HashTable* seqHash, int width, double  
(*ScoringMetric)(SiteList*, int), SiteList* matchCache, int maxMatches)
```

builds motif up greedily

part of stage 3

matches only has sitelists of size 2

Parameters:

slorig - original SiteList
matchesGrid - all the pre-built matches
seqHash - id-to-index conversion
width - width
ScoringMetric - scoring function pointer
matchCache - cache of matches found from sequences in slorig
maxMatches - size to build motif to

RecordNmerFrequency

```
void RecordNmerFrequency(dasl* d, int width)
```

annotate each of the SiteWins for their background probability

Parameters:

d - array of motifs
width - width of sites

UCSF LIBRARY

ScoreSiteListWindows

void **ScoreSiteListWindows**(dasl* list, int width, double (*ScoringMetric)(SiteList*, int))

score all SiteList Windows

Parameters:

list - list of SiteList windows (pairs of SiteWins per SiteList)

width - width of sequences

ScoringMetric - score function pointer

TurboGreedyMotifSearch

dasl* **TurboGreedyMotifSearch**(dastr* seqList, int width, double (*ScoringMetric)(SiteList*, int))

the Papa function to do the greedy search

stage 1: match generation from cache method

stage 2: seed generation

stage 3: greedy search

Parameters:

seqList - list of sequence ids

width - width of resulting motifs

ScoringMetric - scoring function pointer

B.2.17. *sequence.c*

utility functions to handle the Sequence object

Function Summary	
SiteList*	GenerateManySeqSet (dastr* s, int width) Generates all possible SiteWin binding sites for many sequences
SiteList*	GenerateOneSeqSet (Sequence* seq, int width) Generates all possible binding sites for a given sequence
dastr*	GetKeysFromHash (HashTable* seqs) Gets all keys (gene ids) from a hash table
SiteWin*	GetRandomNmer (HashTable* seqs, dastr* seqList, int window) get a random nmer from a random sequence

SiteWin*	<u>GetRandomNmerInSeq</u> (Sequence* seq, int window) gets a random nmer in a sequence
SiteList*	<u>GetRandomNmers</u> (HashTable* seqs, dastr* seqList, int window, int n) Gets a bunch of random nmers
dastr*	<u>GetRandomSeqList</u> (int num) gets a random set of gene ids
HashTable*	<u>LoadSequences</u> (char* fastaFile, int num) Loads a group of sequences from a fasta file
void	<u>SaveFastaSequences</u> (dastr* seqList, char* filename) Saves sequences to file in standard fasta format
void	<u>SaveSequencesRestrictedFasta</u> (dastr* seqList) Saves sequences to OutF in Restricted Fasta format, compatible with Bioprospector
double	<u>Seq_CalcGCCContent</u> (Sequence* seq) calculates GC content of a sequence
void	<u>WhackSites</u> (SiteList* sl) takes binding sites and marks Ns in the original genomic sequence

GenerateManySeqSet

SiteList* **GenerateManySeqSet**(dastr* s, int width)

Generates all possible SiteWin binding sites for many sequences

calls GenerateOneSeqSet

Parameters:

s - list of gene ids

width - width of binding site

GenerateOneSeqSet

SiteList* **GenerateOneSeqSet**(Sequence* seq, int width)

Generates all possible binding sites for a given sequence

Parameters:

seq - input sequence

width - width of binding site

GetKeysFromHash

dastr* **GetKeysFromHash**(HashTable* seqs)

Gets all keys (gene ids) from a hash table

Parameters:

seqs - all sequences

GetRandomNmer

SiteWin* **GetRandomNmer**(HashTable* seqs, dastr* seqList, int window)

get a random nmer from a random sequence

calls GetRandomNmerInSeq

Parameters:

seqs - all sequences

seqList - a string array of all sequences

window - width of nmer

GetRandomNmerInSeq

SiteWin* **GetRandomNmerInSeq**(Sequence* seq, int window)

gets a random nmer in a sequence

Parameters:

seq - input sequence

window - width of nmer

GetRandomNmers

SiteList* **GetRandomNmers**(HashTable* seqs, dastr* seqList, int window, int n)

Gets a bunch of random nmers

calls GetRandomNmer

UCSF LIBRARY

Parameters:

seqs - all sequences
seqList - a string array of all sequences
window - width of nmer
n - number of nmers to obtain

GetRandomSeqList

dastr* **GetRandomSeqList**(int num)

gets a random set of gene ids

Parameters:

num - number of gene ids

LoadSequences

HashTable* **LoadSequences**(char* fastaFile, int num)

Loads a group of sequences from a fasta file

stores as a hash table of sequences

Parameters:

fastaFile - filename of fasta sequences
num - number of sequences to load, 0 for all

SaveFastaSequences

void **SaveFastaSequences**(dastr* seqList, char* filename)

