U N I V E R S I T Y OF T A R T U
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Timo Petmanson

Mining Motifs in DNA Regulatory Area

Barchelor's Thesis (6 ECTS)

Supervisor: Sven Laur, D.Sc. (Tech.)

Author: .... "......" May 2010
Supervisor: .... "......" May 2010
Chairman: .... "......" 2010

TARTU 2010

Contents

Introduction

All living organisms on earth are believed to contain genetic information coded in structured collections of genes and non-coding sequences that make up the DNA. The coded information is used to build organisms, maintain them and it defines a wide range of genetic features that vary from individuals to individuals and from species to species. The non-coding parts have much of the responsibility to regulate the expression of particular genes. Genes with their non-coding regulatory areas form complex signaling networks that together coordinate the life cycle of an organism. Contemporary methods in genetics like ChIP and micro-array measurements make it possible to measure features of thousands of genes in one experiment, generating huge amounts of data. Therefore, the development of new algorithms and methods able to analyze this data is crucial.

Our contributions include the development of novel methods able to combine different sources of experimental data. In Chapter 2, we formalize the theory describing sequence mining with multiple input sequences and multiple data layers. We also describe, how to determine statistically significant motifs using our theory. In Chapter 3, we develop algorithms MaxSupSearch, SafeApproxSearch, InfreqSearch, GFPSearch, that utilize different pruning strategies. For GFPSearch, we define generic-frequent-pattern tree structure that is a generalization of FP-tree [JJYR04]. We also develop NBest, that combines any previously mentioned algorithm with binary search to get fixed number of best motifs. We develop SigMotifs, that goes even further by distilling out statistically significant motifs. Performance study of mentioned algorithms along with experiments on real biological data are given in Chapter 4.

Chapter 1

Preliminaries

1.1 DNA

Currently scientists have described about 1.5 million different species: about five thousand mammals, thirty thousand species of fish and over nine hundred thousand insects among others [WCU07]. Some estimates of comparing samples from various parts of the world seas suggest that in oceans there may be more than 100 million species of bacteria [MHJ06]. This vast diversity of known and unknown species in Earth's biosphere are believed to have one thing in common: the presence of DNA.

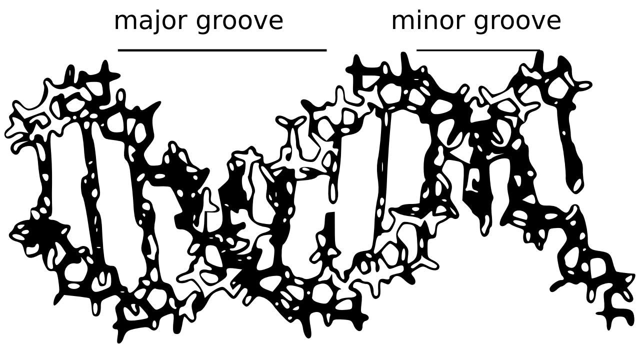


Figure 1.1: DNA Double Helix. The distance between strands varies and forms major and minor grooves.

Chemically DNA is consist of two long strands of polymers, where the backbone of a strand contains alternating phosphate and sugar residues linked with bases. These two strands form a structure known as double helix seen in Figure 1.1, whose stability is maintained by hydrogen bonds between the bases, see Figure 1.2 [RSM05]. There are four types of bases in DNA: *adenine* (abbreviated A), *thymine* (T), *guanine* (G) and *cytosine* (C) that combined with a sugar and one or more phosphate residues form a nucleotide. The *nucleotides* are pairwise aligned, making the structure anti parallel, where adenine bonds only to guanine and cytosine bonds only to thymine. The endpoints of the strands are called 3' and 5' where the first is defined by a terminal phosphate group and the second by a terminal hydroxyl group [Coh04].

DNA nucleotide sequences are usually written only using bases from one strand as the bases on other strand are complementary. Sequence `TATAAA` is complementary to `ATATTT` for example. The order the characters are written depends on the source of the data – sometimes the data is written in direction from 3' to 5' while others are vice versa.

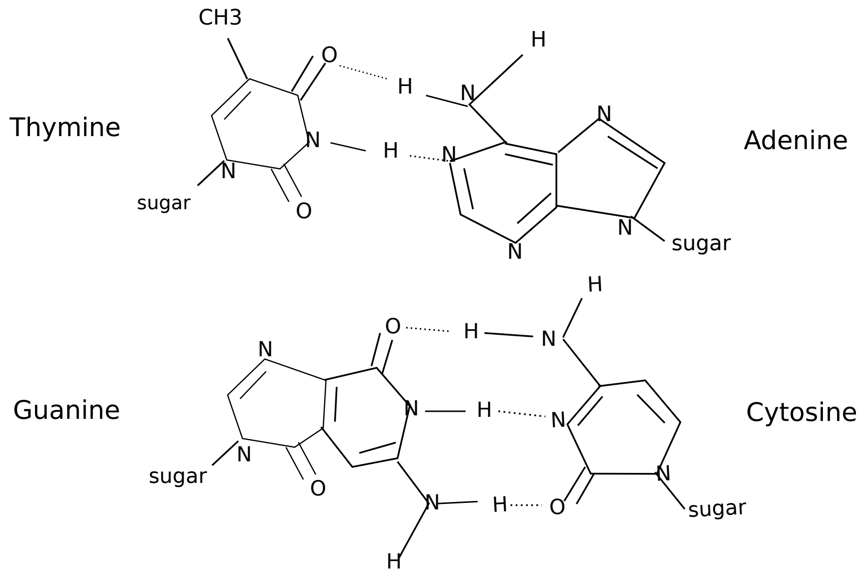


Figure 1.2: TA and GC complementary base pairs. Dotted lines represent hydrogen bonds between bases

1.2 Gene expression

Gene expression means the rate and amount of RNA transcribed from it, which in turn is used to define other proteins necessary for the cell and the organism. The transcription process requires transcription factors that are special proteins able to recognize and attach to particular fragments in gene promoter areas. The transcription factors are required to recruit RNA polymerase that is responsible for carrying out the transcription process.

In more complex eukaryotic cells, the promoters are rather diverse and complicated, but the core elements are a transcription start site, which together with RNA polymerase and transcription factor binding sites are essential for initiating the transcription process. Other important binding sites are typically a little more far away in upstream direction that mainly regulate gene expression by enhancing or restricting recruitment of the main transcription factors. Additionally, there may be even more distant promoter areas that have weaker influence on the gene regulation.

1.3 Data mining

Data mining is a method in statistics for extracting interesting patterns or knowledge from large amounts of available data. This field is very diverse as among general data mining solutions there are many specific procedures developed for business, games, social networks *et cetera* [DP07]. In this work, we concentrate on specialized area of data mining called *sequence mining* that deals with ordered sequences like nucleotide sequences.

The Apriori algorithm is the most general and simple way to find patterns with high support in given data. In standard sequence mining, the *support* is defined as the number of occurrences of a pattern in input data, which is used to decide whether the pattern is frequent or infrequent based on some defined threshold. The Apriori algorithm assumes that the support is downward closed, which means that for any infrequent pattern there do not exist any frequent sup-patterns. For example, a DNA motif `AAATCCC` cannot be present in data more times than sequences `AAA` and `CCC`, because whenever the supmotif occurs, the two submotifs must also occur. Let us clarify, that in this work by a submotif or a subpattern we mean a subsequence with consecutive elements.

Algorithm 1.3.1 The Apriori algorithm.

```
1: $F_1 \leftarrow$ {Frequent one-element patterns}
2: $\ell \leftarrow 2$
3: while $F_{\ell-1} \neq \emptyset$ do
4:     $C_\ell \leftarrow$ GenerateCandidates($F_{\ell-1}$)
5:     $F_\ell \leftarrow \{c \in C_\ell \mid \mathrm{supp}(c) \geqslant \sigma\}$          ▷ $\sigma$ is threshold
6:     $\ell \leftarrow \ell + 1$
7: end while
```

The Apriori algorithm uses downward closeness as a main pruning feature. In Algorithm 1.3.1 on line 4, the GenerateCandidates procedure takes the set of frequent motifs of length $\ell - 1$ as input and generates possible candidates of length ℓ. It does not need to consider any non-frequent motifs as none of their supmotifs are frequent. The algorithm stops running when it has found all frequent motifs in the dataset.

Let us demonstrate Apriori by giving an example. Consider the following sequence:

$$\texttt{GCTTATGGTCGCTATGCTTT} \ .$$

Suppose we want to mine all motifs occurring at least three times in the sequence. This means that we run Apriori with threshold $\sigma = 3$. The set $F_1 = \{\texttt{T}, \texttt{G}, \texttt{C}\}$, because all nucleotides except `A` are present in sequence more than three times. Next, we generate candidate motifs of length two by using only frequent elements in F_1.

$$C_2 = \{\texttt{TT, TG, TC, GT, GG, GC, CT, CG, CC}\}$$

Frequent motifs in this case are

$$F_2 = \{\texttt{TT, GC, CT}\} \ .$$

Note that `TT` matches `TTT` two times. The next candidate set is

$$C_3 = \{\texttt{TTT, GTT, TTG, CTT, TTC, TGC, GCT, GGC, GCG,}$$
$$\texttt{CGC, GCC, TCT, CTT, GCT, CTG, CCT, CTC}\} \ .$$

This time there is only one frequent motif:

$$F_3 = \{\texttt{GCT}\} \ .$$

Candidate motifs of length 4:

$$C_4 = \{\texttt{TGCT, GCTT, GGCT, GCTG, CGCT, GCTC}\} \ .$$

But none of them is frequent, so $F_4 = \emptyset$ and all frequent motifs in our example are

$$F = \{\texttt{T, G, C, TT, GC, CT, GCT}\} \ .$$

There are also algorithms like WINEPI [MTV95], MINEPI [MT96], SPEXS [Vil02] that are able to mine motifs using pattern matching. Still, while APRIORI with other standard sequence mining algorithms are useful, they treat all parts of the sequence with equal weight. In our case, we need methods that are able to work with data that decorates sequences with scores, making some parts of them more relevant than the rest. In Chapter 2, we reformulate standard sequence mining techniques and later devise our own algorithms that handle such requirements.

Chapter 2

Sequence Mining with Multiple Layers of Data

In this chapter, we formalize basic notions and concepts like sequences, motifs and support that are needed to develop our methods. We try to develop our mathematical approach such that it would be convenient to study gene regulation, when we consider several promoter areas and different properties of these sequences described by layers of experimental data.

We also study different properties and relations between these building blocks that are later used in algorithms to cut down the running times and improve overall performance, although we do not cover algorithmic details and other aspects like data structures as they are discussed in later chapters.

2.1 Sequences and Scores

The most basic constructs we will be dealing onwards are DNA sequences and their fragments. In our case, it will be convenient to think of them as a set of nucleotide sequences. *Let $\mathcal{S} = \{a, b, c, \ldots\}$ denote a set of promoter sequences relevant to some gene.* Single elements of a sequence are denoted with subscripts as usual. For example, a_1 means the first element and a_2 the second element of $a \in \mathcal{S}$. As there are four types of nucleotides *adenine*, *thymine*, *cytosine*, *guanine* in DNA that correspond to letters `A`, `T`, `C`, `G`. We write $a_1 = \mathtt{A}$, if first element in the nucleotide sequence is *adenine* and $a_2 = \mathtt{T}$, if the second element is *thymine*. Let us denote the length of sequence a as

$|a|$. It is worth to note that no promoter is with length of zero, nor there are promoters with infinite length in real world. However, depending on particular case, the lengths of the sequences are not usually very short or very long.

In mathematics, a fragment of a sequence is usually written as a list of elements. In this paper, we will be using a shorter notation:

$$a_{i\,:\,j} \overset{\text{def}}{=} a_i, a_{i+1}, \ldots, a_j \ .$$

where i is the beginning and j is the end of the fragment.

We stated in the introduction of this thesis that we are going to deal with multiple layers of data about promoter sequences. For example, if we have data containing *binding* and *conservation* scores from DNA micro-array and sequencing experiments that associate with promoters we are interested in, we can portray them as data tracks over the nucleotide sequence as illustrated in Figure 2.1.

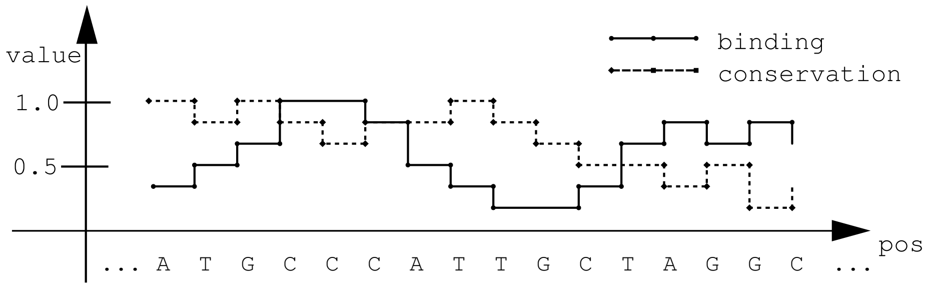


Figure 2.1: An example subsequence having conservation and binding data tracks attached. The scores are variable and may not directly depend on each other.

