

An efficient technique for mining approximately frequent substring patterns

Xiaonan Ji and James Bailey

NICTA Victoria Laboratory

Department of Computer Science and Software Engineering

University of Melbourne, Australia

{xji, jbailey}@csse.unimelb.edu.au

Abstract

Sequential patterns are used to discover knowledge in a wide range of applications. However, in many scenarios pattern quality can be low, due to short lengths or low supports. Furthermore, for dense datasets such as proteins, most of the sequential pattern mining algorithms return a tremendously large number of patterns, which are difficult to process and analyze. However, by relaxing the definition of frequency and allowing some mismatches, it is possible to discover higher quality patterns. We call these patterns Frequent Approximate Substrings or FAS-patterns and we introduce an algorithm called FAS-Miner, to handle the mining task efficiently. The experiments on real-world protein and DNA datasets show that FAS-Miner can discover patterns of much longer lengths and higher supports than standard sequential mining approaches.

1. Introduction

Much research has focused on mining of patterns from collections of sequences. Two of the most popular types of patterns in sequence data mining are frequent subsequence patterns [1] and frequent substring patterns [5]. The first corresponds to subsequences that appear frequently and may contain gaps between adjacent items. The second is a special case of the first and corresponds to subsequences that appear frequently and *do not* contain gaps between adjacent items.

The quality of sequential patterns can be evaluated in two principal ways: *pattern length* and *pattern support*. Allowing a shorter pattern length can be undesirable, particularly when the sequences are long, since the meaning is less specific. Patterns with low supports also are less desirable, since they can be trivial and may not describe general phenomena. Often, and particularly in a biological context, patterns should have long lengths and high supports. However, a main drawback of standard sequential pattern mining

approaches is that they tend to predominantly discover massive amounts of “low quality” patterns, i.e. patterns having either short lengths or low supports. Consider the following example.

Example 1 *In a group of 1048 L1 capsid protein sequences, with a minimum support of 10%, the average length of substring patterns found was 8 and the maximum length was 33. Increasing the minimum support to 60% resulted substring patterns with an average length of 1.4 and a maximum length as 2. By allowing a maximum gap of 2, we still could only discover subsequence patterns with a maximum length of 7 at a 60% minimum support.*

From our experience, this kind of scenario can be very common. Users try many different combinations of constraint thresholds, but it is very difficult or impossible to find thresholds which yield long patterns with high supports. Consequently, users often fall back on using lower support thresholds to find patterns. This in turn increases the mining time and the number of patterns returned.

In this paper, we evaluate a new kind of sequential pattern, which we call the **Frequent Approximate Substring Pattern (FAS-pattern)**. This is a substring which appears frequently, but some mismatches are allowed during support counting. The benefit of such patterns is that they often possess much higher (approximate) supports and longer lengths. An example is given as follows.

Example 2 *Consider the dataset from Example 1. When setting minimum support threshold as 10% and allowing up to 10 mismatches between a pattern and its appearances, we discovered 25310 FAS-patterns whose smallest length was 50 and largest was 80. When setting minimum support threshold as 60% and maximum number of mismatches as 5, we could still discover 1472 FAS-patterns, whose average length was 11.5 and largest length was 14.*

This example tells that by allowing some mismatches, it is possible to discover valuable sequential patterns whose

lengths are much longer and (approximate) supports are much higher than normal frequent subsequence or substring patterns. As a further benefit, due to their prevalence at high supports, users can employ higher support thresholds when mining FAS-patterns. This filters out the trivial ones and reduces overall pattern volume.

Challenges. Several mining challenges arise. The first is that although projection and prefix extension techniques [9] can be used, considerably more space is needed to store many projected databases. Indexing data structures can improve memory and time usage, but special purpose algorithms are required. The second challenge arises with respect to checking mismatches. Every candidate string needs to be compared against all others to find its approximate support and fast incremental methods are vital. Thirdly, FAS-patterns do not use exact minimum support threshold and traditional support pruning is much weaker, necessitating the development of alternative pruning strategies.

Contributions. We devise an algorithm called *FAS-Miner* (*Frequent Approximate Substrings Miner*) to efficiently mine the complete FAS-pattern set. We employ a novel approach, which combines optimized suffix arrays with several pruning techniques. Experimental analysis shows that *FAS-miner* is able to efficiently mine high quality FAS-patterns from some very dense real-world datasets.

