Scalable Sequential Pattern Mining for Biological Sequences [*]

Ke Wang
Simon Fraser University
wangk@cs.sfu.ca

Yabo Xu
Simon Fraser University & Chinese University of Hong Kong
yxu@cs.sfu.ca

Jeffrey Xu Yu
Chinese University of Hong Kong
yu@se.cuhk.edu.hk

ABSTRACT

Biosequences typically have a small alphabet, a long length, and patterns containing gaps (i.e., "don't care") of arbitrary size. Mining frequent patterns in such sequences faces a different type of explosion than in transaction sequences primarily motivated in market-basket analysis. In this paper, we study how this explosion affects the classic sequential pattern mining, and present a scalable two-phase algorithm to deal with this new explosion. The *Segment Phase* first searches for short patterns containing no gaps, called *segments*. This phase is efficient. The *Pattern Phase* searches for long patterns containing multiple segments separated by variable length gaps. This phase is time consuming. The purpose of two phases is to exploit the information obtained from the first phase to speed up the pattern growth and matching and to prune the search space in the second phase. We evaluate this approach on synthetic and real life data sets.

Categories and Subject Descriptors: H.2.8 [Database Applications]: Data Mining; J.3 [Life and Medical Sciences]: Biology and Genetics

General Terms: algorithms, management, performance

Keywords: algorithm, bioinformatics, frequent pattern, pruning technique, sequence, sequential pattern

1. INTRODUCTION

One important problem arising from bio-applications is the discovery of sequential patterns that occur in many biosequences in a given database (i.e., DNA or protein sequences). Such "frequent patterns" typically correspond to residues conserved during evolution due to an important structural or functional role. Finding frequent patterns often is the first step in sequence analysis such as classifying sequences, extracting species-specific features, identifying transcription factor binding sites, etc. The focus of this paper is scalable techniques for mining frequent patterns from a large database of biosequences.

In biology, various tools have been developed for searching for similarity among biosequences. A well known tool is BLAST [4]. The idea is *aligning* sequences so that similarity can be revealed in the presence of small variations in position. *Sequential pattern mining* developed in data mining searches for all frequent patterns in "transaction sequences" motivated in marketplaces, see [6, 18, 10, 17, 16, 15] for example. A transaction sequence can be a purchase sequence, a web link click stream, etc. The focus of those works is on the scalability on large databases. A natural solution is to sequential pattern mining to biosequences. Our experiments show that this solution does not scale because of the following features for biosequences.

Small alphabet. Biosequences have a very small alphabet, i.e., 4 for DNA sequences and 20 for protein sequences, and many short patterns occur in most sequences. In contrast, transaction sequences have a large alphabet, ranging from 1,000 to 10,000, and only a tiny fraction of items occurs in a transaction sequence. With most items occurring in every biosequence, pruning strategies and data structures based on the sparsity or absence of items, such as the hash-tree [18, 17] and the idlist/bitmap representation [6, 16], are not effective for biosequences.

Long sequence length. A biosequence has a typical length of few hundreds, sometime thousands [1]. In contrast, a transaction sequence has a typical length from 10 to 20. A long sequence (especially, with a small alphabet) often contains long patterns. The classic sequential pattern growth of one item at a time, as in [**?**, 16, 18, 15], requires many database scans and high frequency of pattern matching.

Gapped patterns over long regions. Biosequence patterns have the form of $X_1 * \cdots * X_k$ spanning over a long region, where each X_i is a short region of consecutive items, called a *segment*, and * denotes a variable length gap corresponding to a region not conserved in the evolution. The presence of * implies that pattern matching is more permissible and involves the whole range in a sequence.

These features create a different type of explosion of patterns. In this paper, we study the effect of these features on the classic sequential pattern mining, and propose a two-phase mining strategy to better deal with the new type of explosion. The first phase finds frequent segments X_i effi-

[*]The work was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada, grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (CUHK4229/01E, CUHK2050292).

[1]http://www.ncbi.nlm.nih.gov

ciently by an existing technique. The second phase grows patterns $X_1*\cdots*X_{k-1}*X_k$ rapidly one segment at a time, as opposed to one item at a time. The essence of this two-phase approach is leveraging some information about X_i obtained in the first phase to prune patterns and speed up pattern matching in the second phase. Particularly, based on such information, we propose indexing/compression methods to reduce the work of pattern matching, and propose a novel pattern enumeration scheme to prune the search space. The details are explained in subsequent sections.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 defines the problem and presents an overview of our approach. Section 4 presents the first phase of the algorithm. Section 5 presents the second phase. Section 6 evaluates the performance of the proposed method. Section 7 discusses possible extensions. Section 8 concludes the paper.

2. RELATED WORK

In bioinformatics, multiple sequence alignment was used to find similarity of several sequences, see [8] for a detailed survey on this technique. This notion is useful when an entire sequence is similar, but makes no sense if the sequences under comparison are distintly related. Pattern or motif discovery addresses this problem by considering only regions that are conserved among sequences. A survey of algorithms for pattern discovery can be found in [2]. Most approaches work only for small size problems because they enumerate the entire solution space. Several approaches use heuristics or structural restriction of patterns, such as maximum gaps or pattern length allowed, to reduce the search space, but at the expense of missing some useful patterns or sacrificing the completeness of the results [11, 14, 9, 13].