From theoretical point of view, it is not important exactly what kind of data we have, as long we can represent it as numeric values linked to positions in promoter sequences. However, it is important that these values express some property that makes some regions of the nucleotide sequence more relevant than other regions, thus defining important regions in respect to each data track. If we have n data sets containing various scores and m promoters, then we need $n \times m$ mappings that associate relevant scores from a data set to all positions in nucleotide sequences. Also, it is convenient to normalize all data such that all scores fall into range $[0, 1]$ like shown on Figure 2.1. It simplifies writing some formulas, because we know the maximum possible value of any type of score linked to any position of a nucleotide sequence. *Let* $\varphi : \mathbb{N} \longrightarrow \mathbb{R}$ *be a mapping that associates numeric scores to all positions of a nucleotide sequence.* To make this notation more useful, let us agree that by writing $\varphi(a_i)$ we mean the score that φ maps to position i of sequence a and by writing $\varphi(a_{i\,:\,j})$, we mean a sequence of scores $\varphi(a_{i\,:\,j}) \overset{\text{def}}{=} \varphi(a_i), \varphi(a_{i+1}), \ldots, \varphi(a_j)$. By writing $\overline{\varphi}(a_{i\,:\,j})$ we mean the average score

$$\overline{\varphi}(a_{i\,:\,j}) \overset{\text{def}}{=} \frac{1}{j-i+1} \cdot \sum_{k=i}^{j} \varphi(a_k) \ .$$

2.2 Motifs and Matching

In this section, we introduce motifs, which can be thought of as possible subsequences in sequence set. Motifs do not directly associate to any data track, but there are several other metrics like support, frequency, significance of a motif in a particular set of promoter sequences. In addition to nucleotide letters `A, T, G, C`, motifs may also contain special wild card characters that have special meaning and usage. In this work, we will be using only one such symbol `*` that represents any possible nucleotide in one position. Note that this is different from standard usage of this symbol in batch-processing or regular-expression applications where it usually stands for zero or more symbols. In our case, if we have a motif `G**A`, then by that we mean any motif with length of four that starts with letter `G` and ends with letter `A`.

We will be dealing a lot with fixed-length motifs in later sections, so it is necessary to introduce notation that we can use to refer to all motifs with a fixed length ℓ. *Let* $\mathcal{M}_\ell$ *represent a set of all motifs with length* ℓ *where* $\ell \in \mathbb{N}$. We agreed before, that all motifs are consist of five different letters: the nucleotides and the wild card character. This means that the cardinality of the set $\mathcal{M}_\ell$ is equal to $|\mathcal{M}_\ell| = 5^\ell$ as there are five different possible elements per position in a motif.

Often it is necessary, that we could refer to single elements of a motif the same way we do for sequences, so given any motif $m \in \mathcal{M}_\ell$, let m_1 denote the first element of the motif, m_2 the second element of the motif *et cetera.* In addition to that, it is convenient to describe motifs as concatenation of shorter motifs. In our case, it is useful to think of a motif as a concatenation of only a prefix and suffix part. *Let* $||$ *be an concatenation operator. If* $m^p \in M_p$ *and* $m^s \in M_s$ *then motif* $m = m^p \mid\mid m^s$, *where* $m \in M_\ell$ *and* $\ell = p + s$. Let us illustrate this with an example. If $m^p =$ `AAAT` and $m^s =$ `GCCGT`, then the concatenation $m^p \mid\mid m^s$ is `AAATGCCGT`.

Another very useful notion is a wild card extension of some motif. Namely, if we have some fixed motif length ℓ and a motif $m \in M_k$, such that $k \leqslant \ell$, we may pad the motif with wild card characters until it is ℓ elements long. This enables to easily express motifs we know to have a certain prefix. *Let* $m^* \in \mathcal{M}_\ell$ *denote a wild card character extension of motif* $m \in \mathcal{M}_k$ *where* $k \leqslant \ell$ *such that the prefix* $m^*_{1\,:\,k} = m$ *and suffix* $m^*_{k+1\,:\,\ell} = * \ldots *$. For instance, if $m =$ `AATA` and we have fixed motif length $\ell = 10$, then the *wild character extension* $m^* =$ `AATA******`. This notion comes handy when we describe SafeApproxSearch algorithm in Chapter 3. Let us agree that any motif gained from another motif by replacing one or more nucleotides with wild cards is considered a submotif of the original motif.

In standard sequence mining, the support of some motif is usually measured by how many matches it has in data [DP07]. The number of matches of a motif containing no wild card characters is simply the number of times the motif can be viewed as a subsequence of given data sequence. With wild card characters this works different as a wild card character matches any nucleotide. See Figure 2.2 for an illustration.

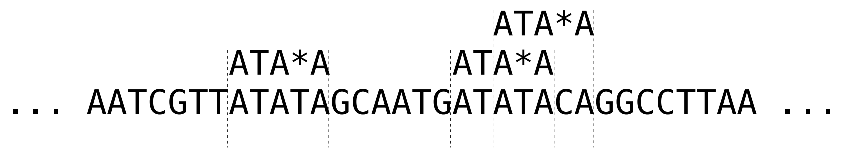


Figure 2.2: Three mathes of motif `ATA*A` in a subsequence.

Definition 2.2.1 *A motif* $m \in \mathcal{M}_\ell$ *matches some fragment* $a_{i\,:\,i+\ell-1}$ *of sequence* a, *if*

$$m_k = * \;\vee\; m_k = a_{i+k-1} \qquad \text{for all} \qquad k = 1, \ldots, \ell \;.$$

Let us denote it as following:

$$\mathsf{match}(a, m, i) = \begin{cases} 1, & \textit{if } m \textit{ matches } a_{i\,:\,i+\ell-1} \\ 0, & \textit{otherwise.} \end{cases}$$

We can extend the number of matches in the sequence a over a set of sequences $\mathcal{S}$ by simply adding all the individual counts together:

$$\begin{aligned} \mathsf{mcount}(a, m) &\overset{\text{def}}{=} \textstyle\sum_{i=1}^{|a|-\ell+1} \mathsf{match}(m, a_i) \\ \mathsf{mcount}(\mathcal{S}, m) &\overset{\text{def}}{=} \textstyle\sum_{s\,\in\,\mathcal{S}} \mathsf{mcount}(m, s) \quad . \end{aligned}$$

2.3 Support Metrics

Standard sequence mining treats all parts of the input sequence with equal value of importance [DP07]. In our case, we have possibly more than one data tracks containing variable scores. Therefore, we need to define support in a different way. We base our approach on a formulation given by Sven Laur [Lau09].

The first thing is to extend the notion of support of one single match. Standard way was summing up all matches of a motif in a sequence, such that each match had equal value of importance. But as we have actual scores linked to positions, we extend the original method by taking an average score of matching positions of a single match.

Definition 2.3.1 *The support of an individual motif $m \in \mathcal{M}_\ell$ with respect to some fragment in sequence a starting from position i:*

$$\mathrm{supp}(a, m, i) = \begin{cases} \overline{\varphi}(a_{i\,:\,i+\ell-1}) & \textit{if } \mathsf{match}(a, m, i) = 1 \\ 0 & \textit{otherwise} \quad . \end{cases}$$

To extend the support of a motif over a sequence, we have several options. The first idea is to add up all the single supports of the motif. This is the simplest way to go and we refer to this method as *additive* support onwards. Let us consider another option: instead of adding up the scores, we can take only the maximal score and be fine with it. The plus side of this method is that it promotes motifs that actually have high scores. Additive support can be high even if all the scores of the single matches are low. So, we also consider this method and we will be referring to it as *maximal* support.

Of course, there are more ways to express the support of some motif in a sequence. We might consider average support that works like additive support, but we divide the result by number of matches of that motif in the sequence. We could also define supports like weighted additive or weighted average support, that considers some regions of the promoter to be more significant than others. The last two are actually not very reasonable, because we express significance of promoter areas through data tracks anyway.

The average support is actually more relevant, but as it seems to have a mixed properties of additive and maximal support, we do not cover this type of support in this work and concentrate on studying only the two mentioned support types.

Definition 2.3.2 *Additive support of a motif $m \in \mathcal{M}_\ell$ in sequence a is*

$$\operatorname{asupp}(a, m) = \sum_{i=1}^{|a|-\ell+1} \operatorname{supp}(a, m, i) \quad .$$

Definition 2.3.3 *Maximal support of a motif $m \in \mathcal{M}_\ell$ in sequence a is*

$$\operatorname{msupp}(a, m) = \max \{ \operatorname{supp}(a, m, i) \mid i = 1, \ldots, |a| - \ell + 1 \} \quad .$$

By writing $\operatorname{supp}(a, m)$, we do not refer directly to neither of the support types in cases we are discussing properties that apply to both of them.

Therefore, Definitions 2.3.2 and 2.3.3 are only two possible ways of expressing the support of a motif in one sequence. Biological importance of the two depends mostly on the actual data used. For example, if we consider conservation, then *additive* support can reveal motifs that coexist in several genetically close species having great structural importance, *maximal* support takes into account only one occurrence of a motif in a promoter, thus ignoring larger scale structural effects. On the other hand, *maximal* support can bring up motifs that are recognized most probably by *transcription factors* as these enzymes require proper locations to enable mounting RNA polymerase and initiate transcription. Thus, the decision about what support type should be used with a particular data track, depends on biological properties the data.

To extend notion of support over a set of sequences, we have several options. The first approach is to consider all promoter sequences having equal impact on the importance of a motif. We can achieve this by taking an average of support of a motif in all sequences.

Definition 2.3.4 *Additive and maximal support of motif $m \in \mathcal{M}_\ell$ in a list of sequences $\mathcal{S}$ are*

$$\mathrm{asupp}(\mathcal{S}, m) = \frac{1}{|\mathcal{S}|} \cdot \sum_{a \in \mathcal{S}} \mathrm{asupp}(a, m) \tag{2.1}$$

$$\mathrm{msupp}(\mathcal{S}, m) = \frac{1}{|\mathcal{S}|} \cdot \sum_{a \in \mathcal{S}} \mathrm{msupp}(a, m) \ . \tag{2.2}$$

The second approach is to consider promoters further away from the gene they regulate having less impact than the ones closer to it. Therefore, we need to give promoter sequences meaningful weights, when calculating support. We could propagate these weights directly into the datasets, enabling the direct use of Equations (2.1) and (2.2). Let us also agree that by writing $\mathrm{supp}(\mathcal{S}, m)$, we do not refer directly to additive, nor maximal support if we are discussing properties that apply to both of them.

In Chapter 3, we discuss algorithms and data structures and usually need support in respect to all data tracks. Also, let us agree that we have fixed the support type for every data track to make semantics easier. In cases we need to use both support types, we can view original track as two duplicates with different support types.

Definition 2.3.5 *Given mappings $\varphi^1, \ldots, \varphi^n$, the support of motif m in sequences $\mathcal{S}$ in respect to all n data tracks is*

$$\overrightarrow{\mathrm{supp}}(\mathcal{S}, m) = (s_1, \ldots, s_n)$$

where

$$s_i = \begin{cases} \mathrm{asupp}(\mathcal{S}, m, \varphi_i) & \textit{for additive type of support} \\ \mathrm{msupp}(\mathcal{S}, m, \varphi_i) & \textit{for maximal type of support.} \end{cases}$$

Mappings $\varphi_1, \ldots, \varphi_n$ given as extra arguments to support operators will be used as the mapping φ in Definition 2.3.1.

2.4 Properties of Support Metrics

In previous section, we defined basic building blocks like sequences, motifs and mappings that gave each position in a promoter sequence one or more weights in regard to available data sets. In this section, we study various properties of newly defined support measures and notions.

When we compare *additive* and *maximal* support, it is rather easy to see that additive support is always as big as maximal support, because additive support considers all occurrences of a motif in a sequence where maximal support only considers the occurrence with maximal support.

Proposition 2.4.1 *For any motif $m \in \mathcal{M}_\ell$ and a set of sequences $\mathcal{S}$*

$$\mathrm{msupp}(\mathcal{S}, m) \leqslant \mathrm{asupp}(\mathcal{S}, m) \ .$$

We have not mentioned that there is a problem with the way we defined our support of some motif. Namely, the definition breaks the standard sequence mining principle of being downward closed as any non-frequent motif may have frequent supmotifs.

