Efficient and Accurate Discovery of Patterns in Sequence Datasets

Avrilia Floratou [#1], Sandeep Tata [*2], Jignesh M. Patel [#1]

[#] *Computer Sciences Department, University of Wisconsin-Madison, WI, USA.*
[1] {floratou,jignesh}@cs.wisc.edu

[*] *IBM Almaden Research Center, CA, USA.*
[2] stata@us.ibm.com

***Abstract*— Existing sequence mining algorithms mostly focus on mining for subsequences. However, a large class of applications, such as biological DNA and protein motif mining, require efficient mining of "approximate" patterns that are contiguous. The few existing algorithms that can be applied to find such contiguous approximate pattern mining have drawbacks like poor scalability, lack of guarantees in finding the pattern, and difficulty in adapting to other applications. In this paper, we present a new algorithm called FLAME (FLexible and Accurate Motif DEtector). FLAME is a flexible suffix tree based algorithm that can be used to find frequent patterns with a variety of definitions of motif (pattern) models. It is also accurate, as it always find the pattern if it exists. Using both real and synthetic datasets, we demonstrate that FLAME is fast, scalable, and outperforms existing algorithms on a variety of performance metrics. Using FLAME, it is now possible to mine datasets that would have been prohibitively difficult with existing tools.**

I. Introduction

In a number of sequential data mining applications, the goal is to discover frequently occurring patterns. To illustrate the characteristics of such an operation, consider Figure 1. This figure shows the percentage change in the stock price for a company over the previous minute's average price, for several minutes in a day. An interesting mining question on this dataset is: "Are there any frequently recurring patterns in this time series dataset?" Finding such patterns in stock price data can provide valuable insights that inform trading strategies. In Figure 1, the bold segments highlight a pattern that occurs four times in the dataset. Note that the recurring subsequences are similar, but not identical. The challenge in discovering such patterns is to allow for some *noise* in the matching process. At the heart of such a method is the definition of a pattern, and the definition of similarity between two patterns. This definition of similarity can vary from one application to another. A simple approach in the case of data such as in Figure 1 is to define a tolerance value, ϵ, and consider two sequences to be similar if the corresponding numerical values in the sequences are within ϵ of each other.

This approximate subsequence mining problem is of particular importance in computational biology, where the challenge is to detect short sequences, usually of length 6–15, that occur frequently in a given set of DNA or protein sequences. These short sequences can provide clues regarding the locations of so called "regulatory regions", which are important

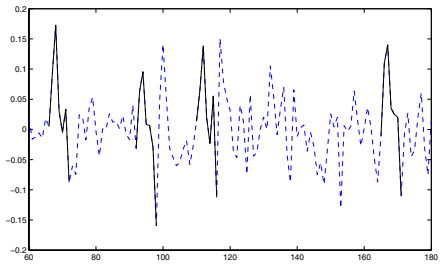

Fig. 1. Stock Data: A frequent approximate pattern is highlighted in bold.

repeated patterns along the biological sequence. The repeated occurrences of these short sequences are not always identical, and some copies of these sequences may differ from others in a few positions. The similarity metric that is used here could be complex – for example, when comparing proteins, a similarity matrix like PAM [1] or BLOSUM [2], may be used for comparing the "distance" between each symbol (protein) pair. These frequently occurring patterns are called *motifs* in computational biology. In the rest of this paper, we use this term to describe *frequently occurring approximate sequences*.

Clearly, different applications require different similarity models to suit the kind of noise that they deal with. It is desirable for a motif mining algorithm to be able to deal with a variety of notions of similarity. In this paper, we present a powerful new model for approximate motif mining that fits several applications with varying notions of approximate similarity, including the examples described above. We also present FLAME (**FL**exible and **A**ccurate **M**otif D**E**tector) – a novel motif mining algorithm which can efficiently find motifs that satisfy our model.

We note that the problem of motif mining is related to the problem of mining for frequent itemsets [3], and frequent subsequences [4]. The problem of finding frequently occurring (non-contiguous) subsequences in large sequence databases has been extensively studied in previous works [4]–[8]. Traditionally, B is called a subsequence of A, if B can be constructed by projecting out some of the elements of sequence A. For instance, if A is the sequence "a,b,a,c,b,a,c", the sequence "a,b,b,c" is a subsequence constructed by choosing the $1^{st}, 2^{nd}, 5^{th}$, and 7^{th} elements from the original

sequence and omitting the rest. While mining for frequent *non-contiguous* subsequences has many uses, it is not appropriate for many applications such as DNA and protein motif mining. A subsequence constructed by gluing together distant parts of the original sequence is not meaningful in these applications. In mining for motifs, we are interested in *contiguous* subsequences. Furthermore, previous work on non-contiguous subsequence models cannot easily incorporate noise tolerance in the way that contiguous motif models can. In short, subsequence mining and motif mining are different data mining operations, and there are distinct applications of each of these. This paper focuses on the contiguous subsequence (motif) mining problem. Readers closely familiar with traditional (non-contiguous) subsequence mining algorithms may note that some of these methods can be adapted to mine for contiguous subsequences (e.g. [6], [7], [9], [10]). In the extended version of this paper [11], we compare our method with some of these methods, and show that FLAME is faster by an order of magnitude.

Motivated by the problem of finding frequent patterns in DNA sequences, which has profound importance in life sciences, the computational biology community has developed numerous algorithms for detecting frequent motifs using the Hamming distance notion of similarity. YMF [12], Weeder [13], MITRA [14], and Random Projections [15] are examples of algorithms in this category. Compared to this class of algorithms, we show that FLAME is more flexible, and can use more powerful match models. We also demonstrate through empirical evaluation that FLAME is more scalable than these existing methods and can be an order of magnitude faster for mining large databases.

There are several applications of motif mining in addition to those mentioned above. It is often the first step in discovering association rules in sequence data ("basic shapes" in [16] and "frequent patterns" in [17]). It can also be used to find good seeds for clustering sequence datasets [18]. Records of medical signals, like ECG or respiratory data [19] from patients can also be mined to find signals that can indicate a potentially critical condition.

We make the following contributions in this paper:

1) We present a powerful new model that is very general and applicable in many emerging applications. We demonstrate the power and flexibility of this model by applying it to datasets from several real applications.
2) We describe a novel motif mining algorithm called FLAME (**FL**exible and **A**ccurate **M**otif D**E**tector) that uses a concurrent traversal of two suffix trees to efficiently explore the space of all motifs.
3) We present a comparison of FLAME with several existing algorithms (YMF [12], Weeder [13], and Random Projections [15], [20]). FLAME never misses any matches (as opposed to some of these methods that apply heuristics). In fact, we show that FLAME is able to identify many true biological motifs that existing algorithms miss.
4) We show that our algorithm is scalable, accurate, and often faster than existing methods by more than an order of magnitude!

The remainder of the paper is organized as follows: Section II presents related work, and Section III describes our model for motifs. In Section IV, we present the FLAME algorithm. Section VI contains our experimental results, and Section VII contains our conclusions.

II. Related Work

There is a vast amount of literature on mining databases for frequent patterns [21]–[23]. Early work focused on mining association rules [3]. The problem of mining for subsequences was introduced in [4]. Subsequence mining has several applications, and many algorithms like SPADE [5], BIDE [6], CloSpan [7] (and several others) have been proposed as improvements over [4]. Yang et al. [8] use a statistical sampling based method with a compatibility matrix to find patterns in the presence of noise. However, they primarily focus on subsequence mining, while we focus on contiguous patterns.

Some subsequence mining algorithms allow certain constraints. Constraints which limit the maximum gap between two items in the subsequence make it possible to use these algorithms to mine for contiguous patterns. Algorithms like EXMOTIF [24] and RISO [25] are designed to efficiently find multi-motifs, i.e. simple motif patterns separated by variable length spaces. FLAME does not target the multi-motif problem, but can be be used as a building block for multi-motif mining. Algorithms such as cSPADE [9], CloSpan [7], Pei et al. [10], [26] can be adapted to mine for exact contiguous motifs. An obvious reason why these are unsuitable for approximate frequent pattern mining is that these algorithms do not include a notion of noise or an approximate match. Furthermore, they tend to be inefficient even when used for exact substring mining. FLAME, on the other hand is extremely efficient even for approximate substrings.