2. Preliminaries

Let Σ denote the alphabet set containing distinct items. A sequence is an ordered list of (possibly duplicated) items. For example, DNA sequences are sequences over $\Sigma = \{A, C, G, T\}$. The i -th item of a sequence S is denoted as $S[i]$.

It is necessary to distinguish subsequences and substrings. A sequence $S_1 = a_1a_2a_3\dots a_n$ is called a *subsequence* of $S_2 = b_1b_2b_3\dots b_m$ and denoted as $S_1 \subseteq_{seq} S_2$ if $n \leq m$ and there exist integers $1 \leq i_1 < i_2 < \dots < i_n \leq m$ such that $a_j = b_{i_j}$ for $1 \leq j \leq n$. For example, $ab \subseteq_{seq} acb$ but $ba \not\subseteq_{seq} acb$. If there exists a set of positions $\{i_1i_2\dots i_n\}$ such that $i_{k+1} = i_k + 1$ for $1 \leq k < n$, then S_1 is a *substring* of S_2 and denoted as $S_1 \subseteq_{str} S_2$. For example, $ab \subseteq_{str} abc$ but $ac \not\subseteq_{str} abc$. There can be more than one such set of positions, which means that S_1 can appear more than once in S_2 . Each position set is called an *appearance* of S_1 in S_2 . For example, ab appears twice as a substring in $abcab$ at positions $\{1, 2\}$ and $\{4, 5\}$. An appearance of a substring in S can also be denoted by its starting and ending positions, $S[i_1..i_n]$. As a special case, $S[1..m]$ is called the m -th prefix of S .

Given a sequence database SDB and a minimum support threshold α , a sequence P is a frequent subsequence pattern in SDB if $\frac{|\{S \in SDB | P \subseteq_{seq} S\}|}{|SDB|} \geq \alpha$ holds and is a frequent

substring pattern if $\frac{|\{S \in SDB | P \subseteq_{str} S\}|}{|SDB|} \geq \alpha$ holds.

The distance or dissimilarity between two items in Σ is defined in *distance matrix*.

Definition 1 (Distance Matrix) The distance matrix M is an $|\Sigma| \times |\Sigma|$ symmetric matrix where $M[i][j] \geq 0$ defines the distance (or dissimilarity) between the i -th and j -th items of Σ .

We will assume that higher values for distance between a pair of items mean higher dissimilarity. Distance matrices are commonly used in bioinformatics, such as BLOSUM series matrices. The distance between two sequences is measured by (weighted) hamming distance.

Definition 2 (Weighted Hamming Distance) The weighted hamming distance of two length- n sequences S_1 and S_2 , denoted as $dist(S_1, S_2)$, is the pairwise summation of distances between items at the same positions.

$$dist(S_1, S_2) = \sum_{i=1}^n M[S_1[i]][S_2[i]]$$

The larger the hamming distance is, the more dissimilar the two sequences are.

3. FAS-Patterns

The relevant definitions of FAS-Patterns are given as follows.

Definition 3 (Approximate Containment \subseteq_{ϵ}) A substring P is approximately contained in a sequence S according to the maximum distance threshold ϵ (denoted as $P \subseteq_{\epsilon} S$) if $\exists S' \subseteq_{str} S$ s.t. $|S'| = |P| \wedge dist(P, S') \leq \epsilon$.

Definition 4 (Approximate frequency and support) Given a distance matrix M and a maximum distance threshold ϵ , a substring P 's approximate frequency is the number of sequences from SDB it is approximately contained in, that is, $freq_{\epsilon}(P, SDB) = |\{S \in SDB | P \subseteq_{\epsilon} S\}|$. The ratio of the approximate frequency and the total number of sequences in SDB is called the approximate support, that is, $supp_{\epsilon}(P, SDB) = \frac{freq_{\epsilon}(P, SDB)}{|SDB|}$.

Definition 5 (Frequent Approximate Substring Patterns) Given a user-defined distance matrix M , maximum distance ϵ , minimum approximate support α and minimum length l , a sequence P is called a FAS-pattern if and only if it satisfies the following constraints:

- **Non-virtual constraint:** $freq_0(P, SDB) \geq 1$
- **Minimum frequency constraint:** $supp_{\epsilon}(P, SDB) \geq \alpha$

Table 1. An example SDB.