Claim 2.4.2 *Let $\sigma \in \mathbb{R}$ be the threshold. For any motif $m \in \mathcal{M}_\ell$, such that support $\mathrm{supp}(\mathcal{S}, m) < \sigma$, may exist a supmotif $m' \in \mathcal{M}_{\ell+k}$, such that $\mathrm{supp}(\mathcal{S}, m') \geqslant \sigma$, whether we consider additive or maximal support.*

Proof. For simplicity, let us assume that there is only one single match of motif m' in positions i to $i + \ell - 1$ in sequence a. Since the scores of all positions are in range $[0, 1]$, the $\mathrm{supp}(a, m, i) \in [0, 1]$. Now, let us consider a situation where support of the prefix $\mathrm{supp}(a, m, i) = \overline{\varphi}(a_{i\,:\,i+\ell-1}) < 1$ and support of the suffix $\overline{\varphi}(a_{\ell\,:\,\ell+k-1}) = 1$. From here we can conclude that

$$\mathrm{supp}(\mathcal{S}, m') = \frac{1}{|\mathcal{S}|} \cdot \frac{\varphi(a_i) + \ldots + \varphi(a_{i+\ell-1}) + k}{\ell + k} > supp(\mathcal{S}, m) \ . \qquad (2.3)$$

If we take $\sigma = supp(\mathcal{S}, m')$, then m is infrequent and m' is frequent.

□

Above proof raises another question: if we know the support of a motif, then what is the maximal possible support of any supmotif? We can approach the answer same way proved above claim. Namely, if we consider the support of the suffix of a possible supmotif to have maximal possible value, then we can calculate the maximal possible support of the supmotif.

Proposition 2.4.3 *For any motif $m' \in \mathcal{M}_{\ell+k}$ and its submotif $m \in \mathcal{M}_\ell$ in a set of sequences $\mathcal{S}$*

$$\mathrm{msupp}(\mathcal{S}, m') \leqslant \frac{\ell \cdot \mathrm{msupp}(\mathcal{S}, m) + |\mathcal{S}| \cdot k}{\ell + k}$$

$$\mathrm{asupp}(\mathcal{S}, m') \leqslant \frac{\ell \cdot \mathrm{asupp}(\mathcal{S}, m) + \mathsf{mcount}(\mathcal{S}, m) \cdot k}{\ell + k} \quad .$$

Proof. Let us consider a set of fragments $\{a_{p\,:\,q}, b_{r\,:\,s}, \ \ldots \ z_{t\,:\,u}\}$ that represent positions on every promoter where motif m has highest support. In that case

$$\mathrm{msupp}(\mathcal{S}, m) = |\mathcal{S}|^{-1} \left(\overline{\varphi}(a_{p\,:\,q}) + \overline{\varphi}(b_{r\,:\,s}) + \ldots + \overline{\varphi}(z_{t\,:\,u})\right) \quad .$$

If we now consider m as prefix of m', then analogous way to Equation (2.3) we can estimate that support of m' cannot be larger than

$$\mathrm{msupp}(\mathcal{S}, m') \leqslant \frac{1}{|\mathcal{S}|} \left(\frac{\varphi(a_{p\,:\,q}) + k}{\ell + k} + \frac{\varphi(b_{r\,:\,s}) + k}{\ell + k} + \ldots + \frac{\varphi(z_{t\,:\,u}) + k}{\ell + k} \right) =$$

$$= \frac{1}{|\mathcal{S}|} \cdot \frac{\varphi(a_{p\,:\,q}) + \varphi(b_{r\,:\,s}) + \ldots + \varphi(z_{t\,:\,u}) + |\mathcal{S}| \cdot k}{\ell + k}$$

that combined with Equation (2.2) becomes

$$\mathrm{msupp}(\mathcal{S}, m') \leqslant \frac{\ell \cdot \mathrm{msupp}(\mathcal{S}, m) + |\mathcal{S}| \cdot k}{\ell + k} \quad . \tag{2.4}$$

Additive support takes into account all occurrences of m, thus replacing the relevant parts in Equation (2.4), we get

$$\mathrm{asupp}(\mathcal{S}, m') \leqslant \frac{\ell \cdot \mathrm{asupp}(\mathcal{S}, m) + \mathsf{mcount}(\mathcal{S}, m) \cdot k}{\ell + k} \quad .$$

□

Claim 2.4.2 implies that frequent motifs may have infrequent submotifs. However, it is important to note that any frequent motif also must have at least one frequent submotif with either additive or maximal type of support.

Lemma 2.4.4 *For all positions $i < j \leqslant k$ in sequence a*

$$\overline{\varphi}(a_{i\,:\,k}) \leqslant \max\{\ \overline{\varphi}(a_{i\,:\,j-1}),\ \overline{\varphi}(a_{j\,:\,k})\ \}\ .$$

Proof. The first possibility is that $\overline{\varphi}(a_{i\,:\,j-1}) > \overline{\varphi}(a_{j\,:\,k})$ or $\overline{\varphi}(a_{i\,:\,j-1}) < \overline{\varphi}(a_{j\,:\,k})$. In that case $\overline{\varphi}(a_{i\,:\,k}) < \max\{\ \overline{\varphi}(a_{i\,:\,j-1}),\ \overline{\varphi}(a_{j\,:\,k})\}$ as the average score of the supsequence must be lower than the subsequence with maximal average score. The second possibility is that $\overline{\varphi}(a_{i\,:\,j-1}) = \overline{\varphi}(a_{j\,:\,k})$, thus $\overline{\varphi}(a_{i\,:\,k}) = \overline{\varphi}(a_{i\,:\,j-1}) = \overline{\varphi}(a_{j\,:\,k})$.

□

Theorem 2.4.5 *Any frequent motif $m \in \mathcal{M}_{p+s}$ can be partitioned into two submotifs $m^p \in \mathcal{M}_p$ and $m^s \in \mathcal{M}_s$, such that $m = m^p \mid\mid m^s$ and either the prefix m^p or suffix m^s is frequent.*

Proof. If we have only a single match of the motif m in sequence a, then according to Lemma 2.4.4

$$\operatorname{supp}(a, m, i) \leqslant \max\{\operatorname{supp}(a, m^p, i),\ \operatorname{supp}(a, m^s, i + p)\}\ .$$

Now, suppose we have more matches of m in one single sequence a. As maximal support only considers one occurrence of m, then according to Lemma 2.4.4 the theorem holds. By Definition 2.3.2, the additive support is the sum of supports of all single matches. Let $\operatorname{asupp}(a, m) = \overline{\varphi}(a_{i_1\,:\,j_1}) + \ldots + \overline{\varphi}(a_{i_n\,:\,j_n})$, where $n = \mathsf{mcount}(a, m)$ and i_n, j_n denote start and end locations of the occurrence. Let $\overline{\overline{\varphi}}(m_i) \overset{\text{def}}{=} (\varphi(a_{i_1+i-1}) + \ldots + \varphi(a_{i_n+i-1}))\,(p + s)^{-1}$ and $\overline{\overline{\varphi}}(m_{i\,:\,j}) \overset{\text{def}}{=} (\overline{\overline{\varphi}}(m_i) + \overline{\overline{\varphi}}(m_{i+1}) + \ldots + \overline{\overline{\varphi}}(m_j))(p + s)^{-1}$. Similarly to Lemma 2.4.4, we could show that

$$\overline{\overline{\varphi}}(m_{1\,:\,p+s}) \leqslant \max\{\ \overline{\overline{\varphi}}(a_{i\,:\,p-1}),\ \overline{\overline{\varphi}}(a_{p\,:\,p+s}))\ \} \tag{2.5}$$

which means that $\operatorname{asupp}(a, m) \leqslant \max\{\operatorname{asupp}(a, m^p),\ \operatorname{asupp}(a, m^s)\}$. Note that any extra occurrences of prefix or suffix motifs in input sequences do not invalidate Equation (2.5).

For either additive or maximal support over a set of sequences $\mathcal{S}$, everything works similarly to above steps, but we have to take into account the constant $|\mathcal{S}|^{-1}$.

□

It is easy to see that we can partition a frequent motif into any number of pieces such that at least one of them is frequent. We know that partitioning works for two submotifs, thus we can iteratevly continue and create as many partitions of the original motif as necessary, because always at least one partition has to be frequent.

Corollary 2.4.6 *Given motif $m \in \mathcal{M}_\ell$ and submotifs $m_1, m_2, \ldots, m_n$, that partition m into n pieces, then at least one of the submotifs must be frequent.*

So far we have described the properties of additive and maximal support without concentrating too much on the actual contents of the motifs. However, we used motif lengths in Proposition 2.4.3 to estimate maximal possible support of a supmotif. While this is useful knowledge, most of the time these estimations do not work best, because they make their estimations solely on the motif support, length and possible supmotif length. We can improve this situation by introducing wild card characters.

Proposition 2.4.7 *Given motifs $m \in \mathcal{M}_\ell$ and $m' \in \mathcal{M}_\ell$, that is constructed from motif m such way, that one nucleotide in m is replaced by a wild card character $*$, the additive or maximal support*

$$\mathrm{supp}(\mathcal{S}, m) \leqslant \mathrm{supp}(\mathcal{S}, m') \ .$$

For example, consider a motif $m^p = \texttt{GCT}$ as a prefix of a longer supmotif $m \in \mathcal{M}_{10}$. If we want to know the maximal support of any such supmotif, we can calculate $\mathrm{supp}(\mathcal{S}, \texttt{GCT*******})$. It certainly does not give higher estimation than former described method. On the other hand, it requires a query on the database, which depending on situation can be costly. We describe both approaches more thoroughly in Chapter 3.

2.5 Statistically Relevant Motifs

In previous sections, we discussed how to determine if a motif is frequent. In this section, we describe how to go even further by deciding, which frequent motifs are statistically more significant. By this, we actually want to measure the amount of surprise for every frequent motif. In our case, we may measure surprise individually even for every data track and we have several options for doing that. The simplest idea is to permute the letters of the promoter sequences. Then we can compare the support measures of the permuted dataset against the original one. If motifs in original data have higher supports, then they are surprising in that sense. To be more specific, we may generate a large amount of datasets by permuting randomly the original sequences. If we consider only one data track, then we can sort the motifs decreasingly by their support such that motif with highest support is the first in the resulting list. Then, for every motif at position i in the list, we can calculate how many motifs at i'th position in generated datasets had support as high as the original motif. We can write it down as

$$p = \Pr[\mathrm{supp}(\mathcal{S}', m') \geqslant \mathrm{supp}(\mathcal{S}, m)]$$

where $\mathcal{S}$ is the original dataset, $\mathcal{S}'$ is the permuted dataset, m is the original motif at i'th position and m' is the motif in $\mathcal{S}'$ at same position. The value p is called *p-value* in statistics and in our case, represents the probability of having the support in a random dataset at least as extreme as in the original one. Therefore, the smaller the p, the more surprising is the motif m.

We can calculate p-value for every frequent motif and for every data track. Of course, we might want to calculate only a single p-value for every motif, but the problem is with sorting frequent motifs. This actually can be done, as discussed in Chapter 3, but having a p-value in respect to each data track may reveal interesting properties of the motifs. We will omit exact algorithm for calculating p-values, but briefly discuss it later in Section 3.5. Let us refer to this algorithm as SigMotifs onwards.

Chapter 3

Algorithms and Data Structures

In this chapter, we will devise algorithms based on formalization and other ideas described in Chapter 2. We start off by describing compact encoding of motifs and continue developing algorithms with different pruning methods and capabilities.

3.1 Compact Encoding of Motifs

It turns out, that there is a rather straightforward way to encode fixed-length motifs as unique integers. If we consider nuclotides and wild card character as a set $\mathcal{X} = \{\mathtt{A}, \mathtt{T}, \mathtt{G}, \mathtt{C}, \mathtt{*}\}$ and have another set with same size $\mathcal{Y} = \{0, 1, 2, 3, 4\}$, then we can define a mapping $\pi : \mathcal{X} \longrightarrow \mathcal{Y}$, such that $\pi(\mathtt{A}) = 0, \pi(\mathtt{T}) = 1, \pi(\mathtt{G}) = 2, \pi(\mathtt{C}) = 3, \pi(\mathtt{*}) = 4$, that would enable us to represent a motif $m \in \mathcal{M}_\ell$ as an integer

$$5^0\pi(m_1) + 5^1\pi(m_2) + \ldots + 5^{\ell-1}\pi(m_\ell) \ . \tag{3.1}$$

For our convenience, let us agree that by writing $\pi(m)$, where $m \in \mathcal{M}_\ell$, we mean the integral representation of motif given in Equation (3.1).

This representation makes it easy to hash any motif of length ℓ and store it in a *hash-table* as for every fixed length motif the integral representation is unique.

If the motif length ℓ is small enough, then we could use a *hash-map* of size 5^ℓ. This way we could directly use the value $\pi(m)$ as a key to store motif's support metrics and this guarantees constant time $O(1)$ access as there would be no collisions.