Saves sequences to file in standard fasta format

Parameters:

seqList - a string array of all sequences
filename - file to save to

SaveSequencesRestrictedFasta

void **SaveSequencesRestrictedFasta**(dastr* seqList)

Saves sequences to OutF in Restricted Fasta format, compatible with Bioprospector

Parameters:

seqList - a string array of all sequences

Seq_CalcGCCContent

double **Seq_CalcGCCContent**(Sequence* seq)

calculates GC content of a sequence

Parameters:

seq - sequence

WhackSites

void **WhackSites**(SiteList* sl)

takes binding sites and marks Ns in the original genomic sequence

Parameters:

sl - list of binding sites

B.2.18. sitelist.c

pseudo-class of a list of binding sites

embeds a dynamic array and a hash so that you can arbitrarily add sites but also look up binding sites quickly

Uses the SiteWin to handle individual sites

Function Summary	
int	CompareSiteListExpression (const void* a, const void* b) Comparison of expression of SiteLists, higher first
int	CompareSiteListExpressionR (const void* a, const void* b) Comparison of expression of SiteLists, lower first
int	CompareSiteListPval (const void* a, const void* b) Comparison of pval of SiteLists, higher first
int	CompareSiteListScore (const void* a, const void* b) Comparison of score of SiteLists, higher first

int	CompareSiteListScore2 (const void* a, const void* b) Comparison of score2 of SiteLists, higher first
int	CompareSiteListScore2R (const void* a, const void* b) Comparison of score2 of SiteLists, lower first
int	CompareSiteListScoreR (const void* a, const void* b) Comparison of score of SiteLists, lower first
int	CompareSiteWinScore (const void* a, const void* b) Comparison of SiteWins, higher first
int	CompareSiteWinScore2 (const void* a, const void* b) Comparison of score2 of SiteWins, higher first
int	CompareSiteWinScore2R (const void* a, const void* b) Comparison of score2 of SiteWins, lower first
int	CompareSiteWinScoreR (const void* a, const void* b) Comparison of SiteWins, lower first
void	SetLocationKey (char* key, SiteWin* sw) sets key to represent location information of a site using a SiteWin
void	SetLocationKeyValues (char* key, char* id, int start, int compl) sets key to represent location information of a site
int	SiteList_Add (SiteList* d, SiteWin* val) add a SiteWin to the SiteList
SiteList*	SiteList_Copy (SiteList* d, int incrSize, int expectedSize) makes a shallow copy of a SiteList, setting parameters of new SiteList
SiteList*	SiteList_DeepCopyBest (SiteList* d, int num, int incrSize, int expectedSize) deep copy the best (assuming SiteList is already sorted) SiteWins
SiteList*	SiteList_New (int incrSize, int expectedSize) creates a new SiteList

CompareSiteListExpression

int **CompareSiteListExpression**(const void* a, const void* b)

Comparison of expression of SiteLists, higher first

Parameters:

a - SiteList 1

b - SiteList 2

CompareSiteListExpressionR

int **CompareSiteListExpressionR**(const void* a, const void* b)

Comparison of expression of SiteLists, lower first

Parameters:

a - SiteList 1
b - SiteList 2

CompareSiteListPval

int **CompareSiteListPval**(const void* a, const void* b)

Comparison of pval of SiteLists, higher first

Parameters:

a - SiteList 1
b - SiteList 2

CompareSiteListScore

int **CompareSiteListScore**(const void* a, const void* b)

Comparison of score of SiteLists, higher first

Parameters:

a - SiteList 1
b - SiteList 2

CompareSiteListScore2

int **CompareSiteListScore2**(const void* a, const void* b)

Comparison of score2 of SiteLists, higher first

Parameters:

a - SiteList 1
b - SiteList 2

CompareSiteListScore2R

int **CompareSiteListScore2R**(const void* a, const void* b)

Comparison of score2 of SiteLists, lower first

Parameters:

a - SiteList 1
b - SiteList 2

CompareSiteListScoreR

int **CompareSiteListScoreR**(const void* a, const void* b)

Comparison of score of SiteLists, lower first

Parameters:

a - SiteList 1
b - SiteList 2

CompareSiteWinScore

int **CompareSiteWinScore**(const void* a, const void* b)

Comparison of SiteWins, higher first

Parameters:

a - SiteWin 1
b - SiteWin 2

CompareSiteWinScore2

int **CompareSiteWinScore2**(const void* a, const void* b)

Comparison of score2 of SiteWins, higher first

Parameters:

a - SiteWin 1
b - SiteWin 2

CompareSiteWinScore2R

int **CompareSiteWinScore2R**(const void* a, const void* b)

Comparison of score2 of SiteWins, lower first

Parameters:

a - SiteWin 1
b - SiteWin 2

CompareSiteWinScoreR