ID	SEQUENCE
1	<i>aabac</i>
2	<i>bbacc</i>
3	<i>cacbb</i>
4	<i>bcbca</i>
5	<i>acbab</i>

<i>sa</i>	<i>lcp</i>	<i>iid</i>
<i>aabac#</i>	0	1
<i>abac#</i>	1	1
<i>acbab#</i>	1	5
<i>acbb#</i>	3	3
<i>acc#</i>	2	2
<i>bab#</i>	0	5
<i>bac#</i>	2	1

Cont.		
<i>bacc#</i>	3	2
<i>bbacc#</i>	1	2
<i>bca#</i>	1	4
<i>bcbca#</i>	2	4
<i>cacbb#</i>	0	3
<i>cbab#</i>	1	5
<i>cbb#</i>	2	3
<i>cbca#</i>	2	4

Figure 1. *sa*, *lcp* and *iid* for SDB in Table 1.

- **Minimum length constraint:** $|P| \geq l$

We use a minimum length constraint to prohibit trivial patterns. The definition concentrates on non-virtual patterns, i.e. patterns appear at least once in *SDB*. Given M , *SDB*, α, ϵ and l , FAS-pattern mining problem is to discover all the FAS-patterns from *SDB*.

4. The Algorithm of FAS-Miner.

FAS-Miner uses a suffix array to store and organize the search space; a prefix extension approach to generate candidates and an incremental way of checking distance and counting frequencies. It also uses techniques for space pruning and other optimizations.

The suffix array. *FAS-Miner* uses a suffix array (abbreviated *sa*) to store all suffixes of sequences from *SDB*, sorted in lexicographic order. Only suffixes whose lengths are greater than minimum length l are stored. We use suffix array rather than suffix tree, since it consumes less memory and it is easier to implement pruning techniques.

Example 3 For *SDB* given in Table 1, the suffix array *sa*, when $l = 3$, is given in the first column¹ in the table of Figure 1. We use a character $\# \notin \Sigma$, to indicate the end of each suffix. *sa* does not contain suffixes such as *ac#*, because their lengths are smaller than l .

The suffixes are not stored literally, only the starting positions of the suffixes are stored in *sa*. Thus, the size of *sa* grows linearly with total length of sequences in *SDB*.

¹The meanings of other columns will be introduced later.

Enumerating candidates by prefix extension. Starting from the first suffix of *sa*, prefixes are incrementally extended. Each prefix is a *candidate* and it is compared with prefixes of the same length of all the other suffixes in the array, to count the approximate support. Candidate generation is performed for all suffixes in the array. The distance information is kept in another array called the *distance array* (abbreviated *da*) for each candidate. *da* has the same dimension as *sa* and $da[i]$ holds the distance between the current candidate and the same-length prefix of the i -th suffix.

Example 4 Consider *sa* in Figure 1, candidates are generated in the order as “*a*” → “*aa*” → ... → “*aabac*” → ... → “*aba*” → ... *da* for the first candidate “*a*”, which is the first prefix of the first suffix, is $\{0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1\}$. The first five 0s mean that this candidate is identical with the length-1 prefixes of the first five suffixes. The middle six 1s mean that this candidate is one item different from the length-1 prefixes of the middle six suffixes (which are “*b*”s). Similarly, *da* of candidate “*aab*” will be $\{0, 2, 1, 1, 2, 1, 2, 2, 3, 3, 2, 2, 3, 2, 3\}$.

Each candidate C is extended by $(|C| + 1)$ -th item within the suffix (if there is any). As the candidate is extended, its *da* needs to be updated. If $da[i] > \epsilon$ holds, it is unnecessary to compare candidates from the current suffix with any prefix of the i -th suffix anymore. When updating *da* for a newly-extended candidate, the distances between the candidate and all other prefixes are not computed from scratch. Suppose the current candidate is the n -th prefix of suffix S , we only need to increase each $da[i]$ of the $(n - 1)$ -th prefix of S by $M[S[n]][sa[i][n]]$, to get each $da[i]$ of the n -th prefix.