In normal circumstances, we do not need to store support for all possible motifs. For example, there are $5^8 = 390625$ possible motifs of length 8 including wild card characters. For a yeast *S. Cerevisiae*, the promoter lengths are not usually longer than a few thousand base pairs. Therefore, if we have one promoter with length of 3000 base pairs, we can actually have maximal of $3000 - 8 = 2992$ different non wild card character motifs of length eight.

3.2 Hash-map of Support Metrics

The integral representation of motifs allows us to effectively build a *hash-map* containing support metrics of all motifs found in promoter sequences. Consider a sequence $a = \texttt{ATCCGTCCG}$. If we are interested in motifs of length 4, then motif $m^1 = \texttt{ATCC}$ matches the first position of a and motif $m^2 = \texttt{TCCG}$ matches the second position of a. The integral representations are following:

$$\pi(m^1) = 1 \cdot 0 + 5 \cdot 1 + 25 \cdot 3 + 125 \cdot 3 = 455$$

$$\pi(m^2) = 1 \cdot 1 + 5 \cdot 3 + 25 \cdot 3 + 125 \cdot 2 = 341 \ .$$

It turns out, that we can update the integral representation of m^1 to m^2 in constant time. By Equation (3.1), the integral representation of motif m^1 is $\pi(m^1) = 5^0 \cdot \pi(a_1) + 5^1 \cdot \pi(a_2) + 5^2 \cdot \pi(a_3) + 5^3 \cdot \pi(a_4)$. By subtracting the first element $5^0 \cdot \pi(a_1)$, dividing the result by five and adding $5^3 \cdot \pi(a_5)$, we get

$$\frac{\pi(m^1) - 5^0 \cdot \pi(a_1)}{5} + 5^3 \cdot \pi(a_5) = 5^0 \cdot \pi(a_2) + 5^1 \cdot \pi(a_3) + 5^2 \cdot \pi(a_4) + 5^3 \cdot \pi(a_5)$$

which is equal to $\pi(m^2)$. So in our example, where $\pi(m^1) = 455$, we can calculate

$$\pi(m^2) = \frac{\pi(m^1) - 5^0 \cdot \pi(a_1)}{5} + 5^3 \cdot \pi(a_5) = \frac{455 - 0}{5} + 125 \cdot 2 = 341 \ .$$

Analogously, we can do this with support of single matches for all tracks. Why this is important, is that we can calculate all support metrics of all motifs present in data in one pass. The negative side effect of this approach with scores are possibly greater floating-point rounding errors. But we can reduce them effectively by recalculating them from data tracks after every 100 or 1000 steps. This of course is not the issue with the integral representation.

Let us give an in-depth example. Consider two sequences $a = \texttt{ATCCGTCCG}$, $b = \texttt{TTCCG}$ and two mappings φ_1, φ_2 representing two data tacks such that

$$
\begin{aligned}
\varphi_1(a_{1\,:\,9}) &= 1.0, 1.0, 1.0, 1.0, 1.0, 0.5, 0.5, 0.5, 0.5\\
\varphi_2(a_{1\,:\,9}) &= 0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0\\
\varphi_1(b_{1\,:\,5}) &= 1.0, 1.0, 1.0, 1.0, 1.0\\
\varphi_2(b_{1\,:\,5}) &= 0.5, 0.5, 0.5, 0.5, 0.5\ \ .
\end{aligned}
$$

We can traverse the promoters step-by-step, such that after every cycle the hash-map contains up-to-date support metrics based on seen occurrences of motifs. All unseen occurrences are regarded as having single supports equal to zero. In our example, details of traversing a and b are given in table below.

Step	m	$\overline{\varphi}_1(m)$	$\overline{\varphi}_2(m)$	Comment
1	`ATCC`	1.0	0.5	Add `ATCC` to hash-map.
2	`TCCG`	1.0	0.5	Do same with `TCCG`.
3	`CCGT`	0.875	0.625	Keep adding unseen motifs into
4	`CGTC`	0.75	0.75	hash-map with their support
5	`GTCC`	0.625	0.875	metrics.
6	`TCCG`	0.5	1.0	Update support metrics of `TCCG`.
7	`TTCC`	1.0	0.5	We are processing b now.
8	`TCCG`	1.0	0.5	Update support metrics of `TCCG`.

For example, consider motif `TCCG`. For additive support over all sequences we sum $1.0/2+0.5/2+1.0/2$ for φ_1 and $0.5/2+1.0/2+0.5/2$ for φ_2. We divide the scores by two, due to Definition 2.3.4. After every update, the additive supports are up-to-date based on data seen so far. For maximal support, we need to do more book-keeping, because when we find an occurrence with bigger maximal score in a sequence, we have to cancel the effect of the previous occurrence. For example, the maximal support after step two is $0.5/2$ for φ_2. At step 6, we discover that it should be $1.0/2$ instead, therefore we subtract $0.5/2$ from the variable containing the support and add $1.0/2$.

With this kind of hash-map construction we calculate all the metrics on the fly. Therefore, we avoid any post-processing, because calculating the support measures over all sequences would otherwise require intermediate lists containing scores of single supports. With motifs without wild card characters, this would not be very big memory overhead, but otherwise it could become an issue. Total runtime complexity of this method is

$$O\left(n \cdot c \cdot \sum_{s \in \mathcal{S}} |s|\right)$$

where n is the number of data tracks and c is the complexity for updating the support of a motif in the hash-map.

3.2.1 Including Motifs with Wild Card Characters

We will discuss SafeApproxSearch in Section 3.4.2, where hash-maps are required to also contain supports of all wild character extensions. This requires us to modify the method described earlier. The integral representation allows us to precompute suffix parts of all extensions. *Let w^i be suffix part of some motif m of length ℓ, such that $1 \leqslant i \leqslant \ell$ and $m_{i\,:\,\ell} = \texttt{*}\ldots\texttt{*}$.* Then $\pi(w^i) = 5^{i-1}\pi(\texttt{*}) + \ldots + 5^{\ell-1}\pi(\texttt{*})$. If we now have the integral representation of a prefix m^p, then $\pi(m^p) + \pi(w^i)$ will yield the integral representation of the wild card character extension. In hash-map construction phase, it requires ℓ steps instead of one to include the support metrics of all wild card character extensions, therefore the complexity is

$$O\left(n \cdot c \cdot \ell \cdot \sum_{s \in \mathcal{S}} |s|\right) \quad .$$

3.3 Naive Search based on Apriori

The simplest search method is based on the Apriori principle described in Chapter 1. Namely, we can mine all motifs present in input sequences by setting the threshold $\sigma = 1$ with Apriori and then check if they are frequent in our terms. This is actually a composition of Apriori and a filtering function. In our case, it is better to implement this as a depth-first search algorithm, because breadth-first nature of Apriori causes too much memory overhead, when mining longer motifs. The Algorithm 3.3.1 incorporates the composition of Apriori and the filtering function. On lines 10 – 12, we see the candidate generation part of the algorithm. Note that we always use motifs `A, T, G, C` for extension. This is due to the fact that there are rarely cases, where a nucleotide in promoter sequences is missing. The Apriori

Algorithm 3.3.1 NaiveSearch

```
1:  procedure NaiveSearch($\mathcal{S}, \vec{\sigma}, m, \ell$)
2:      if $\mathsf{mcount}(\mathcal{S}, \mathsf{m}) = 0$ then
3:          return
4:      else if $|m| = \ell$ then
5:          if IsFrequent ($\vec{\sigma}$, $\overrightarrow{\text{supp}}(\mathcal{S}, m)$) then
6:              SaveMotif($m$)
7:          end if
8:          return
9:      end if
10:     for $e \in \{\texttt{A, T, G, C}\}$ do
11:         NaiveSearch($\mathcal{S}, \vec{\sigma}, m \mid\mid e, \ell$)
12:     end for
13: end procedure
```

pruning principle is in action on lines 2 – 3 and the filtering function is given on lines 4 – 9. Function IsFrequent checks, if all thresholds $\sigma_i \geqslant s_i$ where $\vec{s} = \overrightarrow{\text{supp}}(\mathcal{S}, m)$. Recall, that $\overrightarrow{\text{supp}}$ operator returns a vector of values, where each element determines the support per one data track according to additive or maximal support type. Also, if implementations of $\overrightarrow{\text{supp}}$ and mcount are implemented using data structures like hash-map discussed in previous section, then these need to be constructed before running this algorithm.

As an example, calling NaiveSearch($\mathcal{S}, \vec{\sigma}, \theta, 8$), where θ is the empty zero-length motif, $\mathcal{S}$ is the set of sequences and $\vec{\sigma}$ is the vector of thresholds, we find all frequent motifs of length 8. The complexity of NaiveSearch is $O(4^\ell)$, where ℓ is the fixed motif length.

3.4 Pruning Strategies

In this section, we describe different pruning strategies, which can be used to make more efficient algorithms compared to NaiveSearch. All these methods are based on properties studied in Chapter 2.

3.4.1 Maximal Support Estimation Pruning

The simplest method is based on Proposition 2.4.3. Namely, if we are mining motifs with length $\ell + k$ and we have some motif $m \in \mathcal{M}_\ell$, then the support measures of any of its super motifs with length $\ell + k$ cannot be greater than motif having m as a prefix and hypothetical suffix with score 1.0. Therefore, a motif m and its supmotifs can be pruned, if on any of the data tracks

$$\frac{\ell \cdot \mathrm{msupp}(\mathcal{S}, m) + |\mathcal{S}| \cdot k}{\ell + k} < \sigma$$

if we are mining using maximal support or

$$\frac{\ell \cdot \mathrm{asupp}(\mathcal{S}, m) + \mathsf{mcount}(\mathcal{S}, m) \cdot k}{\ell + k} < \sigma$$

if we are mining using additive support. Of course, the maximal motif length $\ell + k$ must be fixed to make these formulas usable. As an example, let us analyze Figure 3.4.1.

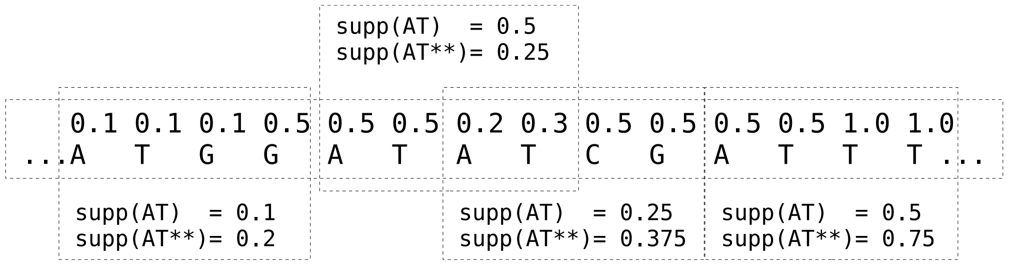


Figure 3.1: Support of motifs `AT` and `AT**` in a sample subsequence.

We see that $\mathrm{msupp}(\mathcal{S}, \mathtt{AT}) = \max\{0.1; 0.5; 0.25; 0.5\} = 0.5$ and $\mathrm{asupp}(\mathcal{S}, \mathtt{AT}) = 0.1+0.5+0.25+0.5 = 2.25$. If we were mining using maximal type of support on this track, then we can prune the motif with its supmotifs if

$$(2 \cdot \mathrm{msupp}(\mathcal{S}, \mathtt{AT}) + 2)/4 = (2 \cdot 0.5 + 2)/4 = 0.75 < \sigma$$

where σ is the threshold. For additive type of support this would be

$$(2 \cdot \mathrm{asupp}(\mathcal{S}, \mathtt{AT}) + 2 \cdot \mathsf{mcount}(\mathcal{S}, \mathtt{AT}))/4 = \; = (2 \cdot 0.5 + 2 \cdot 4)/4 = 2.25 < \sigma \;.$$

Incorporating this pruning method requires only small changes to NAIVE-SEARCH on line 2 of Algorithm 3.3.1. The result is given in Algorithm 3.4.1, where CANPRUNE uses method described above to determine if the motif and supmotifs can be pruned.

Algorithm 3.4.1 Search Using Maximal Support Estimation for Pruning

```
1:  procedure MaxSupSearch($\mathcal{S}, \vec{\sigma}, m, \ell$)
2:      if mcount($\mathcal{S}$, m) $= 0 \vee$ CanPrune($\vec{\sigma}$, $\overrightarrow{\text{supp}}(\mathcal{S}, m)$) then
3:          return
4:      else if $|m| = \ell$ then
5:          if IsFrequent($\vec{\sigma}$, $\overrightarrow{\text{supp}}(\mathcal{S}, m)$) then
6:              SaveMotif($m$)
7:          end if
8:          return
9:      end if
10:     for $e \in \{$A, T, G, C$\}$ do
11:         MaxSupSearch($\mathcal{S}, \vec{\sigma}, m \,||\, e, \ell$)
12:     end for
13: end procedure
```

3.4.2 Safe Over-Approximation Search

Another improvement to NaiveSearch uses slightly different approach. It is based on Proposition 2.4.7 that stated that support of any motif m' gained from motif m by replacing one or more nucleotides with wild card characters, is greater or equal compared to original motif. Also, it holds with either maximal or additive type of support. This allows us to define a support operator that is guaranteed to be downward closed, which was an issue with NaiveSearch and MaxSupSearch [Lau09]. We will be referring to it as *safe over-approximation* type of support onwards.