```
int CompareSiteWinScoreR(const void* a, const void* b)
```

Comparison of SiteWins, lower first

Parameters:

a - SiteWin 1
b - SiteWin 2

SetLocationKey

```
void SetLocationKey(char* key, SiteWin* sw)
```

ets key to represent location information of a site using a SiteWin

Parameters:

key - output key
sw - location contained in SiteWin

SetLocationKeyValues

```
void SetLocationKeyValues(char* key, char* id, int start, int compl)
```

sets key to represent location information of a site

Parameters:

key - output key
id - id
start - start
compl - strand

SiteList_Add

```
int SiteList_Add(SiteList* d, SiteWin* val)
```

add a SiteWin to the SiteList

Parameters:

d - SiteList object
val - SiteWin to add

SiteList_Copy

SiteList* **SiteList_Copy**(SiteList* d, int incrSize, int expectedSize)

makes a shallow copy of a SiteList, setting parameters of new SiteList

Parameters:

d - SiteList object
incrSize - increment size of the new SiteList
expectedSize - expected size of the new SiteList

SiteList_DeepCopyBest

SiteList* **SiteList_DeepCopyBest**(SiteList* d, int num, int incrSize, int expectedSize)

deep copy the best (assuming SiteList is already sorted) SiteWins

Parameters:

d - SiteList object
num - number to copy
incrSize - increment size of the new SiteList
expectedSize - expected size of the new SiteList

SiteList_New

SiteList* **SiteList_New**(int incrSize, int expectedSize)

creates a new SiteList

often incrSize==expectedSize

Parameters:

incrSize - increment size of dynamic array
expectedSize - expected size of hash table

B.2.19. *utils.c*

library of utility functions

Function Summary	
void	<u>ChangeToReverseComplement</u> (char* seq) change a seq to its reverse complement in place
int	<u>CharValid</u> (char nuc) check if nucleotide is valid
void	<u>DNA2Seq</u> (char* frag, int len, unsigned int dna) converts DNA number to sequence
unsigned int	<u>DNArc</u> (int dna, int len) calculates reverse complement of DNA number
int	<u>Db1Cmp</u> (const void* a, const void* b) compares two doubles
char**	<u>DestructiveSplit</u> (char* s) splits string in place looking for tabs
char**	<u>DestructiveSplit2</u> (char* s) splits on ' ' and '\t'
char**	<u>DestructiveSplitOnChar</u> (char* s, char c) generic split on specified char
	<u>ErrExit</u> () A generic fatal error-handler
	<u>Initb2i</u> () initializes b2i data structure
int	<u>IntCmp</u> (const void* inta, const void* intb) compares two integers
int	<u>LogF</u> (const char* fmt, ...) printf like function
int	<u>NumLinesInFile</u> (char* filename) counts number of lines in file
int	<u>OutF</u> (const char* fmt, ...) printf like function
double	<u>PearsonsCoeff</u> (double* x, double* y, int n) calculates pearson's coefficient of two arrays
char*	<u>ReadFasta</u> (FILE* f, char* desc) reads a single fasta entry from an open file
char*	<u>ReverseString</u> (char* s) returns a new string that is the reverse of the original
void	<u>ReverseStringInPlace</u> (char* s) reverse string in place
unsigned int	<u>Seq2DNA</u> (char* frag, int len) converts sequence into DNA format number
unsigned int	<u>Seq2DNAFast</u> (char* frag, int len) fast version of Seq2DNA
int	<u>SeqValid</u> (char* frag)

	check if sequence has legit nucleotides
void	SetF (FILE* f) set file pointer to be used in OutF
void	SetLog (FILE* f) set file pointer to be used in LogF
	StandardDeviation () Returns the Standard Deviation of x[0]..x[n-1]