Frequency counting. After *da* is updated for the newly-extended candidate, the candidate’s approximate frequency can be counted by examining how many elements in *da* have a value no larger than ϵ . For the i -th element, if $da[i] \leq \epsilon$, then the distance between the candidate and the same length prefix of the i -th suffix in *sa* is smaller than or equal to ϵ and vice versa. In order to compute approximate frequencies, suffixes needs to be mapped to sequences of *SDB* containing them. Another array called *iid* (instance ID array) with the same length as *sa*, is used to record IDs of sequences containing each suffix. The *iid* values of *sa* built from *SDB* of Table 1 are given in the third column in Figure 1. Frequency counting of candidate prefixes from the same suffix can be done incrementally.

Pruning the candidate space within a suffix. The n -th prefix of suffix S is only examined when $(n - 1)$ -th prefix is approximately frequent. If this prefix is frequent and is longer than l , it is output as a FAS-pattern. A prefix candidate C is not extended any more if and only if it meets one of the *two stopping conditions*: (1) C is not approximately frequent (monotonicity) or (2) C is equal to the whole suffix, that is,

Table 2. Experimental datasets.

Data set	$ \Sigma $	# Seq.	max. seq. len.	avg. seq. len.
snake	20	174	121	67
PF00500	22	1048	523	209
PF03880	22	330	95	86
DNA	4	1000	301	258

$n = |S|$.

Reusing da for candidates within suffixes. Recall that da has the same dimensionality as sa . The updating of da can nevertheless be time-consuming. So, for a candidate being extended from its prefix, its da is updated from the da of its prefix, rather than recalculated from scratch. This is only possible when candidates are extended from their prefixes within the same suffix. When moved to generate candidates for a new suffix, a basic strategy would need to start extending the prefix again from empty and also calculate da from scratch. We next discuss how to optimize the reuse of da when moving to a different suffix.

Reusing da for candidates crossing suffixes. The *longest common prefix array* (abbreviated as *lcp*) is used for this technique. It has the same dimension as sa and $lcp[i]$ contains the length of the longest common prefix of $sa[i-1]$ and $sa[i]$. For sa in Figure 1, lcp is given in the second column of the table. There are two cases to consider. In the ideal case, if two adjacent suffixes $sa[i-1]$ and $sa[i]$ share some common prefix, then da computed for prefix in $sa[i-1]$ can be preserved and reused when moved to candidate generation in $sa[i]$. The second, more complicated case, is that sometimes even though two adjacent suffixes might share a common prefix, we may not be able to straightforwardly preserve a necessary da during candidate generation within the first suffix. This is because the smallest suffix $sa[x]$ sharing the longest common prefix with current suffix $sa[i]$ may not be $sa[i-1]$. A longer eyesight is used to identify the smallest suffix sharing longest common prefix with each $sa[i]$ so a best da can be preserved for each suffix.

5. Experimental Results

Datasets. 4 real-world challenging biomolecular datasets are used to test the performance of *FAS-Miner*. Details of the datasets are given in Table 2²

Algorithms. *FAS-Miner* was compared against a frequent substring mining approach, a frequent subsequence mining approach (PrefixSpan [9]), a frequent subsequence with gap constraint mining approach and a closed frequent subsequence mining approach (BIDE [12]). For (exact) substring

²The middle two datasets can be obtained from <http://www.sanger.ac.uk/Software/Pfam/> and the last one obtained from NCBI using the query term “200:300[sequence length] AND 2002/12:2003/02[publication date]” in the category of “Nucleotide”.

mining, we implemented a mining algorithm (referred to as *Substr*) using a lazy suffix trie construction, rather like [4]. We also extended this algorithm to mine subsequences according to maximum gap constraint g (referred to as *Subseq*), using the gap checking approach from [7].

We used the complement of the identity matrix, as distance matrix. In our discussions, support value (α) refers both to the approximate minimum support threshold for *FAS-Miner* and the exact minimum support threshold for other algorithms. In all diagrams, ϵ is denoted as e , l is the minimum length threshold and g is the maximum gap threshold. We did not apply minimum length constraint in any algorithm other than *FAS-Miner*. Results are only given for *PrefixSpan* and *BIDE* on snake dataset, since they could not finish in reasonable time on the other datasets. Missing data points indicate that either a timeout (> 3 hours) occurred or no pattern was found.