The vast body of work in bioinformatics for finding patterns in long noisy DNA sequences [27], can be divided into two classes – pattern based and statistical. The patterns based algorithms typically search through the space of potential patterns and find a motif that satisfies the minimum support. Marsan and Sagot [28] proposed a suffix trie based algorithm to find structured motifs tolerating a few mismatches as noise. This method is primarily focused at finding pairs (or sets) of motifs that co-occur in the dataset within a short distance of each other. This method only considers a simple mismatch based definition of noise, and does not consider other more complex motif models such as a substitution matrix or a compatibility matrix as in [8]. Furthermore, Marsan and Sagot do not have optimizations, such as the ones we describe in Section V. These optimizations make FLAME faster by an order of magnitude. Zhu et al. [29] proposed an algorithm for mining approximate substrings but it only accomodates the Hamming distance model.

Several other algorithms such as the Yeast Motif Finder [12] (YMF), Weeder [13], MITRA [14] have been used for finding motifs. YMF is a simple algorithm that computes the statistical

significance of each motif. YMF scales very poorly with increasing complexity of motifs, and thus cannot be easily adapted to other applications. Weeder is a suffix tree based algorithm that makes certain assumptions about the way the mismatches in an instance of the motif are distributed. This makes Weeder extremely fast, but it is not guaranteed to always find the motif. Weeder too, cannot be adapted for other motif models. MITRA is a mismatch tree based algorithm which uses clever heuristics to prune the large space of possible motifs. MITRA is very resource intensive and requires large amounts of memory.

Statistical approaches use techniques such as Expectation Maximization [30], Sampling [31], Random Projections [15], etc. to search for frequent patterns in the data. All of these heuristic approaches run the risk of finishing at a local optimum, and may not be able to find the right motif. Furthermore, these methods are specifically tailored for the problem of simple mismatch based motifs, and cannot easily be adapted for more complicated models.

The comprehensive study by Tompa et al. [32] compared several different statistical and pattern based algorithms on a variety of real and synthetic datasets, and identified Weeder [13] and YMF [12] as the most effective methods. In our evaluations, we compare with these two methods.

A host of techniques have been developed to find sequences in a time series database that are similar to a given query sequence [33]–[36]. However, there is little published work in finding motifs in time series databases. Time series data such as stock prices, economic indexes, time varying measurements from sensors and medical signals like ECG's can be mined for motifs, and all have compelling applications. Patel et al. [18] show that time series data can be discretized and converted into a sequence over a fixed alphabet and mined using existing motif mining algorithms. Another algorithm that finds frequent trends in time series data was proposed by Udechukwu, Barker, and Alhajj in [37]. However, these algorithms mine for *exact* frequent patterns, and are difficult to employ in the case of noisy datasets. Chiu et al. describe an algorithm in [20] (based on the Random Projections algorithm [15]) which accounts for noise in the data. However, this algorithm is also limited to a simple mismatch based noise model. In addition, this is a probabilistic algorithm, and is not always guaranteed to find all existing patterns. FLAME, on the other hand provides the option of a variety of models, and is guaranteed to find the motif (i.e. it is an accurate algorithm and not a heuristic method).

III. The Model

A critical aspect of the motif mining problem is defining the model under which two or more sequences are considered to match (approximately). Developing such models poses an interesting challenge: On the one hand, we want a model that is robust enough to detect the occurrence of a pattern even in the presence of noise, and on the other hand, we do not want it to be so general that it matches unrelated subsequences. Since different applications may have different criteria for how to strike this balance, a natural approach is to develop a flexible model with a few intuitive parameters that can be set by the user based on the application characteristics. In this section, we present a powerful new model for motifs that can be used for pattern mining in many different domains.

Throughout this section, we will assume that the input sequence is composed of symbols from a discrete alphabet set. However, our methods can also be applied to continuous time series datasets by converting such datasets into a *symbolic* sequence dataset by simply discretizing the numeric data. In fact such a transformation is frequently carried out for mining continuous time series datasets [18], [20].

We call our motif model the (L, M, s, k) model after the four parameters that determine it. L is the length of the motif, M is a distance matrix that is used to compute the similarity between two strings, s is the maximum distance threshold within which two strings are considered similar, and finally, k is the minimum support required for a pattern to qualify as a motif.

The (L, M, s, k) model is a very intuitive and powerful model, and permits the user a lot of flexibility in making the right tradeoff between specificity and noise tolerance of a model. As we describe below, much of this power comes from the ability to use any matrix M as the distance matrix. This property makes it useful for a variety of complex motif mining tasks. The matrix M allows us to define a *distance penalty* when a symbol X in the model matches a symbol Y in the data sequence. The penalty is specified by M(X,Y), an entry in the matrix. The total distance between the two strings is computed by summing the distance penalties of the corresponding symbols. That is, if $A = a_1a_2a_3...a_n$ and $B = b_1b_2b_3...b_n$ are two strings, then the distance between A and B under this model is $d(A, B) = \Sigma_{i=1}^{n} M(a_i, b_i)$.

Formally speaking, a string S is an (L, M, s, k) motif if there exist at least k strings $T_1, ..., T_k$ in the database such that each of them is of length L, and $d(S, T_i) \leq s$, where $d(A, B) = \Sigma_{i=1}^{n} M(a_i, b_i)$ is the distance function. Every string S that satisfies the above is an (L, M, s, k) motif. Note that the string S need not actually appear in the database for it to qualify as a motif. Only the instances T_i need to be in the database.

Protein motif mining is an example of a domain which requires a matrix based measure of similarity. Finding regions in protein sequences that appear frequently in different proteins is useful in inferring the functional sites in proteins. As in the case of DNA, the patterns in protein sequences do not repeat exactly. The instances of the pattern usually differ from the model in a few positions. To complicate things further, not all mismatches are equally bad. Some amino acids are very similar to each other, while some are very different. For instance Alanine and Valine are both hydrophobic amino acids, while Glycine and Serine are both hydrophilic. The matrix can be used to award a small penalty for M(X,Y) when X and Y are similar (Alanine and Valine, for instance) and a larger penalty otherwise (say, Alanine and Glycine) [2]. Popular substitution matrices such as PAM [1] and BLOSUM [2] can

easily be used in our model.

Next, we demonstrate how this model can also be applied to the stock price example of Section I. Suppose that we had normalized the data for firm ABC. Assume that the normalized stock price values are between 0-10. If we discretized them to integers, we could use letters A – K to represent 0 – 10. Suppose further that we wanted to find sequences of length 10 that appeared (approximately) in the database at least 20 times. If we wanted to use the sum of squared differences as the distance metric to check for similarity, we can simply use a matrix where M(X,Y) is set to $(v(X) - v(Y))^2$ where v(X) is the numerical value corresponding to the symbol X. Using this matrix, we can specify that an instance matches the model if the Euclidean distance between them is within a given threshold. We model this problem as a $(10, M, s, 20)$ motif finding problem, where s is an appropriately chosen similarity threshold.

The matrix can be adapted to allow other kinds of models. In fact, the matrix approach lets us simulate any L_p-norm (Manhattan distance, Euclidean distance, etc.). If we wanted to match two sequences only if the corresponding values (in the two sequences) were within 2 units of each other, (the ϵ-error tolerance model from Section I), we would just set M(X,Y) = 0 where $|v(X) - v(Y)| \leq 2$, and ∞ everywhere else. In general, any measure that can be computed in an incremental fashion by comparing the symbols in the corresponding positions can be simulated by constructing an appropriate distance penalty matrix.

We now discuss two special cases of the (L, M, s, k) model that are commonly used in computational biology and other domains - the (L, d, k) and (L, f, d, k) models.