Definition 3.4.1 *Let* $\text{supp}^*(\mathcal{S}, m)$ *of motif* $m \in \mathcal{M}_\ell$ *denote the support of its wild character extension* $m^* \in \mathcal{M}_k$*, where* $\ell \leqslant k$*.*

Recall that a wild card character extension of m was a fixed length motif that contained m as a prefix and rest of the elements (wild card characters) as the suffix. As an example, if we are interested in mining sequences of length $\ell = 3$, we first start by checking the support of wild card character extensions of motifs in $\mathcal{M}_1$, namely `A**, T**, G**, C**` (note that we do not include motif `*` in this list, as it is anyway the most frequent motif and we are not interested in it). If any of these motifs is infrequent, for example `T`, then we prune all its supmotifs `TAA, TAT, TAG, TAC, TTA` *et cetera*. But if `T` is frequent, we continue to check its submotifs `TA, TT, TG, TC` using supp^* operator. We only have to keep in mind, that it is downward closed only when mining motifs with fixed length, so that Proposition 2.4.7 would hold.

Algorithm 3.4.2 Safe Over-Approximations Search

```
1:  procedure SafeApproxSearch($\mathcal{S}, \vec{\sigma}, m, \ell$)
2:      if IsFrequent($\vec{\sigma}$, $\overrightarrow{\text{supp}}^*(\mathcal{S}, m)$) then
3:          if $|m| = \ell$ then
4:              SaveMotif($m$)
5:              return
6:          end if
7:      else
8:          return
9:      end if
10:     for $e \in \{$A, T, G, C$\}$ do
11:         SafeApproxSearch($\mathcal{S}, \vec{\sigma}, m \,||\, e,\ \ell$)
12:     end for
13: end procedure
```

The Algorithm 3.4.2 defines SafeApproxSearch. Note that we use $\overrightarrow{\text{supp}}^*$ operator instead of $\overrightarrow{\text{supp}}$ and use IsFrequent to determine, whether we can prune the motif with its supmotifs. This is possible due to downward-closeness of $\overrightarrow{\text{supp}}^*$ operator.

Both MaxSupSearch and SafeApproxSearch have similar theoretical runtime complexity $O(f \cdot 4^{\ell})$, where pruning factor $f \in (0, 1]$ is maximal, if no pruning occur and minimal, if all motifs are pruned.

3.4.3 Infrequent Sub-Motifs Pruning Method

This alternative search method is directly based on Theorem 2.4.5. Namely, if we are interested in motifs with length ℓ, then for any partitioning of a frequent motif $m \in \mathcal{M}_{\ell}$ into two pieces m^1, m^2, at least one of the pieces must be frequent. The idea is to generate two sets $\mathcal{F}$ and $\mathcal{I}$, where $\mathcal{F}$ contains the frequent motifs and $\mathcal{I}$ the infrequent ones of length $\ell/2$. Thus, we combine motifs from $\mathcal{F}$ and $\mathcal{I}$ to enumerate final candidates. Note, that we need $\mathcal{I}$, because any frequent motif of length ℓ may have infrequent prefix or suffix. We do not need to consider combinations of infrequent submotifs as due to Theorem 2.4.5 we know, that the resulting motif is also infrequent. Also, there are many ways to partition the motifs, but making them with same length enables us to enumerate them faster. The Algorithm 3.4.3 describes this process.

Algorithm 3.4.3 Infrequent Sub-Motifs Search

```
1: procedure InfreqSearch($\mathcal{S}, \vec{\sigma}, m, \ell$)                ▷ $\ell$ must be even
2:     $(\mathcal{F}, \mathcal{I}) \leftarrow$ EnumerateMotifs($\mathcal{S}, \vec{\sigma}, \ell/2$)
3:     $\mathcal{C} \leftarrow \{(a,b) \mid a \in \mathcal{F}, b \in \mathcal{F} \cup \mathcal{I}\}$
4:     for $c \in C$ do
5:         if CanPrune($\vec{\sigma}$, $\overrightarrow{\text{supp}}(\mathcal{S}, c)$) then
6:             continue
7:         else if IsFrequent($\vec{\sigma}, \overrightarrow{\text{supp}}^*(\mathcal{S}, c_1 \mid\mid c_2)$) then
8:             SaveMotif($c_1 \mid\mid c_2$)
9:         else if $c_1 \neq c_2$ then
10:            if IsFrequent($\vec{\sigma}, \overrightarrow{\text{supp}}^*(\mathcal{S}, c_2 \mid\mid c_1)$) then
11:                SaveMotif($c_2 \mid\mid c_1$)
12:            end if
13:        end if
14:    end for
15: end procedure
```

On line 3, we enumerate all the candidate motifs of length ℓ. On line 5, we first try to eliminate candidates by using information we know about their prefix m_1 and suffix m_2. We try this, because querying the database, depending on data structures used, can be more costly. The CanPrune method checks on every track if

$$\text{msupp}(\mathcal{S}, m_1 \mid\mid m_2) \leqslant \frac{\text{msupp}(\mathcal{S}, m_1) + \text{msupp}(\mathcal{S}, m_2)}{2} < \sigma$$

for maximal support type and

$$\text{asupp}(\mathcal{S}, m_1 \mid\mid m_2) \leqslant \frac{\text{asupp}(\mathcal{S}, m_1) + \text{asupp}(\mathcal{S}, m_2)}{2} < \sigma$$

for additive support type. These formulas are derived from equations in Proposition 2.4.3. If we can prune $m_1 \mid\mid m_2$ using above equations, then we can also prune $m_2 \mid\mid m_1$ as there is no difference, in what order we consider the prefix and suffix part.

3.5 Mining Fixed Number of Best Motifs

The search algorithms discussed in earlier sections concentrate on finding all frequent motifs in respect to some threshold vector. But suppose we want to mine 100 "best" motifs. Doing this by hand using any previously mentioned search algorithm would require following process. First, we determine some reasonable thresholds and support types for data tracks. Second, we mine frequent motifs using these thresholds and decide, whether the number of motifs was too small or too large. Third, we modify the thresholds by increasing or decreasing them and mine again until we have desired number of frequent motifs.

The process we just described is actually similar to *binary search* known in computer science. The Algorithm 3.5.1 implements it to automate this process. On line 3, we determine two scalars α and β, such that mining with $\alpha \cdot \vec{\sigma}$ returns all motifs present in data and mining with $(\beta + \varepsilon) \cdot \vec{\sigma}$ returns none of the motifs where $\varepsilon > 0$. It is trivial, that $\alpha = 0$, because in that case all motifs will be frequent. Determining β is more complicated, because we do not have any prior knowledge about maximal supports in data. First option is to make a guess, but a better alternative is to find out the supports by calculating $\vec{s} = \overrightarrow{\mathrm{supp}}^*(\mathcal{S}, *)$ and set

$$\beta = \max\{s_i/\sigma_i \mid i = 1, \ldots, n\} \tag{3.2}$$

where n is the number of data tracks and $\vec{\sigma}$ contains user-defined thresholds. This way $\beta \cdot \vec{\sigma}$ may return only minimal possible number of frequent motifs. Having these boundaries fixed, we can easily combine any previously defined search method with binary search. In other words, we keep scaling the original vector of thresholds $\vec{\sigma}$, until we get desired number of frequent motifs. The linearity of this approach may not always be the best choice, because the relations between the reasonable thresholds depend on the nature of the data. We do not study further possibilities in this work, but it could be a possible research area in the future.

Algorithm 3.5.1 Algorithm for Mining Fixed Number of Best Motifs

```
 1: procedure NBest($\mathcal{S}, \vec{\sigma}, N, \ell$)
 2:     $\vec{s} \leftarrow \overrightarrow{\mathrm{supp}}^*(\mathcal{S}, *)$
 3:     $\alpha \leftarrow 0, \beta \leftarrow \max\{s_i/\sigma_i \mid i = 1, \ldots, n\}$      ▷ $n$ is the number of tracks
 4:     $C \leftarrow \infty$                                ▷ The closest number of best motifs
 5:     $\delta \leftarrow 0$       ▷ Scalar to be used to mine closest number of best motifs
 6:     while $\beta - \alpha > \varepsilon$ do                    ▷ $\varepsilon > 0$ limits the recursion depth
 7:         $\gamma \leftarrow (\alpha + \beta)/2$
 8:         $k \leftarrow$ NumFreqMotifs($\mathcal{S}, \gamma \cdot \vec{\sigma}, \theta, \ell$) ▷ $\theta$ is the zero-length motif
 9:         if $\mathrm{abs}(k - N) < C$ then
10:             $C \leftarrow k, \delta \leftarrow \gamma$
11:         end if
12:         if $k > N$ then
13:             $\alpha \leftarrow \gamma$
14:         else if $k < N$ then
15:             $\beta \leftarrow \gamma$
16:         else if $k = N$ then
17:             break
18:         end if
19:     end while
20:     return MineMotifs($\mathcal{S}, \delta \cdot \vec{\sigma}, \theta, \ell$)
21: end procedure
```

Function NumFreqMotifs can be used as a wrapper around search methods described in earlier sections. There are still a few things to consider. First, not always there exist some fixed number of best motifs, because two motifs may have exactly same support measures. In that case, binary search goes into infinite loop. Same happens, when the number of desired motifs is greater than there are motifs present in input data. In both situations, we need to limit the maximal depth of the recursion. But we can still return the number of motifs, that is very close to desired number of motifs. On line 3, we define C that will remember, what was the closest number of frequent motifs to the desired fixed number of motifs. Scalar δ can be used to scale $\vec{\sigma}$ to get C frequent motifs. On line 6, we use $\varepsilon > 0$ to limit the recursion depth. On lines 12 – 18, we see binary search in action. The while loop terminates when the recursion depth limit is reached or scalar, that returns desired number of frequent motifs, is found. After that, the MineMotifs function used as a wrapper around any previously defined search method finally returns the motifs.

The complexity of this approach is $O(d \cdot 4^{\ell})$, where d is the maximal recursion depth of binary search and 4^{ℓ} is the worst-case complexity of NaiveSearch, MaxSupSearch and SafeApproxSearch where ℓ is the fixed motif length.

Another, rather naive, but reasonable alternative is to mine at least the desired number of motifs from input data set and sort them. A reasonable criteria for sorting can be derived from Equation (3.2), that we used to calculate the value β. Suppose we have mined f frequent motifs $m^1, m^2, \ldots, m^f$. Given an vector of thresholds $\vec{\sigma}$, we can calculate scalars

$$\gamma^j = \max\{s_i/\sigma_i \mid i = 1, \ldots, n\}$$

where $j \in \{1, \ldots, f\}$, n is the number of data tracks and $\vec{s} = \overrightarrow{\text{supp}}^*(\mathcal{S}, m^j)$. These scalars have an interesting property. For any motif m^j present in input data, IsFrequent$(\vec{\sigma}, \gamma^j \cdot \overrightarrow{\text{supp}}^*(m^j)) = \mathsf{true}$, where γ^j is minimal such scalar for m^j. If we sort motifs $m^1, \ldots, m^f$ decreasingly using γ^j as the key for motif m^j, then we get a list where first N motifs are the "best" mined motifs. The complexity of sorting is $O(f \cdot log_2 f)$. Also, if we do not want to guess thresholds and mine frequent motifs before sorting, then we can get a list of present motifs in the data along with hash-map construction in linear time to the total length of input sequences, because we need the support metrics of the motifs anyway. Therefore, if the number of present motifs is small, then sorting definitely has the advantage. On the other hand, we are not usually interested in more than 100 frequent motifs. Therefore, in a large set of promoter sequences, NBest could work faster.