Pattern quality analysis. We evaluated the patterns according to *supports* and *lengths*. The maximum pattern length distributions for varying support thresholds are given in (a), (b), (c), (d) of Figure 2. It is obvious that FAS-patterns with high supports have consistently longer lengths than frequent substring or subsequence patterns³. The larger ϵ was set, the longer FAS-patterns could be found. Among all types of patterns, frequent substrings have the shortest maximum length. Although not shown in these diagrams, FAS-patterns also achieved a much longer *average* length than the other types of patterns. From these diagrams we can also see that by using a maximum gap threshold g , patterns longer than pure substrings could be found, but unfortunately it is expensive to mine such patterns with large gaps.

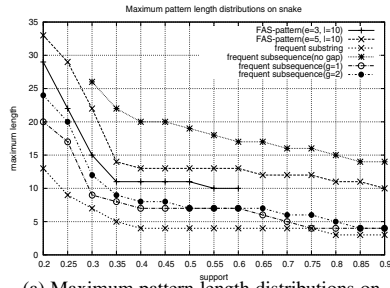
Pattern distributions over different lengths for two datasets are shown in (e) and (f) of Figure 2. We can see that substring and subsequence patterns mainly have small lengths, from 1 to 6. In contrast, FAS-patterns were found to have much longer lengths.

We also studied the distribution of the total volume of patterns for different algorithms on PF03880. Figure 2(g) shows that frequent substring and subsequence patterns disappear quickly as α increases, while the number of FAS-patterns drops more slowly. Note that all FAS-patterns in this diagram are longer than 15, whereas other patterns are permitted to have shorter lengths. So even for a support as high as 90%, we still discovered as many patterns as *Substr* and *Subseq* did, but with far longer lengths.

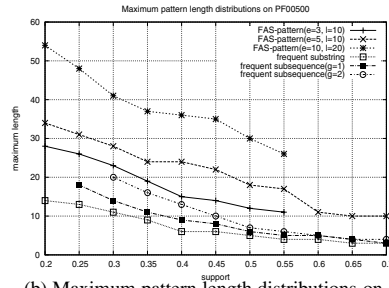
Runtime (varying α). The running times of different algorithms are shown in diagrams (h), (i), (j) and (k) of Figure 2. *FAS-Miner* with full optimization is denoted as *FAS-Miner_{op}* and a version without optimization of preserving

³There are some missing values for FAS-patterns, when $\epsilon = 3$, since no patterns was found. The other algorithms didn’t have this length constraint applied and thus some patterns with shorter lengths could be found.

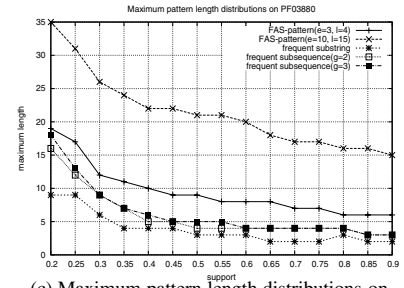
Figure 2. Experimental results.



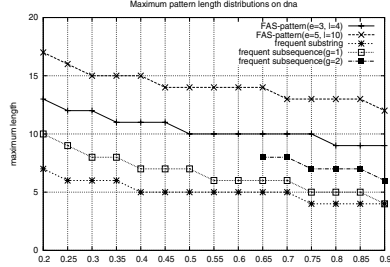
(a) Maximum pattern length distributions on snake.



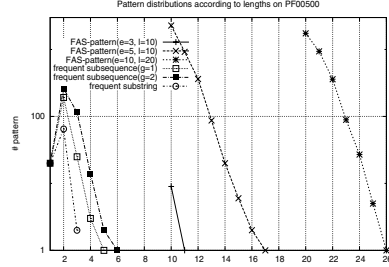
(b) Maximum pattern length distributions on PF00500.



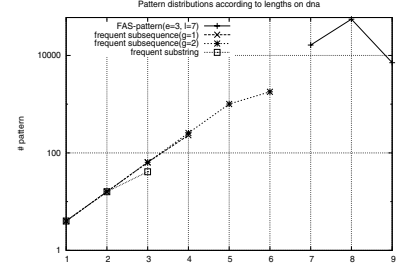
(c) Maximum pattern length distributions on PF03880.



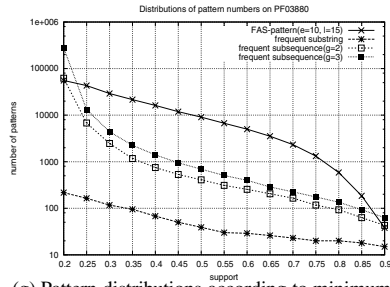
(d) Maximum pattern length distributions on DNA.