A. Special Case: The (L, d, k) Model

The (L, d, k) model is a mismatch based model commonly used in computational biology for finding DNA motifs. The distance measure between two strings is the Hamming distance, or merely the number of mismatches. The (L, d, k) model is parameterized by the length of the string that we want to find (L), the maximum Hamming distance (d), and the support (k). The parameter d controls the amount of noise we wish to tolerate.

The (L, d, k) model is a special case of the (L, M, s, k) model. It can easily be simulated by a matrix by setting $M(X, Y) = 1$ if $X \neq Y$ and $M(X, Y) = 0$ if $X = Y$. This way, the distance function simply counts the number of mismatches. We set s to d and use the k from (L, d, k) as our minimum support.

One of the applications of this model is in the field of computational biology. The (L, d, k) model and its derivatives have been considered a good fit for DNA regulatory motifs [32]. Briefly, the related problem of using this model to find regulatory motifs in DNA is as follows: Biologists today are interested in understanding how different genes in the genome are regulated and the way they interact with each other. To this end, biologists often study genes that exhibit similar expression patterns to extract clues about the proteins that control their expression. It is believed that genes that are co-regulated by the same protein (called a transcription factor) share some signal that allows the transcription factor to recognize the gene and turn it on. This signal is usually present in the region upstream of a gene (within a few thousand base pairs) called the promoter region. The signature is usually a short string of DNA 6-15 bases long. As is often the case in biology, these signatures are seldom identical, and differ in a few positions from one gene promoter region to another. Finding this noisy signature that is common across all the genes is a very important step towards locating the binding site for the transcription factor. Modeling the set of promoter regions as our database, and the signature binding site as an (L, d, k) pattern, we can simply apply the FLAME algorithm to solve this problem. We show in Section VI, that this is indeed an effective approach.

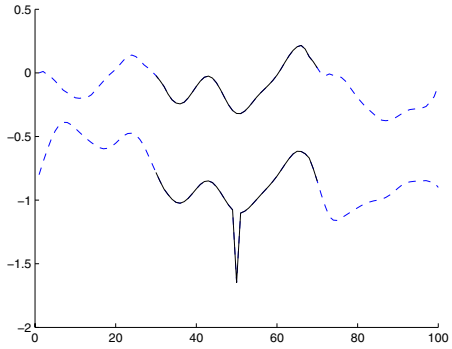

Fig. 2. Potential use of the (L, d, k) model – the lower segment is identical to the upper segment except for the single spike. The (L, d, k) model can match these.

In most practical situations we don't know the exact value of L, and therefore, we might have to try several values. In the case of DNA regulatory patterns, we know that the signature is usually between 6 to 15 bases long, and therefore we can try these lengths with varying number of mismatches.

The (L, d, k) model can also be used in other applications to tolerate an occasional burst of noise. If two sequences were identical except for the addition of a noise spike in one of them, they will match under a 1-mismatch model. Consider the two sequences shown in Figure 2. The two bold segments are identical except for the single spike in the lower sequence. Such spikes may occur due to measurement error or other reasons, and an (L, d, k) model will be able to tolerate this noise and correctly match the two sequences.

We show in Section VI that FLAME is faster than several existing algorithms that can only find (L, d, k) motifs.

B. Special Case: The (L, f, d, k) Model

The (L, f, d, k) builds on the (L, d, k) model to include positional constraints on the mismatches. We introduce this model using an example: Consider the three sequences {ABCD, ACCD, ABCA}. If ABCD is the model sequence, the other two sequences are within one mismatch of the motif, so these sequences would constitute a (4,1,3) motif

in the (L, d, k) model. Now consider the sequences {ABCD, ACCD, ADCD}. This set also forms a (4,1,3) motif, but the mismatches, whenever they occur, are always in position two (AcCD, AdCD). The (L, f, d, k) model allows us to specify the number of $fixed - position$ mismatches (f) along with just the number of free mismatches (d). This allows us to screen out patterns of the latter kind. In other words, instead of allowing a mismatch anywhere in the substring, we look for all model strings whose instances always differ from it (if they differ at all) in the same positions.

The (L, f, d, k) model is also a special case of the (L, M, s, k) model. In order to model the fixed position mismatches, we simply augment the alphabet A with a wildcard symbol, say "?". For symbols in A, the distance matrix M is as in the (L, d, k) model, with $M(X, Y) = 1$ if $X \neq Y$ and zero everywhere else. The wildcard symbol is allowed to match any symbol with no penalty, so we set $M(?, X) = 0$ for all X. FLAME considers all model strings of length L over the augmented alphabet such that there are at most f occurrences of the wildcard symbol. This way, the (L, M, s, k) model can simulate the (L, f, d, k) model.

In general, this model is useful in applications where the noise has a positional bias as it allows us to be more specific in finding the right patterns while ignoring extraneous matches. Some DNA motif finding applications [32] use models that are somewhat similar to the (L, f, d, k) model.

We illustrate the advantage of being able to use positionally biased scoring with an example. Consider a DNA dataset consisting of 5 sequences,each of length 500. Assume that each sequence has in it the motif GTGAACAC, and each instance of the motif has a mismatch at the fifth position. In other words, the dataset contains an (8,1,0,5) motif. Note that an (8,1,0,5) motif is also an (8,1,5) motif in the (L, d, k) model since a free mismatch can capture a fixed mismatch. If we use the (L, d, k) model to retrieve this pattern, we will end up with many extraneous hits that might not be meaningful. When we search for an (8,1,0,5) pattern, FLAME (correctly) returns the result GTGA?CAC. On the other hand,if we search for (8,1,5), FLAME returns several additional hits that satisfy (8,1,5) but not (8,1,0,5). A post-processing step is needed to check if these are actually fixed position mismatch motifs. An (L, d, k) model can be used to simulate an (L, f, p, k) model if $f + p = d$ with some post processing. However, as we will explain in Section V-B , using an (L, f, d, k) model produces a huge cost saving when compared to $(L, d + f, k)$ with post-processing.

IV. The FLAME Algorithm

In this section, we describe the FLAME algorithm, which can be used to find (L, M, s, k) motifs. For ease of exposition, we explain the algorithm using an (L, d, k) model, and then describe how we extend it to the full-fledged (L, M, s, k) model.

Recall that an (L, d, k) motif is a string of length L that occurs k times in the dataset, with each occurrence being within a Hamming distance of d from the model string. Given, L, d, and k, a naive algorithm is to consider all possible strings of length L over the alphabet (the space of all models), and compute the support for each of them by scanning the dataset. This algorithm is exponential and becomes infeasible with large L and d values. One might be tempted to improve this method by considering only those strings of length L that actually occur in the dataset. However, this approach might miss motifs as the model string might not actually occur in the dataset even once. To illustrate this point, suppose that the string ABCDEF is the true motif. Assume that we are looking for a $(6, 2, 3)$ pattern, and that the instances of this pattern in the dataset are FFCDEF, ABFFEF, and ABCDAA. Each instance is at a distance of 2 from the model ABCDEF, but the distance between any two instances is 4. If we consider only instances from the dataset (which need not contain ABCDEF), then we will not find the motif.

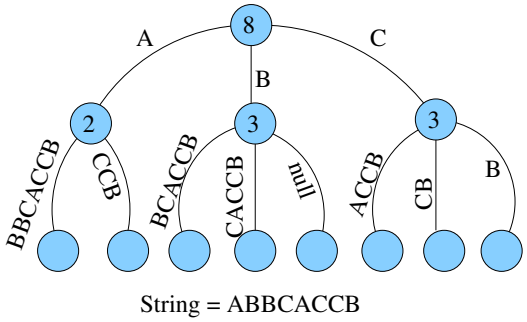

Fig. 3. A count suffix tree on the string ABBCACCB. The counts are indicated inside the node.

The approach we take in FLAME explores the space of *all* possible models. In order to carry out this exploration in an efficient way, we first construct two suffix trees: a count suffix tree on the actual dataset (called the *data suffix tree*), and a suffix tree on the set of all possible model strings (called the *model suffix tree*). This second set is typically the set of all strings of length L over the alphabet. As we describe below, the model suffix tree helps guide the exploration of the model space in a way that avoids redundant work. The data suffix tree helps us quickly compute the support of a model string. Recall that a count suffix tree is merely a suffix tree in which every node contains the number of leaves in the subtree rooted at that node. In other words, every node contains the number of occurrences of the string corresponding to that node. Essentially, the data suffix tree combines the work common to finding the support for models like ABCDE and ABCDF (having a common prefix) and perform it only once.