void	StreamFasta (char* seq, int lineLength, int (*printfn)(const char* format, ...)) writes to function pointer with line breaks
int	SubSeqValid (char* frag, int len) checks if sub-sequence has legit nucleotides
void	SwitchToComplement (char* s) converts string to complement in place
void	WriteFasta (char* seq, int lineLength, FILE* f) writes sequence with line breaks
	cant () An fopen() replacement with error trapping
void	chomp (char* str) removes newline and carriage return from string
	flength () determines the size of a file
	genrand () generates one pseudorandom real number (double)
double	k_tau (double *actual, double *predicted, int n, double delta1, double delta2) calculate Kendall's Tau
void	lc (char* str) converts string in place to lowercase
double	mean (double* x, int n) computes mean of array
void	set_nuc (unsigned int* dna, int pos, int val) sets nucleotide in DNA representation
	sgenrand () seeds random number generator
double	ttest (double* x, double* y, int n1, int n2) computes t-test
	ttestprobability () calculates the probability for a two tailed test
void	uc (char* str) converts string in place to uppercase

ChangeToReverseComplement

void **ChangeToReverseComplement**(char* seq)

change a seq to its reverse complement in place

Parameters:

seq - input sequence

CharValid

int **CharValid**(char nuc)

check if nucleotide is valid

Parameters:

nuc - nucleotide letter

DNA2Seq

void **DNA2Seq**(char* frag, int len, unsigned int dna)

converts DNA number to sequence

doesn't check for valid index

Parameters:

frag - output sequence

len - length of DNA number

dna - input DNA number

DNArc

unsigned int **DNArc**(int dna, int len)

calculates reverse complement of DNA number

Parameters:

dna - sequence in DNA representation

len - length of DNA seq

DblCmp

int **DblCmp**(const void* a, const void* b)

compares two doubles

Parameters:

a - double 1
b - double 2

DestructiveSplit

char** **DestructiveSplit**(char* s)

splits string in place looking for tabs

similar to perl function

Parameters:

s - input string

DestructiveSplit2

char** **DestructiveSplit2**(char* s)

splits on ' ' and '\t'

Parameters:

s - input string

DestructiveSplitOnChar

char** **DestructiveSplitOnChar**(char* s, char c)

generic split on specified char

see also [DestructiveSplit](#)

Parameters:

s - input string
c - split using this char

ErrExit

ErrExit()

A generic fatal error-handler

by Dave Schaumann

Parameters:

fmt - format of output

Initb2i

`Initb2i()`

initializes b2i data structure

b2i contains array for quick lookup from nucleotide to index

IntCmp

`int IntCmp(const void* inta, const void* intb)`

compares two integers

Parameters:

inta - integer 1

intb - integer 2

LogF

`int LogF(const char* fmt, ...)`

printf like function

prints to file pointer designated by `SetLog`

Parameters:

fmt - format of output

NumLinesInFile

`int NumLinesInFile(char* filename)`

counts number of lines in file

Parameters:

filename - input filename

OutF

```
int OutF(const char* fmt, ...)
```

printf like function

prints to file pointer designated by SetF.

This was used so there could be a globally accessible location to write output.

in retrospect, I could have defined my libraries more carefully and gotten around this problem.

In principle, it is similar to C++ streaming of files

Parameters:

fmt - format of output

PearsonsCoeff