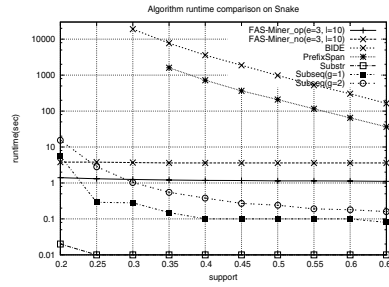
(e) Pattern distributions according to lengths on PF00500. $\alpha = 55\%$.



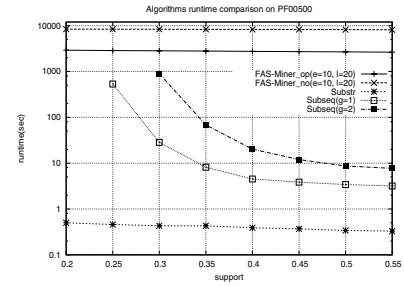
(f) Pattern distributions according to lengths on DNA. $\alpha = 90\%$.



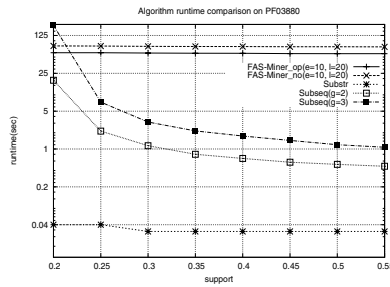
(g) Pattern distributions according to minimum supports on PF03880.



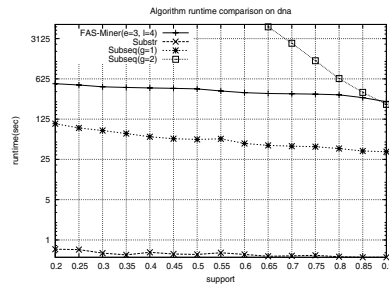
(h) Algorithms runtime comparison on snake.



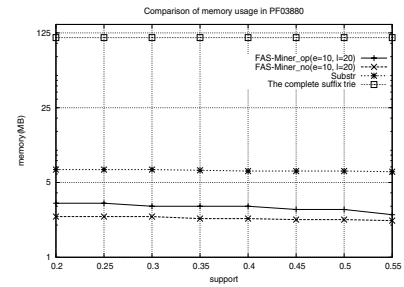
(i) Algorithms runtime comparison on PF00500.



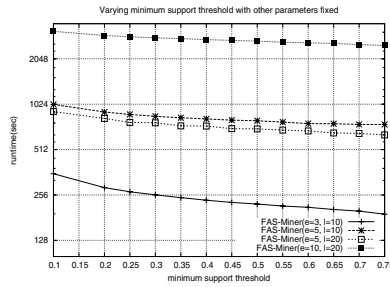
(j) Algorithms runtime comparison on PF03880.



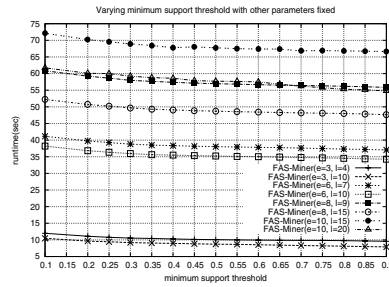
(k) Algorithms runtime comparison on DNA.



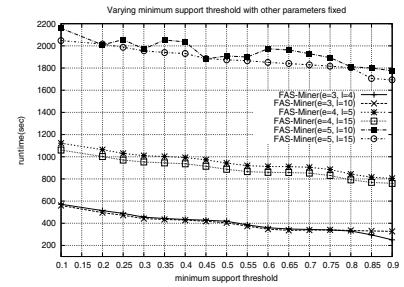
(l) Memory usage comparison on PF03880.



(m) Varying α on PF00500.



(n) Varying α on PF03880.



(o) Varying α on DNA.

da with a longer eyesight is denoted as *FAS-Miner_{no}*.

From Figure 2(h), we see that *PrefixSpan* and *BIDE* are insufficient to handle mining tasks from long sequences with small alphabet sets. The running time of *Subseq* with small gaps is faster than *PrefixSpan* and *BIDE*, but its performance markedly deteriorates for larger gaps (Figure 2(k)). *Substr* is easily the most efficient algorithm overall.