Since the second suffix tree (built on all possible model strings) can be extremely large, FLAME does not actually construct this suffix tree. Rather, it algorithmically generates portions of this tree as and when needed. FLAME then explores the model space by traversing this (conceptual) model suffix tree. Using the suffix tree on the dataset, FLAME computes support at various nodes in the model space and prunes away large portions of the model space that are guaranteed not to produce any results under the model. This careful pruning (described in more detail below), ensures that FLAME does

TABLE I

THE LIST OF MATCHES FOR THE MODEL A

Node	Number of mismatches	Count
A	0	100
B	1	50
C	1	45
D	1	120
E	1	15
Support	-	330

not waste any time exploring models that do not have enough support. The FLAME algorithm simply stops when it has finished traversing the model suffix tree and outputs the model strings that had sufficient support.

To understand our strategy of pruning the model suffix tree, consider the following example: Assume that the dataset consists of sequences over the alphabet {A,B,C,D,E}. The dataset and the values of L, d, and k are specified as input. All the strings of length L starting with the symbol A form a subset of the model space. We call this the A partition. This partition corresponds to all the nodes in the model suffix tree under the subtree corresponding to node A. This partition is further divided into sub-partitions with prefix AA, AB, AC, AD, and AE. These partitions continue on for L levels, and at the last level, we have only one model string for each partition.

Suppose that we start by considering the models in partition A. Assuming no mismatches are allowed, if the support for A is less than k, then, clearly any model that starts with A cannot qualify as a valid motif since there will be fewer than k instances of it, and it will not have the minimum support. Consequently, we can safely toss away the entire space of models starting with the symbol A. This step essentially prunes away the subtree corresponding to A in the model suffix tree. After pruning A, we proceed to consider the B partition. An important step here is to compute the support for models starting with A. This value is simply the number of times A occurs in the dataset, and this value can be quickly looked up from the count suffix tree on the dataset.

When mismatches are allowed, computing the support of a (partial) model string is more complicated. Suppose that $d = 1$. When considering matches for models starting with A, we cannot rule out strings that start with B (or any other symbol), since a string starting with B could match a model starting with A by only differing in the first position. Now assume that the data suffix tree nodes at depth 1 labeled A, B, C, D, and E have counts of 100, 50, 45, 120, and 15 respectively. The possible number of strings starting with B that could match a model starting with A is simply the count of node B, namely 50. In a similar fashion, the count value from other nodes at most d mismatches away is read, and a list of potential matches for A is constructed as shown in Table I. The list contains the node in the data suffix tree, the number of mismatches corresponding to this node, and the count from that node. For instance, node A in the data suffix tree has a count of 100 and perfectly matches the model string (A) - we store this

```
FLAME (modelTree, dataTree, l, d, k)
1. model = modelTree.FirstNode()
2. While (model ≠ modelTree.LastModel())
3.     Evaluate_Support(model,dataTree)
4.     If ( isValid(model) ) Print "Found Model: ", model
5.     Else If(model.support() < k)
6.         modelTree.PruneAt(model)
7.     model = NextNode(model,modelTree)
8. End While
9.End

Sub Evaluate_Support (model, dataTree)
1. newsymbol = last symbol of model.String
2. oldmatches = model.Parent().Matches()
3. newmatches = EmptyMatches()
4. If (model.Parent() == root)
5.     newmatches = Expand_Matches(root,newsymbol,dataTree)
6. Else
7.     ForEach match x in oldmatches
8.         newmatches = newmatches U
               Expand_Matches(x,newsymbol,dataTree)
9.     End ForEach
10.model.SetMatches(newmatches)
11.Return

Sub Expand_Matches (x, newsymbol, dataTree)
1. Let Y = Set of all single character expansions of x.String
       in dataTree
2. ForEach element b in Y
3.     If b's last symbol ≠ newsymbol
4.         b.mismatches ++
5.         If b.mismatches > max_mismatches
6.             Remove b from Y
7.     End ForEach
8. Return Y
```

Fig. 4. The FLAME Algorithm

information in the list as (A, 0, 100). The total support for the partial model is now computed by summing up the individual counts. In the example for Table I, this sum is 330. Those nodes where the number of mismatches with the model being considered is greater than d are pruned away and not included in the list of matches. The algorithm then proceeds to consider the next partial model – AA.

Observe that the list of matches for any partial model can be constructed incrementally using the list of matches for that model's longest prefix. For instance, the list of matches for AC can be constructed using the list for A(Table I). We take each string from the list, and extend it by one symbol. The first string A, for instance can be extended by one symbol to AA, AB, ..., AE. The string AC has 0 mismatches to itself, the remaining strings have 1 mismatch each. The support for each of these string can be quickly looked up in the count suffix tree. We locate the model suffix tree node corresponding to A (stored in the list of matches). This node points to its children: AA, AB, ..., AE. The support for each of them is read from the suffix tree, and a new list of matches is constructed for AC to compute its support. Similarly, when B is extended to length 2, all strings except BC have more than one mismatch

with the model string AC. Therefore only BC is included in the match list. The remaining nodes (C, D, and E) are expanded similarly.

We take advantage of this method for incrementally computing the support by traversing the model suffix tree in the depth first order. If $L = 3$, the partitions will be considered in the order A, AA, AAA, AAB, AAC, etc. At each node, the match list and the support for the parent node has already been computed, and can be used to compute the support of the current node.

Observe that if we want to distinguish between multiple matches within a single sequence or matches within different sequences, we can simply replace the count in each node of the count suffix tree with the count of sequence separator node in its subtree. That is, while building the count suffix tree, we simply store the number of distinct sequences the patterns occurs in instead of the total count. This allows FLAME to easily support both models.

The pseudocode for FLAME is given in Figure 4. The algorithm simply puts together the ideas described above. FLAME uses a suffix tree on the model space and a count suffix tree on the dataset. It starts by traversing the nodes of the model space in depth first order. At each node in the model suffix tree, the subroutine Evaluate_Support is called to compute the list of matches and the new support. This routine uses the match list from the parent node to speed up the computation. The routine Expand_Matches ensures that the number of mismatches to the model string does not exceed d. At any node, if FLAME discovers that the support is lower than k, it prunes away that subtree in the model suffix tree, and continues its traversal. If it finds a model of length L with the required support, it simply outputs the result.

The algorithm described in Figure 4 works with (L, d, k) models. For the (L, M, s, k) model, the Expand_Matches function becomes more sophisticated (Figure 5). Instead of merely keeping track of the number of mismatches, they keep track of the substitution distance score. That is, for each node, the match list stores $\Sigma_{i=1}^{n} M(x_i, y_i)$ where x_i is the symbol from the prefix of the partition, and y_i is the symbol it is being matched to in the data set. If this distance score exceeds the preset threshold (s), we prune the model suffix tree at that point, and continue the depth first traversal just as in the case of the simpler (L, d, k) model.

For the (L, f, d, k) model, we use the augmented alphabet to generate model strings that contain at most f wildcard characters and use the scoring matrix described in Section III.

V. Practical issues

When applying FLAME to a practical problem, there are opportunities for optimization one might exploit. Next, we describe a few techniques that can be used to great benefit.

A. Combining Computation for range of lengths

Very often in a real application, the exact length of the motif is not known apriori. One often merely has a rough idea of the range in which it may lie. For instance, in regulatory DNA motif finding, scientists believe that motifs are typically 6 to 15 bases long. One often ends up trying several (L, d, k) values such as $(6-15, 1, 20)$, $(6-15, 2, 20)$, $(6-16, 1, 15)$, $(6-15, 2, 15)$, etc. Given the way in which FLAME computes the support for various candidate models, the algorithm can easily combine the computation for many different lengths if the number of mismatches is common across all lengths.