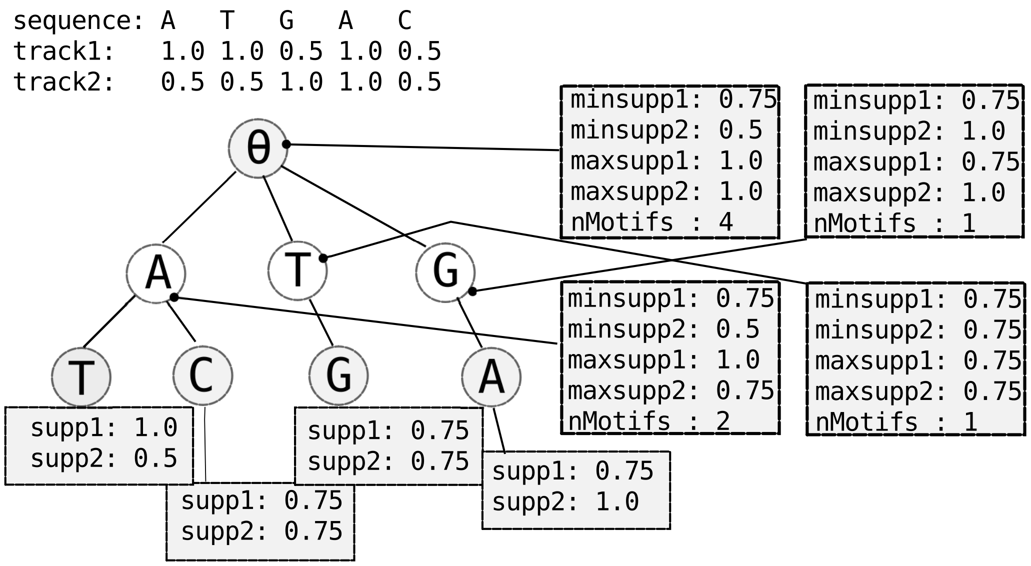
Another thing to be considered is the NumFrequentMotifs function used in Algorithm 3.5.1. It only needs to know the *number* of frequent motifs not the actual motifs themselves. This allows us to prune the search even better than MaxSupSearch and SafeApproxSearch do. We will describe this in the next section.

Also, both NBest and sorting approaches can be used as a central part in mining statistically significant motifs with SigMotifs described in Section 2.5, because SigMotifs requires lists of "best" mined motifs from permuted datasets to determine the p-values of original motifs. We won't discuss

SIGMOTIFS any further here, but we use it to mine significant motifs in a experiment discussed in Chapter 4.

3.6 Generalized FP-Tree

In this section, we describe a data structure that is optimized to tell us how many motifs in input data are frequent, given some vector of thresholds. We will use a generalization of FP-Tree [JJYR04] that is widely used in standard data mining applications. The general idea is simple: the tree contains support of all fixed-length motifs in promoter data and maintains relationships between sub and supmotifs, such that given a motif we can tell how many supmotifs there are and what are the minimum and maximum values of scores per each data track. This way it is easy to determine the number of frequent motifs in the tree given the vector of thresholds $\vec{\sigma}$.



sequence: A T G A C
track1: 1.0 1.0 0.5 1.0 0.5
track2: 0.5 0.5 1.0 1.0 0.5
θ
A
T
G
T
C
G
A
minsupp1: 0.75 minsupp2: 0.5 maxsupp1: 1.0 maxsupp2: 1.0 nMotifs : 4
minsupp1: 0.75 minsupp2: 1.0 maxsupp1: 0.75 maxsupp2: 1.0 nMotifs : 1
minsupp1: 0.75 minsupp2: 0.5 maxsupp1: 1.0 maxsupp2: 0.75 nMotifs : 2
minsupp1: 0.75 minsupp2: 0.75 maxsupp1: 0.75 maxsupp2: 0.75 nMotifs : 1
supp1: 1.0 supp2: 0.5
supp1: 0.75 supp2: 0.75
supp1: 0.75 supp2: 0.75
supp1: 0.75 supp2: 1.0

Figure 3.2: Generalized FP-Tree of Sequence `ATGAC` of motifs with length of two.

Consider sequence `ATGAC` given in Figure 3.2. If we were to mine motifs of length two, then the we would build the tree shown in the figure. Note that every leaf of the tree represents one motif present in data and contains the support metrics of that motif. Every intermediate node contains information about how many leaves it has and what are the minimal and maximal support measures of them.

For example, if thresholds were $\vec{\sigma} = (0.5, 0.5)$, then already at root node we know, that all the motifs in the tree are frequent. Thus, we do not need to look any further, but just return the number of motifs. If $\vec{\sigma} = (0.5, 1.0)$, then we have to recurse from the root node to make any decisions. At intermediate node `A`, we see that all submotifs are frequent in respect to threshold of first track, but they are all infrequent against the threshold of the second track. Thus, the intermediate node has no frequent submotifs. At intermediate node `T`, exactly same applies. At intermediate node `G`, we see that all submotifs are frequent regarding the thresholds, thus at root node we compute that the number of frequent submotifs is $0 + 0 + 1 = 1$. This example demonstrated the pruning capabilities with GFP-Tree from above and below, therefore making this structure optimized for returning the *number* of frequent motifs regarding some thresholds. Also, using this tree structure for support metrics retrieval in SafeApproxSearch instead of a hash-map with additive support type, pruning infrequent motifs is possible much earlier in the search process. This is due to fact that GFP-Tree is capable of returning the actual maximum support among supmotifs, whereas hash-map sums the supports of the supmotifs.

We will refer to this search method as GFPSearch onwards, but we omit exact algorithm for the sake of space. Still, let us once more clarify the pruning step part of the algorithm. Suppose we have two data tracks and we are in an intermediate node, trying to decide, what is the number of frequent submotifs in this subtree. We can compose a table containing subnodes as row headers and data tracks as column headers. For every data track, we can write if relevant threshold is equal or below of the minimal support of the subnode, above the maximal support or between the minimal (not included) and maximal (included).

Subnode	$track_1$	$track_2$	Comment
`A`	above	above	No motifs are frequent in this subtree
`T`	middle	above	No motifs are frequent in this subtree
`G`	below	middle	We have to look further to decide
`C`	below	below	All motifs are frequent in this subtree

If a row contains value above, then there are no frequent motifs in that subtree. If all values are below, then all submotifs in the subtree are frequent. In case the values are a mix of below and middle, we have to recurse to the subtree to decide the number of frequent submotifs. After that, we sum up the total number of frequent submotifs at this intermediate node and return the result to parent that deals with it onwards.

The complexity of telling how many motifs are frequent, given a threshold vector, is with similar complexity to SafeApproxSearch, but in addition to that, we can prune the search from below as we only want to know the number of motifs. Composing this functionality with binary search can effectively find thresholds that yield desired number of frequent motifs or at least the number of results that are closest to them. Constructing such a tree takes $O(n \cdot \ell)$ time, where n is the total length of sequences and ℓ is the fixed motif length.

Of course, GFP-Tree can be also used to actually fetch the frequent motifs, but this eliminates the pruning possibility from below, as we actually have to recurse to the leaves to reach the motifs. In that case, the theoretical runtime complexity is exactly the same as with SafeApproxSearch.

Chapter 4

Experimental Results

In this chapter, we describe several experiments we have performed to further study and compare different algorithmic capabilities of methods studied in previous chapters. In Section 4.1, we will discuss run-time performance of search algorithms and in Section 4.2, we discuss the biological significance of mined motifs. For these purposes, we have written a C++ application that implements all search algorithms described in this work, see Appendix A. The computer we used to run the tests had following specs: Intel Pentium M CPU 1.73 GHz with 2MB of L2 cache, 1GB of DDR2 RAM, Fedora 12 (kernel version 2.6.31.5) operating system.

4.1 Runtime Performance of Search Algorithms

In this section, we run two types of tests. First, we compare the run-time performance of algorithms NaiveSearch, MaxSupSearch, SafeApproxSearch, InfreqSearch and GFPSearch by mining motifs from data sets with given thresholds. Secondly, we compare NBest combined with GFPSearch against MergeSort and test, how fast they manage to retrieve fixed number of frequent motifs from input data.

4.1.1 Mining Frequent Motifs

For testing all search methods with given thresholds, we need to also consider one other aspect. Namely, NaiveSearch, MaxSupSearch, SafeApproxSearch, InfreqSearch all need hash-maps discussed in Section 3.2

for support retrieval. What is more, SafeApproxSearch needs hash-map, that contains also wild card character extensions. NaiveSearch and MaxSupSearch do not need wild card characters, but they need hash-maps for all motif lengths up to ℓ, if we are mining motifs of length ℓ. InfreqSearch needs two hash-maps without wild card characters: one, that contains support metrics for motifs of length ℓ and another, that contains metrics for motifs of length $\ell/2$. And finally, GFPSearch requires GFPTree for being able to perform at all. As we are interested in practical value of the algorithms, we also studied the time required to build necessary data structures.

For benchmarking, we decided to mine motifs of length 8 and use automatically generated datasets with total length of the promoters from 500 up to 25000, where the length of one promoter was exactly 500 nucleotides long. We generated ten datasets with given number of promoters for every search method and measured the average running time of the search algorithm. Also, total working time including data structure construction was measured and we calculated, how many motifs were processed. We generated four data tracks for each promoter where the scores of the data tracks were generated randomly. Each promoter sequence was generated using a Markov chain, but with different characteristics. We mined the datasets using three types of thresholds: *low, medium* and *high. Low* setting means, that thresholds are equal to 0.05 for all data tracks, not depending whether we mine using maximal or additive support. *Medium* settings means, that thresholds are equal to 0.3 and *high* setting means, that thresholds are equal to 0.55. Such settings were chosen without no particular reason, but in hope of finding interesting patterns in behavior of the algorithms. Also, two tracks were mined using additive and two using maximal support type.

Let us analyze the results given in Figure 4.1. When we compare algorithm running times, then InfreqSearch is the slowest method with low and medium thresholds. There are two possible reasons. First, the method has to enumerate all motifs of length four. Second, pruning strategy of InfreqSearch does not work well with very low thresholds. On the other hand, with high thresholds, it is as fast as SafeApproxSearch and GFPSearch. Also, running times of the MaxSupSearch seem to be very dependent on thresholds. With low thresholds, it performs similarly to NaiveSearch, with medium thresholds it works much faster and even better with high thresholds. The fastest algorithms are SafeApproxSearch and GFPSearch and they perform similarly well. Only differ-

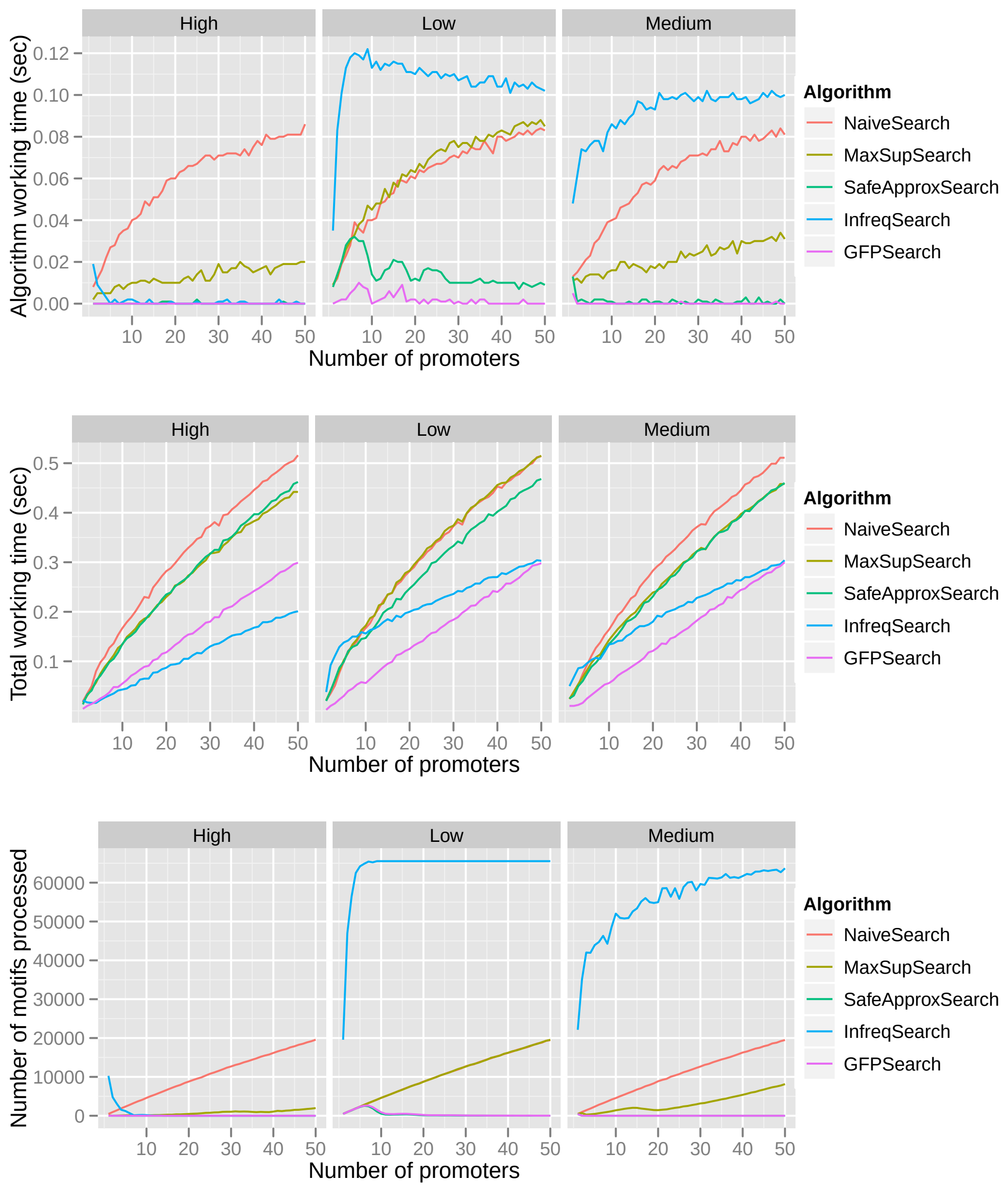


Figure 4.1: Comparison of algorithm running time (top), total running time (middle) and number of motifs not pruned(bottom) of the search algorithms, when mining motifs of length eight with either low, medium or high supports.

ence is with medium thresholds, where SafeApproxSearch works about 20 milliseconds slower on average. The observation that GFPSearch is faster, is due to nature of GFP-Tree and ability to help deciding faster, what branches to prune. Other than that, we actually see two types of behavior here. First, running times of NaiveSearch and MaxSupSearch grow constantly when the number of promoters is increased. This happens, because the number of motifs present in input data grow, but pruning strategy of MaxSupSearch is looser than strategies of SafeApproxSearch, GFPSearch and NaiveSearch. Second, running times of InfreqSearch, SafeApproxSearch and GFPSearch seem to decrease or be constant, when the number of promoters is increased. This implies that the efficiency of their pruning strategies grow with the number of infrequent motifs present in data.