There are two main points regarding the efficiency of *FAS-Miner*. Firstly, the improvement gained through using the optimization is obvious. Indeed, a 25% runtime improvement in all datasets was observed. Secondly, *FAS-Miner* experiences little change in runtime as α varies, because most of the time was spent in checking the distance between the candidate and all the substrings of the same length. Another reason is, we have to fully generate the search space of all candidates with lengths smaller or equal to ϵ and this takes up considerable proportion of the total running time.

We also measured *FAS-Miner* running time by varying α and fixing other thresholds. The results are given in diagrams (m), (n) and (o) of Figure 2. They show that *FAS-Miner* is affected mostly by the increase of ϵ .

6. Related work

Frequent and closed subsequence pattern mining has been studied thoroughly in [1, 9, 12, 13]. Algorithms considering various constraints, such as [10, 14] have also been popular. [2] introduces a pattern which is like a combination of short frequent substrings having a maximum gap constraint between any two adjacent substrings. Emerging substrings and distinguishing subsequences are studied in [4, 7]. Suffix trees and suffix arrays are frequently used data structures in substring matching problems. [6, 5] introduce how to use suffix arrays and trees for fast mining of frequent string patterns.

[8] discusses mining approximate sequential patterns. Sequences are clustered into groups, based on edit distance. Then, an approximate multiple alignment is done within each cluster and profiles are generated from frequencies of aligned items. [3] looks at mining approximate sequential patterns, where the patterns mined are constrained to be within a certain distance from a user-defined reference pattern.

There are varieties of string patterns in biomolecular sequences known as motifs. Many motif finding techniques require multi-sequence alignment, which is rather different from the case of *FAS*-patterns. Other motif algorithms are enumeration based [11], but typically require the length of the motif to be specified in advance. Also, they are designed to discover motifs that are relatively short (around 10-15 items in length). In contrast, *FAS-Miner* can discover very

long patterns, sometimes reaching up to 80 items in length and often over 50.

7. Conclusion

We have studied the mining of frequent approximate substrings. These patterns tend to have high quality, i.e. having both long lengths and high supports. They cannot be found using previous sequential mining algorithms. Experiments show that *FAS-Miner* works well for a number of challenging datasets, being able to discover very high quality patterns within reasonable time and memory limits.

References

- [1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of IEEE ICDE*, pages 3–14, 1995.
- [2] H. Arimura, H. Asaka, H. Sakamoto, and S. Arikawa. Efficient discovery of proximity patterns with suffix arrays. In *Proc. Comb. Pattern Matching*, pages 152–156, 2001.
- [3] M. Capelle, C. Masson, and J. F. Boulicaut. Mining frequent sequential patterns under a similarity constraint. In *Proc. of IDEAL*, pages 1–6, 2002.
- [4] S. Chan, B. Kao, C. L. Yip, and M. Tang. Mining emerging substrings. In *Proc. of DASFAA*, 2003.
- [5] S. Dan Lee and L. De Raedt. An efficient algorithm for mining string databases under constraints. In *Proceedings of KDD*, pages 108–129, 2004.
- [6] J. Fischer, V. Heun, and S. Kramer. Fast frequent string mining using suffix arrays. In *Proc. of IEEE ICDM*, pages 609–612, 2005.
- [7] X. Ji, J. Bailey, and G. Dong. Mining minimal distinguishing subsequence patterns with gap constraints. In *Proc. of IEEE ICDM*, pages 194–201, 2005.
- [8] H. Kum, J. Pei, W. Wang, and D. Duncan. Approxmap: Approximate mining of consensus sequential patterns. In *Proc. of SDM*, 2003.
- [9] 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 *Proc. of IEEE ICDE*, pages 215–224, 2001.
- [10] J. Pei, J. Han, and W. Wang. Mining sequential patterns with constraints in large databases. In *Proc. of CIKM*, pages 18–25, 2002.
- [11] M. F. Sagot. Spelling approximate repeated or common motifs using a suffix tree. In *Proc. of LATIN*, pages 374–390, 1998.
- [12] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. of ICDE*, pages 79–90, 2004.
- [13] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large databases. In *Proc. of SDM*, 2003.
- [14] M. J. Zaki. Sequence mining in categorical domains: Incorporating constraints. In *Proc. of CIKM*, pages 422–429, 2000.