Sub Expand_Matches_lMsk (*x, newsymbol, dataTree*)
1. Let Y = Set of all single character expansions of x.String in dataTree
2. ForEach element b in Y
3. b.distance += Distance_Matrix(b.lastsymbol,newsymbol)
4. If b.distance > max_distance
5. Remove b from Y
6. End ForEach
7. Return Y

Fig. 5. Functions for (L,M,s,k)

Recall that the suffix tree of all models is traversed in a depth first fashion. We build the suffix tree on all strings of length L_{max}. At any node, if the length of the model happens to be in the range of lengths considered, and the support is greater than the minimum support, we output that model, and *continue* the traversal. When we were considering only one length at a time, a valid model would only be found at a leaf node of the suffix tree since it consisted of strings only of length L. However, by allowing lengths in the range of L_{min} to L_{max} we can output valid models at depth starting at L_{min}.

This optimization can be applied in much the same way to the (L, f, d, k) model and the (L, M, s, k) model. We demonstrate the impact of this optimization in Section VI.

B. Optimizing opportunity with (L,f,d,k)

When mining a database for an (L, f, d, k) pattern, a special opportunity for speedier execution exists if $d = 0$. When $d = 0$, it means that the pattern must have all the mismatches in fixed positions and have no free mismatches. Therefore, instead of considering all string of length L with at most f wildcard characters (?) over the alphabet $AU\{?\}$, we can consider a smaller modelspace. We only consider those strings of length L that occur in the dataset with at most f of the characters replaced with a wildcard character. The absence of free mismatches guarantees that the model string actually occurs in the database, so we don't need to consider all possible strings over the alphabet.

This reduced modelspace can be constructed by enhancing the suffix tree on the data sequence by adding a node with its edge labeled "?" as a child node for every existing node. The algorithm proceeds as previously described with this new model tree. Before a (partial) model is evaluated, the algorithm checks to make sure that the number of "?"s is no greater that f.

As a result of this much smaller modelspace to consider, (L, f, d, k) searches can be orders of magnitude faster when $d = 0$. In the interest of space, we don't present these experimental results here.

VI. Evaluation

In this section, we present results from various experiments that were designed to test the effectiveness and performance of FLAME. We also compare FLAME with pattern mining algorithms from different application domains. Most existing algorithms can only work with (L, d, k) motifs and do not support the more general (L, M, s, k) model. Therefore, we carry out the comparison between FLAME and these existing methods using only the (L, d, k) model. Since we do not have a competing algorithm to compare the performance of FLAME on (L, M, s, k), we present a detailed analysis of the performance of FLAME as different parameters in (L, M, s, k) are varied.

We use a variety of datasets for our experiments:

Snake: This is a snake protein dataset from [38] that was considered for subsequence mining in [6]. It consists of 352 different snake venom protein sequences of varying lengths. The size of the dataset is about 28,000 symbols. The alphabet of amino acids (that make up the proteins) is of size 20. Such protein datasets are often analyzed in bioinformatics to find common patterns that might provide insights into their function.

Washington: The Washington dataset is actually a collection of 52 different datasets. It includes DNA sequences taken from several genes in Yeast, Mouse, Fruit Fly, and Humans, and also includes a few synthetic sequences. For a complete description, see [32]. The total size of this collection is 1.3 Million symbols.

IBM: This dataset contains second by second average price of IBM stock for all the trading days in December 1999 [39]. To reduce the noise in the detailed dataset, we preprocess the data using the following standard data processing techniques that are designed to deal with short term volatility in stock price information [40]: First, the data is converted into a minute wise average price using a sliding window. And next, the price values are transformed into a percentage change with respect to the price in the previous minute. This technique is routinely used to compare movement data across different stocks that have a different face value. The resulting dataset contained 21 sequences from 21 days, each of length approximately 400 numbers, totaling 8,400 numbers.

Synthetic: In order to fully explore the space of data sizes and alphabet sizes, we use a synthetic data generation method that has been extensively used in several previous efforts [13]–[15], [41]. The data is generated as follows: Given the alphabet size, the number of sequences, and the size of each sequences, we generate random sequences by uniformly drawing symbols from the alphabet. We then randomly choose k sequences and *implant* a pattern of length L with d mismatches at random positions in each of the k sequences. This results in a dataset containing an (L, d, k) motif. The sizes of datasets we generate are comparable to those used in previous related papers [13]–[15], [41].

All the experiments in this section were performed on a 2.8 GHz Intel Pentium 4 processor with 2 GB of main memory. The operating system was Fedora Core 4 Linux, kernel version 2.6.11. The YMF implementation was obtained from [42], Weeder from [43], and Random Projections (RP) from [44]. RP is a widely used technique for motif mining (and has been applied in various domains like time-series mining and DNA motif mining), and YMF and Weeder are the leading popular DNA motif mining methods [32].

As discussed in Section I, FLAME solves a different problem compared to traditional sequence mining methods like cSPADE [45] and CloSpan [7]. Nevertheless, for completion, we modified CloSpan to mine for contiguous subsequences (which improved its performance by 3 orders of magnitude) and adapted cSPADE to mine contiguous motifs. The comparison is presented in the extended version of this paper [11]. The results show that FLAME outperforms these methods by an order of magnitude or more and scales significantly better.

All suffix trees were constructed using the TDD suffix tree construction algorithm [46]. By using TDD we were able to build a suffix tree for a database size of 1.3 million symbols, which is the largest dataset used in the experiments, in approximately 3 seconds.

A. Comparison with Random Projections

The Random Projections (RP) algorithm of Bulher and Tompa [15] has recently been applied to time series data for motif mining [20]. RP is an approximate motif finding technique based on the idea of "locally sensitive hashing" from [47] that works only for the special case of (L, d, k) patterns. This algorithm has also been applied to finding DNA motifs and is considered faster [15] than several popular algorithms such as MITRA [14] and WINNOWER [41].

Given L, d, and k, the algorithm chooses a p-position mask as a hash function. Then, the algorithm hashes all the l-mers in the database. If a sufficient number of l-mers hash to the same bucket, it is likely that there is a motif that is similar to the l-mers in the bucket. Once a candidate bucket is identified, any local search algorithm can be used to search in the vicinity of the l-mers in the bucket for the (L, d, k) motif. In particular, RP uses an expectation maximization based algorithm like MEME [48] to search in the vicinity of "enriched" buckets. The main contribution in [15] is that they describe how to compute p, and the number of iterations for which the algorithm needs to be repeated for a certain level of confidence.

We compare FLAME and RP by performing a typical (L, d, k) motif mining task on synthetic DNA datasets of varying (following the well established methods that have been used before for similar comparisons [14], [15]). Each dataset contains 20 sequences. We vary the length of each sequence from 200 to 1000 symbols. The datasets are implanted with a motif of length between 8 and 14 (chosen randomly). The algorithms do not know the actual length of the motif in advance (as is the case in any real task [32]). Both algorithms try to find $(L, d, 20)$ motifs for d = 1, 2 and L varying from 8 to 14. FLAME takes advantage of the technique described in V-A to combine the computation from different lengths. RP

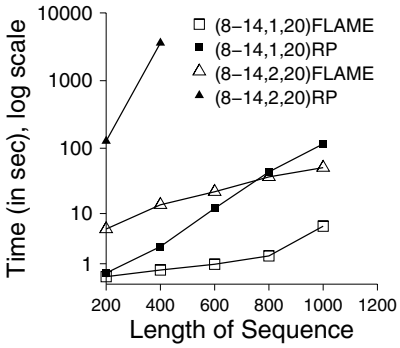

Fig. 6. RP vs FLAME for varying database sizes

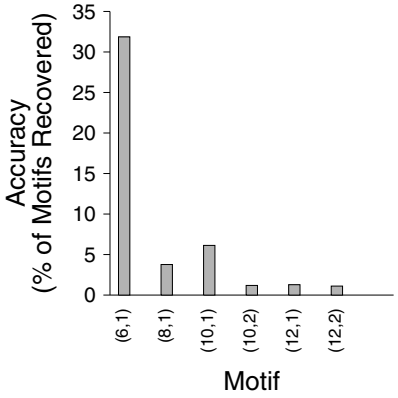

Fig. 7. Weeder: Accuracy on real DNA datasets

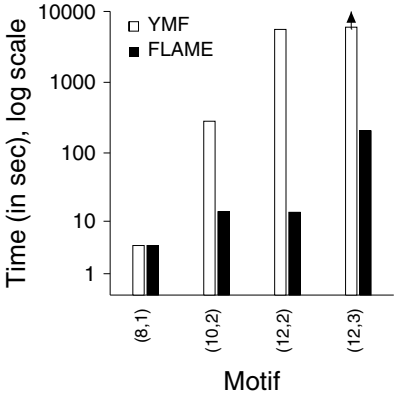

Fig. 8. YMF vs FLAME on synthetic datasets

is run once for each value of L since it does not lend itself to combining computation.