```
double PearsonsCoeff(double* x, double* y, int n)
```

calculates pearson's coefficient of two arrays

Parameters:

x - array of doubles

y - array of doubles

n - length of each array

ReadFasta

```
char* ReadFasta(FILE* f, char* desc)
```

reads a single fasta entry from an open file

Parameters:

f - input file pointer

desc - output description on first line

ReverseString

char* **ReverseString**(char* s)

returns a new string that is the reverse of the original

Parameters:

s - input string

ReverseStringInPlace

void **ReverseStringInPlace**(char* s)

reverse string in place

Parameters:

s - input string

Seq2DNA

unsigned int **Seq2DNA**(char* frag, int len)

converts sequence into DNA format number

no check for valid sequence!!! so make sure beforehand

Parameters:

frag - sequence to convert

len - length of sequence

Seq2DNAFast

unsigned int **Seq2DNAFast**(char* frag, int len)

fast version of Seq2DNA

Parameters:

frag - sequence to convert

len - length of sequence

SeqValid

int **SeqValid**(char* frag)

check if sequence has legit nucleotides

Parameters:

frag - input sequence

SetF

void **SetF**(FILE* f)

set file pointer to be used in OutF

See also OutF

Parameters:

f - input file pointer

SetLog

void **SetLog**(FILE* f)

set file pointer to be used in LogF

See also LogF

Parameters:

f - input file pointer

StandardDeviation

StandardDeviation()

Returns the Standard Deviation of x[0]..x[n-1]

Returns '100.0' if n<=1

Parameters:

x - array of doubles

n - size of x

StreamFasta

```
void StreamFasta(char* seq, int lineLength, int (*printfn)(const char*  
format, ...))
```

writes to function pointer with line breaks

similar to WriteFasta, except uses function pointer

Parameters:

seq - sequence to output

lineLength - length per line

printfn - output function pointer, like OutF

SubSeqValid

```
int SubSeqValid(char* frag, int len)
```

checks if sub-sequence has legit nucleotides

Parameters:

frag - input sequence

len - length of input sequence

SwitchToComplement

```
void SwitchToComplement(char* s)
```

converts string to complement in place

Parameters:

s - input string

WriteFasta