If we now consider also the time needed to build necessary data structures, then first thing we notice, is that InfreqSearch performs rather well compared to other search algorithms using hash-maps for support retrieval. This is due to fact that InfreqSearch needs support only for motifs of length eight and four, where NaiveSearch and MaxSupSearch require that we have information about all motifs present in datasets up to length of eight. Recall that SafeApproxSearch required hash-maps to contain wild card character extensions. This requirement seems to put SafeApproxSearch almost on the same bar with NaiveSearch and MaxSupSearch. Surprisingly, GFPSearch outperforms all other search methods, except with high thresholds InfreqSearch is faster. On all three plots we see that the construction time of the data structures seem to be more or less linear to the total length of input sequences. One other observation is that if we modified NaiveSearch to be even more naive, such that it does not do Apriori pruning check, then it could work with a hash-map containing only fixed-length motifs. Therefore, the total running times could easily compete with GFPSearch, because the time needed to construct the hash-map would be roughly half the time necessary for the InfreqSearch.

If we now analyze the number of motifs that were not pruned, then InfreqSearch really seems to have the pruning strategy, that is very effective with high thresholds, but bad with low and medium thresholds. This also suggests that support of motifs goes really low in the generated data if the number of promoters goes higher than ten. Again, we see that MaxSupSearch is very dependent on the thresholds and pruning strategies of SafeApproxSearch and GFPSearch work very well with low, medium and high thresholds.

To sum these results up, then the results would have differed quite a bit,if we used data with other characteristics. But clear conclusions are, that SafeApproxSearch and GFPSearch are superior to others search methods. Let us also remind, that the reason, why GFPSearch was faster in our tests, was due to different data structures used to retrieve support. Hash-map used by SafeApproxSearch was designed to be more modular to enable working with different algorithms. But GFP-Tree and GFPSearch were designed to work only with each other.

4.1.2 Mining Fixed Number of Frequent Motifs

In this experiment, we were interested if NBest with its binary search approach can outperform MergeSort, when mining fixed number of motifs from datasets. For that purpose, we generated datasets exactly with same characteristics as we did in last section. Again, we took average running times from 10 runs, where the total length of input sequences were between 500 and 25000 nucleotides, where we were mining 100 "best" motifs. Based on the results of last experiment, we decided to use NBest in conjunction with GFP-Tree and GFPSearch. We enabled all pruning capabilities of GFPSearch here, as we only need to know the number of motifs instead of fetching the actual motifs, given some thresholds. We compared it against MergeSort, where we enumerated all motifs by first building a hash-map containing the support metrics and then fetching the motifs present in data into a sortable vector. The results are given in Figure 4.2.

We see that growth of both methods is roughly linear to the number of input sequences, where NBest seems to perform slightly faster where number of promoters is less than 18 and slightly slower afterwards. The reason here is that the running times of constructing a GFP-Tree increase faster than building a hash-map for fixed length motifs. Actually, based on observations in the previous section, we can say that the running times of GFPSearch and MergeSort make up only a fraction of the total due to time required by calculating the support metrics. Also, the results given here can differ on multiprocessor systems due to *divide and conquer* nature of MergeSort and possibility to construct the hash-map with several threads traversing different promoters simultaneously. In a similar fashion, it is possible with GFPSearch and GFP-Tree construction. In that case, both approaches could work a few times faster. Still, complexity of GFP-Tree construction

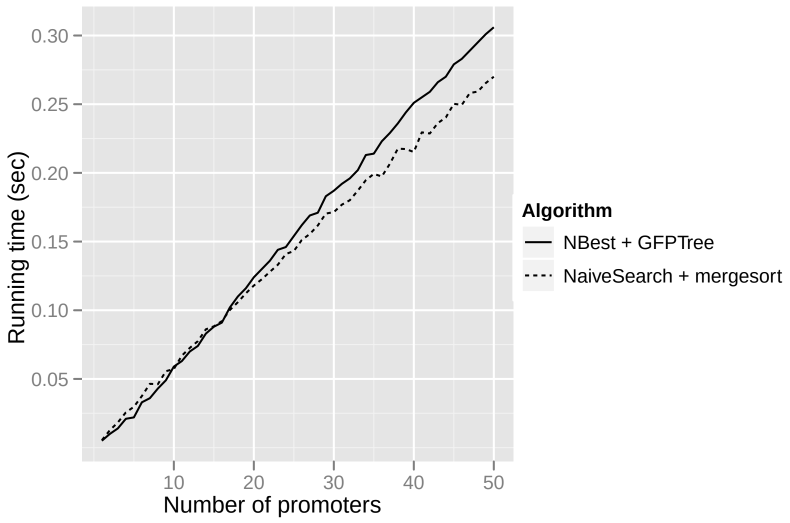


Figure 4.2: Comparison of runtime speed of NBest and MergeSort, finding 100 best motifs in datasets.

is slightly higher, therefore it is probable that sorting method could outperform binary search approach significantly on multiprocessor machines. On the other hand, MergeSort can work only if comparing supports of two motifs is possible. Recall, that in Section 3.5 we concluded that modifying thresholds by scaling them linearly may not be the best possible method to find "best" motifs. An *ad-hoc* approach might suggest completely different schemes for doing that. In that case, using MergeSort may be impossible, but NBest stays a viable alternative, because it can be easily modified to handle more complex threshold changing schemes.

4.2 Mining Biologically Significant Motifs

4.2.1 Data Preparation

For this experiment, we decided to use data of yeast *S.Cerevisiae*, that we collected from several online databases and compiled them into individual tracks and sequences usable by our computer program. Exact pieces that we collected were promoter sequences with their coordinates and direction on the DNA strand [MOJ+08], *phastCons* conservation data of all chromosomes [KSF+02], binding enrichment scores for transcription factors [NB08] and *invivo* nucleosome occupancy data [NIY+09].

The next step was to put all promoter sequences on one strand, therefore we had to reverse the sequences and get the *complementary reverses* where necessary. Next, we normalized all conservation and nucleosome data, such that all values fell in range between zero and one. From *invivo* nucleosome occupancy scores we calculated nucleosome freeness scores, so higher scores would mean higher chance for a transcription factor binding site. Last step was to cut all data from relevant positions in the datasets and connect them with the promoter sequences. For binding scores, though, we need to calculate the score tracks for every promoter sequence separately from enrichment scores, before we start mining. We do it by taking the average of all enrichment motifs that match the particular position in the sequence. This process is largely automated by helper scripts, that come with the computer program, see Appendix A.

4.2.2 Results

We decided to use gene MCM1, that has an important role in cell life cycle regulation of *S.Cerevisiae*. We used following promoter sequences that are documented or potential transcription factors of MCM1: FHL1, OAF1, ABF1, ADR1, ASH1, AZF1, CUP2, FKH2, GAL4, GCN4, GCR1, GIS1, GSM1, HAC1, HSF1, MSN2, MSN4, NRG1, RTG1, RPH1, RTG3, SKN7, STB5, STP1, STP2, SWI4, YER130C [MPP+06]. We associated three data tacks with each one of them: conservation, binding and invivo nucleosome freeness scores. The support types were additive, additive and maximal respectively and when mining 100 best motifs, we gave each track equal weight by setting threshold vector $\vec{\sigma} = \{1.0, 1.0, 1.0\}$. SigMotifs generated 10 000 background datasets by permuting randomly the nucleotide sequences of original data track. Then, we associated a p-value with each mined motif and data track. We also calculated test statistic for every motif with *Fisher's method* known from statistics:

$$(-2) \cdot \sum_{i=1}^{i=n} \ln(p_i)$$

where n is the number of data tracks and p_i is the p-value of the motif on i'th track. Let us refer to this statistic as *significance* onwards.

In Figure 4.3, we see relations between conservation, binding and freeness scores of 100 mined motifs. We see that conservation and binding scores seem to be very correlated. Most motifs have both support measures less than 1.5, only two motifs have support double that much. The middle and bottom scatter plots are rather similar. It is due to high correlation between conservation and binding scores, but there are no motifs with freeness score greater than 0.5. In all three plots, significance of motifs seem to be also correlated to the support measures.

In Figure 4.4, we see relation between p-values of individual tracks. When we compare conservation and binding, then surprising motifs in terms of binding are `CTCTTT, CTTCTT, CAAAAT`. When comparing conservation and freeness, then surprising motifs in terms of freeness are `TTTACT, CAAAAT, TTTCCC, TTCCTT`. When comparing binding and freeness, then there are no such motifs, that would have too much difference in p-values. Let us now list some motifs returned by our application. The columns `cons, bind, free` are the conservation, binding and freeness scores of the motifs. Columns `p_cons, p_bind, p_free` are corresponding p-values.

```
   MOTIF  COUNT  cons    bind    free    p_cons  p_bind  p_free
1  GAAAAA   50   1.235   1.160   0.452   0.000   0.000   0.000
2  AAAAAA  137   3.289   3.351   0.446   0.000   0.000   0.000
3  TTTTTT  123   2.865   3.007   0.434   0.000   0.000   0.000
4  TTCTTT   30   0.665   0.662   0.413   0.032   0.000   0.000
5  TTTTTC   43   0.902   0.973   0.405   0.000   0.000   0.000
...
93 GAAAAT   15   0.397   0.313   0.221   0.583   0.208   0.259
94 ATTAAT   17   0.412   0.341   0.221   0.494   0.389   0.255
95 GAGAAA   23   0.584   0.487   0.219   0.058   0.006   0.317
96 AAAGTT   15   0.365   0.273   0.216   0.510   0.374   0.321
97 CTTCTT   16   0.249   0.331   0.216   0.924   0.324   0.315
```

And let us list some documented binding sites [MOJ+08]: `AAGAAAAA, CTTCC, AGGGG, CCAGC, TTTTCGCT, ATGGAT, CCCCT, CTCGA, GGTAC, CTCAC, CGCCTC`. Although we do not see many similarities, then motifs `GAAAAA` and `TTCTTT`, which is the complementary of `AAGAAA`, seem to partially match `AAGAAAAA` in the list of documented binding sites. Motifs, that had high p-values mostly in respect to one data track seen in Figure 4.4 like `CTCTTT, CTTCTT`, seem to be similar to `CTTCC, CCCCT`. Although there are similarities, it is not possible to make any strong conclusions based upon these results. On the other hand, the significant motifs suggested by our algorithm did not seem to very misleading. The first motif `GAAAAA` matched part of one longer documented motif `AAGAAAAA`. Of course, questionable motifs are `AAAAAA` and `TTTTTT`, because they seem to be in the top only because they had most matches in the input data. This is actually a side-effect of additive and maximal types of support, because these motifs match long consecutive elements with same letters and therefore introduce bias in the support. Of course, this means also that these motifs have great structural importance, but they are not exactly what we are looking for in gene regulation problems. Different approaches of determining significance and mining with average type of support mentioned in Chapter 2 could reduce this bias, but it is the material for further research and out of the scope of this work.

To sum it up, our tool seems to have great potential mining significant motifs from many promoter sequences. Still, while much research remains to be done in this area, our tool can be helpful for scientists to help confirming existing documented results or even suggest motifs that may need further attention.