For the task of finding motifs with L varying from 8 to 14, and d=1 (denoted as $(8-14, 1, 20)$ in Figure 6), the RP algorithm works well for small database sizes. However, as the database size increases, we see that its performance begins to deteriorate rapidly. A detailed explanation for FLAME's performance advantage is presented in [11]. For the $(8-14, 2, 20)$ task, RP takes too long to complete for sequence lengths beyond 400, and we do not report these times in Figure 6.

When many buckets need to be explored to find the real (L, d, k) pattern, RP ends up taking much longer. FLAME, one the other hand, is relatively less sensitive to increases in the database size (Figure 6). A larger database will lead to a model being pruned deeper in the model tree, but FLAME still manages to avoid a lot of redundant computation by virtue of using the suffix tree to efficiently prune the model space.

B. Comparison with Weeder and YMF

Many algorithms have been proposed in the field of computational biology for finding motifs. Most of these algorithms deal with (L, d, k) type motifs [12], [14], [15], [41]. A recent study [32] compared several algorithms, and determined that Weeder [13] and YMF [12] performed among the best. Weeder scored highest on many performance metrics, and YMF did nearly as well. In this section, we compare FLAME with these two algorithms.

1) Comparison with Weeder: Weeder is a very fast heuristic algorithm that was specifically designed to find motifs in DNA datasets. The algorithm is limited to the (L, d, k) model and does not work with the more powerful (L, M, s, k) model. Weeder is extremely fast because it assumes that the mismatches are distributed uniformly across the length of the motif. As a result of this assumption, Weeder can aggressively prune the search space very quickly, but it is not guaranteed to be accurate.

We perform a simple experiment to determine the accuracy of Weeder. We use the Washington dataset [32] that is based on the real motifs found in the TRANSFAC [49] database. We run both algorithms on this dataset using a variety of models. We present the number of motifs found by Weeder as a percentage of the total number of motifs present in the dataset in Figure 7. Since FLAME is an accurate algorithm, it always finds all the motifs in the dataset, and we do not show its accuracy (100%) in the graph. As one can readily observe, Weeder misses a large number of motifs. In fact, for the case of $(12, 2)$ motifs, Weeder finds less than 5% of the total number found by FLAME. The one point in favor of Weeder is speed. It takes only one second to find a (10,2,20) motif while FLAME takes close to 40 seconds. Weeder pays the price for this speed with a very low accuracy.

The task of predicting regulatory elements is a two step process. First a pattern finding tool such as Weeder or FLAME can be used to find all the patterns that frequently occur in the dataset being considered. The second step is to examine these patterns and score them on various factors such as strength of the motif, biological importance, statistical significance, etc. The second step requires domain knowledge to distinguish between patterns that are real regulatory sequences versus random matches to the background "junk DNA". Biologists employ many heuristics for the second phase. The first phase is orthogonal, and any pattern finding tool can be used and paired with a different scoring/ranking procedure.

Figure 7 shows that while FLAME finds all the candidate motifs, Weeder might miss a significant fraction. Finding more results in the first phase of the computation is certainly beneficial since we will be better informed going into the second phase of ranking the patterns found, and therefore stand a better chance of identifying the best motifs.

To demonstrate the effectiveness of FLAME in finding real biological motifs that are missed by Weeder, we performed the following experiment: We list all the candidate motifs found by FLAME in the Washington dataset and rank them using the *same* scoring function as Weeder's. We observed that FLAME was able to correctly identify several motifs that Weeder missed. For instance, FLAME reports TCGTAACG on human dataset *hm08r*, CGACGTATGC on *hm11g*, and CGTACGAT on *hm16r*. Weeder misses these motifs because of its aggressive pruning strategy. Since Weeder has a very

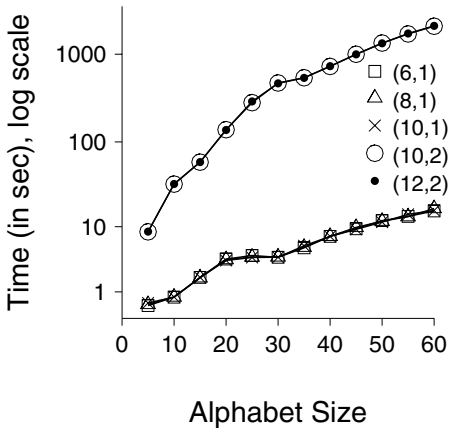

Fig. 9. Performance as the alphabet size varies

low accuracy, we do not consider it for experiments in the remainder of this section.

2) Comparison with YMF: Another algorithm that performed well in the comparison in [32] is YMF (Yeast Motif Finder). YMF is a simple and accurate algorithm that finds *all* patterns that appear more frequently than expected in a set of DNA sequences. Like Weeder, YMF too cannot be used for (L, M, s, k) models. It simply has a counter corresponding to each possible motif in the model space. It scans the database once using a sliding window and augments the count for each motif that matches the sliding window. One can easily see that YMF will scale linearly with the size of the database, but will scale very poorly with the size of the model space since it keeps a counter for each possible model. YMF becomes impractical for longer, complex motifs.

We demonstrate this behavior using a synthetic dataset containing 20 sequences, each 600 symbols long. We implant different (L,d,20) motifs in the sequence. We run YMF and FLAME on a variety of (L, d, k) motifs. The results are averaged over 50 datasets. The results of this experiment are presented in Figure 8. For the (8,1) motif, both YMF and FLAME finish very quickly. However, we can easily see that YMF does not scale well as the motif complexity increases. For the $(12, 3)$ motif, YMF did not finish in a reasonable amount of time, and we had to terminate the program after two hours. FLAME, on the other hand, completes in less than two minutes. We conducted similar experiments by varying the sequence length from 200 to 1000. FLAME continues to be faster than YMF for these settings, and we omit presenting the results in the interest of space.

We devote the rest of the evaluation section to study the performance characteristics of FLAME as different parameters in the problem setting are varied.

C. Performance Characteristics of FLAME

1) Alphabet Size: Our next experiment studies the effect of alphabet size on execution time. For this task, we again use the synthetic dataset generator. We vary the alphabet size from 5 to 50, and at each point evaluate the execution time for various implanted patterns. Each dataset consisted of 20 sequences, each of length 600, totaling 12,000 symbols. The execution time for various implanted motifs is summarized in Figure 9.

As can be seen in the figure, execution times for simpler motifs such as $(6, 1), (8, 1)$, and $(10, 1)$ grow slowly with alphabet size. Complex motifs, such as $(8, 2)$ and $(10, 2)$, which inherently require the algorithm to search a larger space, grow faster with alphabet size. Nevertheless, the mining task is often completed within a few hours even for very large alphabets. Several real world applications such as DNA sequence mining, and protein sequence mining typically require an alphabet of size less than 25, and can be mined very quickly with FLAME.

2) Mining Time Series Data: We now study the performance of FLAME for different parameters of the (L, M, s, k) motif model. (Since existing algorithms do not support the (L, M, s, k) model, we do not compare FLAME with any other algorithms for the rest of this section.)