```
void WriteFasta(char* seq, int lineLength, FILE* f)
```

writes sequence with line breaks

Parameters:

seq - sequence to output
lineLength - length per line
f - output file pointer

cant

`cant()`

An fopen() replacement with error trapping

public domain by Bob Stout

Parameters:

fname - filename to open
fmode - file mode, e.g. "w", "r", etc.

chomp

void `chomp`(char* str)

removes newline and carriage return from string

similar to perl version

Parameters:

str - input string

flength

`flength()`

determines the size of a file

a simple function using all ANSI-standard functions

Public domain by Bob Jarvis.

Parameters:

fname - filename

genrand

genrand()

generates one pseudorandom real number (double)

which is uniformly distributed on [0,1]-interval, for each call

k_tau

double **k_tau**(double *actual, double *predicted, int n, double delta1,
double delta2)

calculate Kendall's Tau

with modification for deltas to specify equivalence for close real values. Use 0.0 for
deltas for standard Tau definition

got this from Ajay

Parameters:

actual - first array
predicted - second array
delta1 -
delta2 -

lc

void **lc**(char* str)

converts string in place to lowercase

Parameters:

str - input string

mean

double **mean**(double* x, int n)

computes mean of array

see code for attribution

Parameters:

x - array of doubles
n - size of x

set_nuc

void **set_nuc**(unsigned int* dna, int pos, int val)

sets nucleotide in DNA representation

Parameters:

dna - input DNA number
pos - position to change
val - value to change to

sgenrand

sgenrand()

seeds random number generator

set initial values to the working area of 624 words. Before `genrand()`, `sgenrand(seed)` must be called once. (seed is any 32-bit integer except for 0).

LGPL, see code for details

Parameters:

seed - seed for random number generator

ttest

double **ttest**(double* x, double* y, int n1, int n2)

computes t-test

see code for attribution

Parameters:

x - array of doubles
y - array of doubles

n1 - size of x
n2 - size of y

ttestprobability

ttestprobability()

calculates the probability for a two tailed test

see code for attribution

Parameters:

t - input t value
df - degrees of freedom

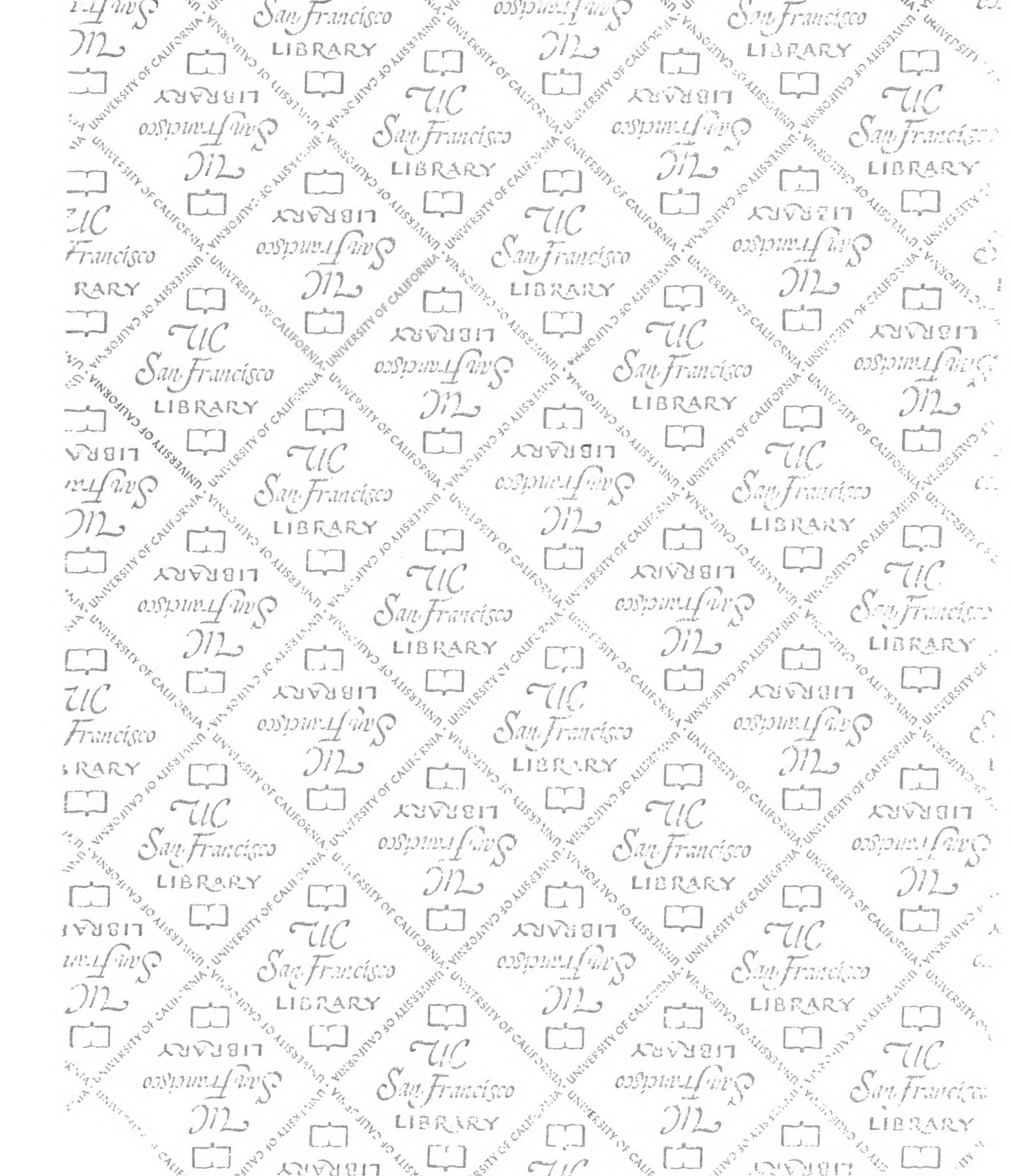
uc

void **uc**(char* str)

converts string in place to uppercase

Parameters:

str - input string



San Francisco
LIBRARY

San Francisco
LIBRARY

7487112



3 1378 00748 7112

For reference

Not to be taken
from the room.