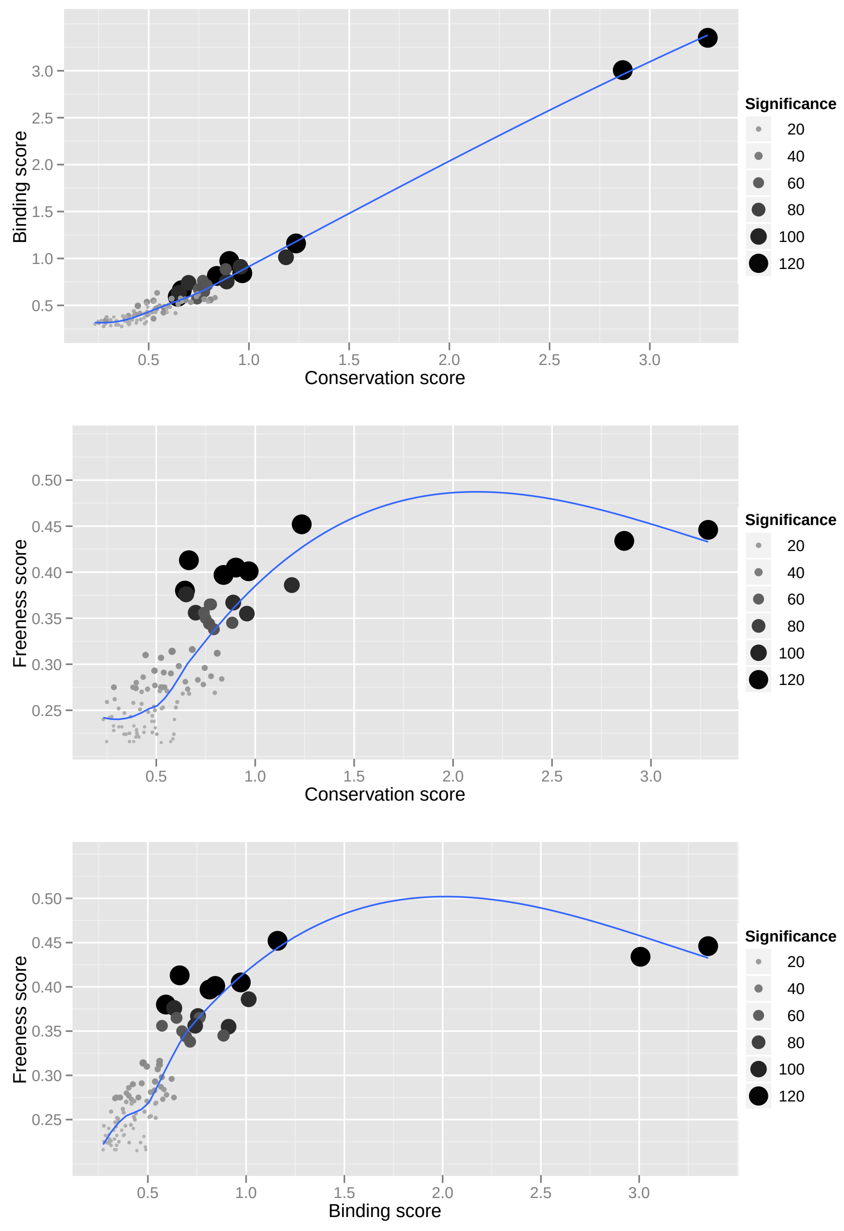


Figure 4.3: Relations between conservation, binding and freeness scores among 100 most frequent motifs. The larger dots have higher *Fisher* statistic, being more significant.

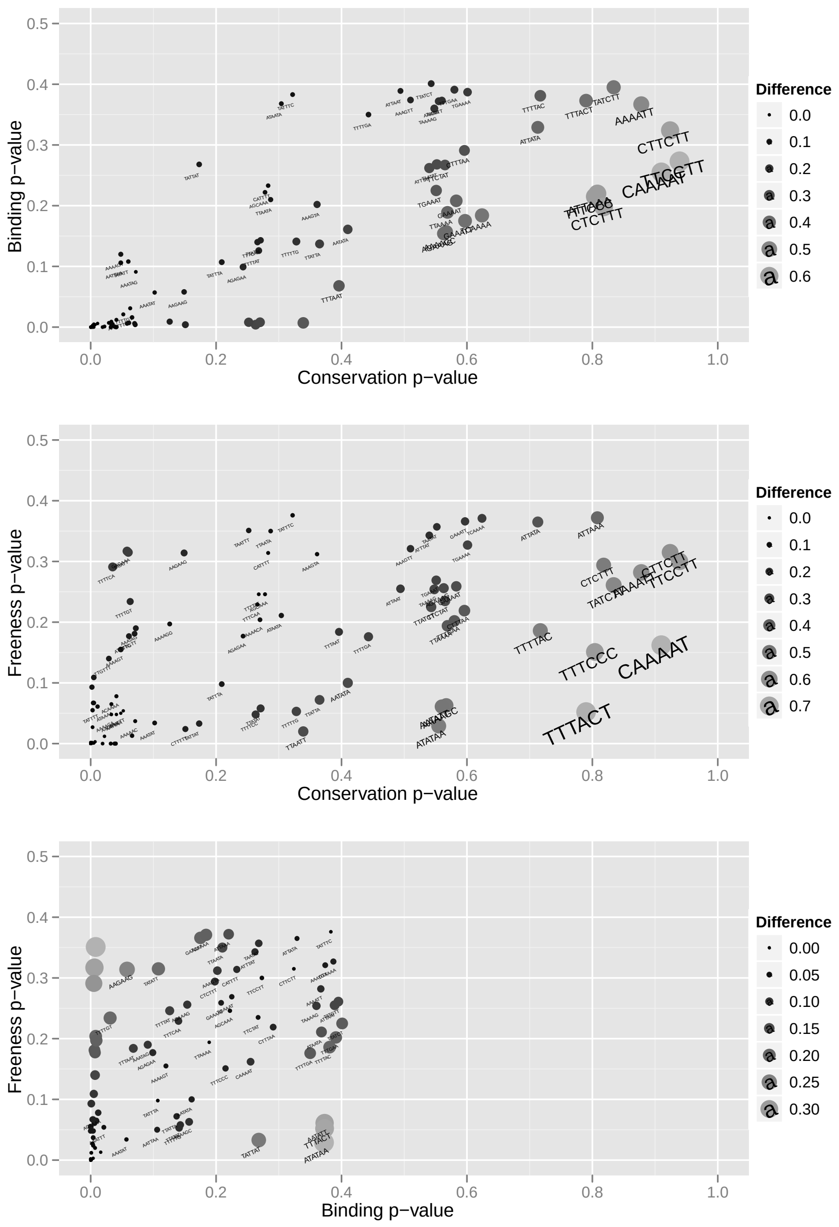


Figure 4.4: Relation between p-values of individual data tracks. Larger dots mean larger difference in p-values.

Summary

In this work, we introduced and developed novel mathematical formalization, algorithms and data structures needed to describe data mining methods using multiple input promoter sequences and several layers of data. We reformulated standard sequence mining techniques and studied different properties of our new formalization in Chapter 2. We also discussed briefly a method to decide the statistical significance of frequent motifs.

In Chapter 3, we described compact encoding of fixed length motifs. We gave instructions, how to efficiently construct hash-maps containing support metrics of motifs. We discussed NaiveSearch and several improvements to it like MaxSupSearch, SafeApproxSearch and InfreqSearch. Next, we introduced NBest algorithm for mining fixed number of frequent motifs. We also described a data structure called GFP-Tree and algorithm GFP-Search, that is able to mine frequent motifs, but is optimized for telling the number of frequent motifs in the dataset. We also discussed briefly, how SigMotifs and NBest can be fused together to provide information of statistical significance of motifs.

We wrote a C++ application and implemented all algorithms and data structures discussed in this work and benchmarked the runtime speed of the application and algorithms in Chapter 4, realizing that GFPSearch seemed be superior to other algorithms in terms of runtime speed. We also used real biological data and mined significant motifs of length 6 for gene MCM1. We concluded, that while much more research needs to be done, given proper input data, our search methods can provide meaningful results.

Motiivide otsimine DNA regulatiivsetest aladest

Bakalaureusetöö (6 EAP)

Timo Petmanson

Resümee

Käesolev töö uurib algoritme, mille abil on võimalik uurida organismide geeniregulatsiooni probleeme eksperimentaalsete andmete põhjal. Keskendutakse DNA regulatiivsetest aladest oluliste motiivide ning fragmentide otsimisele, millel võb olla kriitiline roll organismi elutalitluse reguleerimisel ja kordineerimisel.

Töö teoreetilises osas kirja pandud matemaatilise formalisatsiooni abil uuritakse ja tõestatakse mitmeid omadusi, mis panevad aluse võimalikele otsingualgoritmidele ja nende analüüsimisele. Töö praktiline osa käsitleb väljatöötatud algoritmide ajalist efektiivsust ning võimekust töötada bioloogiliste andmetega.

Bibliography

[Coh04] J. Cohen. Bioinformatics. An introduction for computer scientists. *ACM Computing Surveys*, 2004.

[DP07] G. Dong and J. Pei. *Sequence Data Mining.* Springer, 2007.

[Hee07] Dimitri Heesch. Doxygen - documentation system for various programming languages, 1997-2007.

[JJYR04] J.Han, J.Pei, Y.Yin, and R.Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approachâĹŮ. *Data Mining and Knowledge Discovery*, 8, 2004.

[KSF$^+$02] Kent, Sugnet, Furey, Roskin, Pringle, Zahler, and Haussler. The human genome browser. *Genome Res*, 2002.

[Lau09] S. Laur. Advanced sequence mining, October 2009. Available from http://courses.cs.ut.ee/2009/fast-counting/uploads/Main/sequence-mining-hints-ii.pdf.

[MHJ06] M.Sogin, H.Morrison, and J.Huber. Microbial diversity in the deep sea and the underexplored "rare biosphere". *National Academy Science USA*, August, 2006.

[MOJ$^+$08] M.Chollier, O.Sand, J. V. Turatsinze Janky, M. Defrance, E. Vervisch, S. Brohee, and J. van Helden. RSAT: regulatory sequence analysis tools. *Nucleic Acids Res.*, 2008.

[MPP$^+$06] M.Teixeira, P.Monteiro, P.Jain, S.Tenreiro, A.Fernandes, N.Mira, M.Alenquer, A.Freitas, A.Oliveira, and I.Correia. The yeastract database: a tool for the analysis of transcription regulatory associations in saccharomyces cerevisiae. *Nucl. Acids Res*, 2006.

[MT96] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurrences. In *Knowledge Discovery and Data Mining*, pages 146 – 151, 1996.

[MTV95] H. Mannila, H. Toivonen, and A. Verkamo. Discovering frequent episodes in sequences. In *In First International Conference on Knowledge Discovery and Data Mining*, pages 210 – 215. AAAI Press, August 1995.

[NB08] Newburger and Bulyk. Uniprobe: an online database of protein binding microarray data on protein-DNA interactions. *Acids Res*, 2008.

[NIY$^+$09] N.Kaplan, I.K.Moore, Y.Mittendorf, A.Gossett, D.Tillo, Y.Field, E.LeProust, T.Hughes, J.Lieb, J.Widom, and E.Segal. The DNA-encoded nucleosome organization of a eukaryotic genome. *Nature*, March 2009.

[RSM05] R.Deonier, S.Tavare, and M.Waterman. *Computational Genome Analysis. An Introduction*, chapter 2 and 9. Springer Science and Business Media, LLC, 2005.

[Vil02] J. Vilo. *Pattern Discovery from Biosequences.* PhD thesis, University of Helsinki, 2002. ISBN 952-10-0819-9.

[WCU07] The World Conservation Union. Red list of threatened species. summary statistics for globally threatened species, 2007.

Appendix A

Multi-constraint miner tool for gene expression analysis

We needed an implementation of the studied algorithms for run time speed bench-marking and working with biological data. Thus, we decided to develop an application, that could be used for such purposes. The source code of the application along with data preprocessing scripts is available for downloading at `http://mcminer.sourceforge.net`.

Features:

1. Mine from up to 128 promoter sequences using up to 8 different data tracks (these settings can be changed by modifying the source code).
2. Choose between different search algorithms: NaiveSearch, MaxSupSearch, SafeApproxSearch, InfreqSearch, GFPSearch, NBest and SigMotifs.
3. Set maximal or additive support for different data tracks.
4. Set thresholds on every data track.

Supported platforms: The application is written and tested only on Fedora 12, but it should be possible to build it on all platforms that are supported by GCC 4.3 and Boost 1.39 libraries.

Documentation: The documentation of the source code can be generated with Doxygen tool [Hee07], instructions for building and using the applications, description of the file format the program uses to read promoter data are given in `README` file of the project.

License: The application is released under the GNU General Public License (version 3).