In this experiment, we use the (L, M, s, k) model to mine the IBM dataset. We use a 20 bucket histogram that partitions the dataset into roughly equal sized buckets. We then assigned a symbol to each bucket, and encoded the numerical series into a symbolic sequence. The dataset totaled about 8,400 symbols. The distance penalty matrix is a squared error matrix using the numerical values corresponding to each symbol. That is, $M(A, B) = |v(A) - v(B)|^2$, where $v(A)$ is the numerical value corresponding to the symbol A (the midpoint of the bucket in the histogram).

We present the time taken by FLAME to find several (L, M, s, k) motifs. First we set the support to be 21 (equal to the number of sequences in the dataset). We run FLAME for L = 5, 8, and 11, while varying the distance threshold s. The results of this experiment are shown in Figure 10. We observe from the figure that as the threshold is increased the time taken to execute the search increases. This is because at higher thresholds, the pattern is more relaxed, and the space of potential models that needs to be searched is larger. FLAME is able to find models of length 11 within 16 seconds. We then repeated the experiment for higher support values of 60 and 120. Increasing the support causes more aggressive pruning of the search space, and hence, a lower execution time was observed.

3) Scaling to Large Datasets: Finally, we demonstrate the scalability of the FLAME algorithm for mining motifs on very large datasets. Motif mining is a difficult task, and existing algorithms focus on relatively small datasets (of the order of 10,000 symbols). We show that using FLAME, it is possible to scale to much larger database sizes. We generate synthetic datasets, and embed a motif of length chosen randomly between 8 and 14 in 10% of the sequences. The datasets contain sequences of length 1000, and the number of sequences is increased gradually to generate database of increasing sizes. The total database size is varied from 20,000 symbols to 1 million symbols. We run FLAME on these datasets to find $(8 - 14, 1)$ and $(8 - 14, 2)$ models with 10% support. The

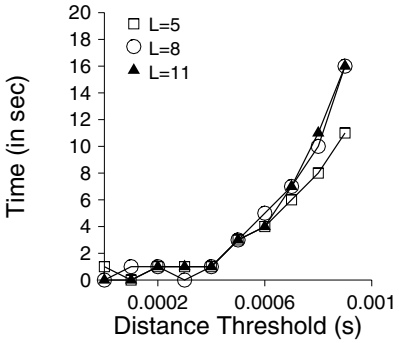

Fig. 10. FLAME: Distance threshold vs time taken on IBM stock price data

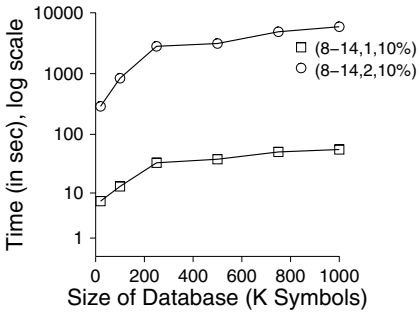

Fig. 11. Scalability of FLAME with increasing database size

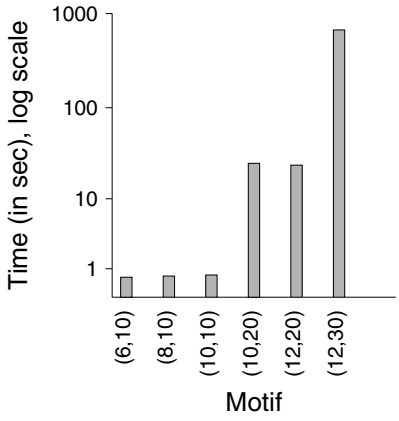

Fig. 12. FLAME: (L,M,s,k) motifs on the Snake dataset.

results for this experiment are shown in Figure 11.

The execution time increases relatively slowly (Figure 11) as we increase the database size. In the case of $(8-14, 1, 10\%)$ motifs, the time increases from 7 seconds to 55 seconds over the entire range. In the case of $(8-14, 2, 10\%)$ motifs, the time increases from 290 seconds to 5900 seconds. As one would expect, the time to mine more complex motifs grows a little faster. The reason for this behavior is that the number of models considered before pruning begins in the model tree is exponential in d. The running time is proportional to the total number of candidate motifs considered, and this number also grows exponentially with d. For example, when the database size is 20,000 symbols, in the $(8-14, 1, 10\%)$ case, the number of models considered before pruning is 13,240 and the number of candidate motifs is 1,323. In the $(8-14, 2, 10\%)$ case, the number of models explored is 200,620 and the number of candidate motifs is 20,061. However, even patterns of length 14 in a database this large can be mined in a few hours. To our knowledge, none of the existing algorithms can accurately scale to such large database sizes.

D. Protein Motif Mining

Mining for motifs in protein sequences is an application where the (L, M, s, k) model offers a significant advantage over using less powerful models such as (L, d, k) by allowing the use of similarity matrices like PAM30 [1] and BLOSUM [2]. These matrices are popular in life sciences applications and are crucial to capturing the notion of similarity in this domain.

In this experiment, we look for (L, M, s, k) motifs using PAM30 as the distance matrix and the Snake dataset. We fix the support to be 175 (roughly half the number of sequences) to find patterns that are common to snake venom proteins. (Protein sequence mining typically uses high thresholds [38].) We varied (L, s) as $(6, 10)$, $(8, 10)$, $(10, 10)$, $(10, 20)$, $(12, 20)$, and $(12, 30)$. The results are shown in Figure 12. As we can see from this figure, the computation time increases with an increase in the distance threshold. A higher distance threshold indicates a more relaxed pattern – which in turn means that FLAME has to proceed deeper down the model tree before it can start eliminating models. As can be observed from Figure 12, even the longest motifs are found reasonably quickly.

We note that this experiment also highlights the difference between FLAME and previous sequence mining methods like cSPADE [9], CloSpan [7], and BIDE [6]. As mentioned in Section I, FLAME address a different but related problem to the problem of mining of frequent itemsets. Specifically, none of these other methods can support a general match matrix like PAM30 above.

E. Summary

In this section, we evaluated FLAME on a number of real and synthetic datasets. The results demonstrate that FLAME is faster, and scales better than other algorithms that have been used for time series mining, such as Random Projections. In addition, comparison of FLAME with two of the best algorithms used in computational biology [32], namely Weeder and YMF, shows that:

1) Weeder is fast, but misses a significant number of motifs (more than 90% for complex motifs). On the other hand, FLAME is guaranteed to find *all* motifs in the dataset.
2) YMF, like FLAME, is 100% accurate, but is very slow. Compared to YMF, FLAME is faster by more than an order of magnitude.

We also conducted experiments to test various characteristics of FLAME. These experiments reveal that FLAME performs well in a variety of mining situations, and scales to datasets much larger (1 million symbols) than has been attempted before.

VII. Conclusions

In this paper, we presented a powerful new model: (L, M, s, k) for motif mining in sequence databases. The (L, M, s, k) model subsumes several existing models and provides additional flexibility that makes it applicable in a wider variety of data mining applications. We also presented FLAME, a flexible and accurate algorithm that can find

(L, M, s, k) motifs. Through a series of experiments on real and synthetic datasets, we demonstrate that FLAME is a versatile algorithm that can be used in several real motif mining tasks. We also show that FLAME outperforms existing time series mining algorithms (Random Projections) by more than an order of magnitude. FLAME is also superior to motif finding algorithms used in computational biology (more accurate than Weeder, significantly faster than YMF). We also presented experiments which show that FLAME can scale to handle motif mining tasks that are much larger than attempted before.

ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation under grant DBI-0926269.

We thank J. Buhler, G. Mauri, P. Mereghetti, G. Pavesi, G. Pesole, S. Sinha, M. Tompa, and M. J. Zaki for providing the source code of the tools that were used in the comparison with FLAME. We also thank M. Tompa and his colleagues for making available a framework to evaluate motif finding tools on real biological datasets.

REFERENCES

[1] M. O. Dayhoff, R. M. Schwartz, and B. Orcutt, "A Model for Evolutionary Changes in Proteins," *Atlas of Protein Sequence and Structure*, vol. 5, pp. 345–352, 1978.

[2] S. Henikoff and J. Henikoff, "Amino Acid Substitution Matrices from Protein Blocks," *National Academy of Sciences, USA*, vol. 89, no. 22, pp. 10 915–9, 1992.

[3] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," in *VLDB*, 1994, pp. 487–499.

[4] ——, "Mining Sequential Patterns," in *ICDE*, 1995, pp. 3–14.

[5] M. J. Zaki, "SPADE: An Efficient Algorithm for Mining Frequent Sequences," *Machine Learning*, vol. 42, no. 1/2, pp. 31–60, 2001.

[6] J. Wang and J. Han, "BIDE: Efficient Mining of Frequent Closed Sequences," in *ICDE*, 2004, pp. 79–90.

[7] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Datasets," in *SDM*, 2003.

[8] J. Yang, W. Wang, P. S. Yu, and J. Han, "Mining Long Sequential Patterns in a Noisy Environment," in *SIGMOD*, 2002, pp. 406–417.

[9] M. J. Zaki, "Sequence Mining in Categorical Domains: Incorporating Constrains," in *CIKM*, 2000, pp. 442–429.

[10] J. Pei, J. Han, and W. Wang, "Mining Sequential Patterns With Constraints in Large Databases," in *CIKM*, 2002, pp. 18–25.

[11] A. Floratou, S. Tata, and J. M. Patel, "Finding Hidden Patterns in Sequences, Tech. Rep. http://www.pages.cs.wisc.edu/∼floratou, 2009.

[12] S. Sinha and M. Tompa, "YMF: A Program for Discovery of Novel Transcription Factor Binding Sites by Statistical Overrepresentation," *Nucleic Acids Research*, vol. 31, no. 13, 2003.

[13] G. Pavesi, P. Mereghetti, G. Mauri, and G. Pesole, "Weeder Web: Discovery of Transcription Factor Binding Sites in a Set of Sequences From Co-Regulated Genes," *Nucleic Acids Research*, vol. 32(Web Server issue), pp. W199–W203, 2004.

[14] E. Eskin and P. A. Pevzner, "Finding Composite Regulatory Patterns in DNA Sequences," in *ISMB*, 2002, pp. S354–63.

[15] J. Buhler and M. Tompa, "Finding Motifs Using Random Projections," *Journal Computational Biology*, vol. 9, no. 2, pp. 225–242, 2002.

[16] G. Das, K.-I. Lin, H. Mannila, G. Renganathan, and P. Smyth, "Rule Discovery From Time Series," in *KDD*, 1998, pp. 16–22.

[17] S. Hoppner, "Discovery of Temporal Patterns – Learning Rules about the Qualitative Behaviour of Time Series," in *5th European Conference on Principles and Practice of Knowledge Discovery in Databases*, 2001, pp. 192–203.

[18] P. Patel, E. Keogh, J. Lin, and S. Lonardi, "Mining Motifs in Massive Time Series Databases," in *ICDM*, 2002, pp. 370–377.

[19] H. Wu, B. Salzberg, G. C. Sharp, S. B. Jiang, H. Shirato, and D. Kaeli, "Subsequence Matching on Structured Time Series Data," in *SIGMOD*, 2005, pp. 682–693.

[20] B. Y.-C. Chiu, E. J. Keogh, and S. Lonardi, "Probabilistic Discovery of Time Series Motifs," in *KDD*, 2003, pp. 493–498.

[21] W. Wang and J. Yang, *Mining Sequential Patterns from Large Data Sets*. Springer-Verlag, 2005, vol. 28.

[22] M. Das and H. K. Dai, "A Survey of DNA Motif Finding Algorithms," *BMC Bioinformatics*, vol. 8, 2007.

[23] G. K. Sandve and F. Drabløs, "A Survey of Motif Discovery Methods in an Integrated Framework," *Biology Direct*, vol. 1, 2006.

[24] Y. Zhang and M. J. Zaki, "ExMOTIF: Efficient Structured Motif Extraction," *Algorithms for Molecular Biology*, vol. 1, no. 21, November 2006.

[25] A. M. Carvalho *et al.*, "An Efficient Algorithm for the Identification of Structured Motifs in DNA Promoter Sequences," *IEEE/ACM Transactions on computational biology and bioinformatics*, vol. 3, 2006.

[26] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "PrefixSpan: Mining Sequential Patterns by Prefix-Projected Growth," in *ICDE*, 2001, pp. 215–224.

[27] A. Brazma, I. Jonassen, I. Eidhammer, and D. Gilbert, "Approaches to the Automatic Discovery of Patterns in Biosequences," *Journal of Computational Biology*, vol. 5, pp. 279–305, 1998.

[28] L. Marsan and M.-F. Sagot, "Algorithms for Extracting Structured Motifs Using a Suffix Tree with Application to Promoter and Regulatory Site Consensus Identification," *Journal of Computational Biology*, vol. 7, no. 3/4, pp. 345–360, 2000.

[29] F. Zhu, X. Yan, J. Han, and P. S. Yu, "Efficient discovery of frequent approximate sequential patterns," in *ICDM*, 2007.

[30] T. L. Bailey and C. Elkan, "Unsupervised Learning of Multiple Motifs in Biopolymers using EM," *Machine Learning*, vol. 21, no. 1-2, pp. 51–80, 1995.

[31] W. Thompson, E. C. Rouchka, and C. E. Lawrence, "Gibbs Recursive Sampler: Finding Transcription Factor Binding Sites," *Nucleic Acids Research*, vol. 31, no. 13, pp. 3580–3585, 2003.

[32] M. Tompa *et al.*, "Assessing Computational Tools for the Discovery of Transcription Factor Binding Sites," *Nature Biotechnology*, vol. 23, pp. 137–144, 2005.

[33] A. W.-C. Fu, E. J. Keogh, L. Y. H. Lau, and C. A. Ratanamahatana, "Scaling and Time Warping in Time Series Querying," in *VLDB*, 2005, pp. 649–660.

[34] M. Vlachos, G. Kollios, and D. Gunopulos, "Discovering Similar Multidimensional Trajectories," in *ICDE*, 2002, pp. 673–684.

[35] L. Chen, M. Tamer Ozsu, and V. Oria, "Robust and Fast Similarity Search for Moving Object Trajectories," in *SIGMOD*, 2005, pp. 491–502.

[36] Y. Zhu and D. Shasha, "Warping Indexes with Envelope Transforms for Query by Humming," in *SIGMOD*, 2003, pp. 181–192.

[37] A. Udechukwu, K. Barker, and R. Alhajj, "Discovering all frequent trends in time series," in *Proc. of Winter Int. Sym. on Information and Comm. Tech.*, vol. 58, 2004, pp. 1–6.

[38] I. Jonassen, J. F. Collins, and D. G. Higgins, "Finding Flexible Patterns in Unaligned Protein Sequences," *Protein Science*, vol. 4, no. 8, pp. 1587–1595, 1995.

[39] "Data Sets from Analysis of Financial Time Series," http://www.gsb.uchicago.edu/fac/ruey.tsay/teaching/fts/.

[40] R. S. Tsay, *Analysis of Financial Time Series*, 1st ed. Wiley-Interscience, October 2001.

[41] P. A. Pevzner and S.-H. Sze, "Combinatorial Approaches to Finding Subtle Signals in DNA Sequences," in *ISMB*, 2000, pp. 269–278.

[42] "YMF Source Code," http://bio.cs.washington.edu/software.html.

[43] "Weeder Source Code," http://www.pesolelab.it/Tool/ind.php.

[44] "Random Projections Source Code," http://www.cse.wustl.edu/ jbuhler/pgt/.

[45] "cSPADE Source Code," http://www.cs.rpi.edu/ zaki/software/.

[46] S. Tata, R. A. Hankins, and J. M. Patel, "Practical Suffix Tree Construction," in *VLDB*, 2004, pp. 36–47.

[47] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," in *VLDB*, 1999, pp. 518–529.

[48] T. L. Bailey and C. Elkan, "Fitting a Mixture Model by Expectation Maximization to Discover Motifs in Biopolymers," in *ISMB*, 1994, pp. 28–36.

[49] "TRANSFAC," http://www.gene-regulation.com/pub/databases.html.