Mining sequential patterns was studied in market-basket analysis [6, 18, 10, 15, 17, 16], where a sequence represents purchase transactions for a customer. Pattern matching is by hash-partitioning of candidate patterns [18, 17], intersecting idlists or bitmaps of patterns [6, 16], partitioning sequences [15]. Based on the absence of items or patterns, these techniques aim at the type of explosion for short sequences over a large alphabet, but are less effective for biosequences that are typically long and over a small alphabet. In addition, these methods do not take advantage of the gapped structure $X_1*\cdots*X_k$ for pattern pruning and pattern matching.

Mining long patterns was examined for set-valued transactions where each item occurs *at most once* [1, 7, 3]. The idea is to lookahead longest patterns first, by extending the current pattern with all "remaining items" [1, 7, 3]. For sequential patterns, since an item can occur *repeatedly* in a pattern, there is no corresponding notion of "remaining items". Our algorithm finds all frequent patterns because sequence analysis typically considers both support and confidence, therefore, maximal patterns are not necessarily ranked highest. The sampling/bordering method [20, 19] finds the border between frequent patterns and infrequent ones using a sample of the database, then adjusts the border on the entire database. For biosequences, sampling is less effective because sampling does not change the long sequence length or the small alphabet size, which are the major factors for explosion of patterns.

3. THE OVERVIEW

A database D is a collection of sequences $\{s_1,\cdots,s_N\}$. Each sequence s_i is an ordered list of items chosen from a fixed alphabet. $< s_i, j >$ denotes the jth position in a sequence s_i, where $j \geq 1$. A *segment* refers to one or more items at consecutive positions in a sequence. A *pattern* has the form $X_1*\cdots*X_n$ $(n \geq 1)$, where X_i is a segment and $*$ denotes the variable length "don't care" (VLDC). A pattern $X_1*\cdots*X_n$ *matches* a sequence s_i if each segment X_j matches itself and each * can substitute for zero or more items.

Useful patterns for sequences in D should occur frequently in sequences in D, but not in other sequences. For long sequences over a small alphabet, a segment X_i of a short length tends to occur in every sequence, similar to "stop words" that occur in every text document. Such trivial similarity is not discriminating, therefore, not useful for biology analysis. For example, it is known to biologists that a transcription factor binding site has a length from 6 to 15 [12]. We can specify a minimum segment length to exclude trivial segments.

Definition 3.1. *The* support *of a pattern is the percentage of the sequences in D that contain the pattern. Given a minimum segment length MinLen and a minimum support MinSup, a pattern $X_1*\cdots*X_n$ is* frequent *if $|X_i| \geq MinLen$ for $1 \leq i \leq n$ and the support of the pattern is at least MinSup. The problem of* mining sequence patterns *is to find all frequent patterns.* ■

We find all frequent patterns in two phases. The first phase, *Segment Phase*, finds all frequent segments X_i satisfying the minimum length. The second phase, *Pattern Phase*, generates frequent patterns $X_1*\cdots*X_k$ using X_i found in the first phase.

A key observation that our algorithm heavily uses is stated as follows.

Observations. Consider a pattern

$$P = X_1*\cdots*X_{k-1}*X_k$$

and a super-pattern of the form

$$P' = X_1*\cdots*X_{k-1}*X_k' \text{ or}$$
$$P' = X_1*\cdots*X_{k-1}*X_k*X_{k+1},$$

where X_k' contains X_k as a prefix. The following pruning strategies hold:

Pattern Generation Pruning. If $P*X$ fails to be a frequent pattern, so does $P'*X$. Therefore, we can prune $P'*X$.

Pattern Matching Pruning. If $P*X$ fails to occur before position i in sequence s, so does $P'*X$. Therefore, we only need to examine the positions after i when matching $P'*X$ against s.

To support these prunings, we need a strategy for enumerating the pattern space $X_1*\cdots*X_k$ so that P is enumerated before P', and we need to answer the following queries efficiently.

Definition 3.2. *A* position query $Q(X,s,i)$*: given a frequent segment X, a sequence id s, and a position i in s, find the smallest start position of X in s greater than i. If such a position j is found, return $< s, j >$; otherwise, return nil.* ■

In the subsequent sections, we present a two-phase algorithm based the above ideas of Pattern Generation Pruning and Pattern Matching Pruning.

4. SEGMENT PHASE

This phase finds all frequent segments and builds an auxiliary structure for answering position queries.

4.1 Finding base/frequent segments

We use the *generalized suffix tree (GST)* [14] to count support of segments. The time and space needed for constructing the GST is $O(|D|)$, where $|D|$ is the total length of the sequences in D. We extract the following information from the GST. (1) The frequent segments of length $MinLen$, B_i, called *base segments*, and the *position lists* for each B_i, $s : p_1, p_2, \cdots$, where $p_j < p_{j+1}$ and each $< s, p_j >$ is a start position of B_i. (2) All frequent segments of length greater than $MinLen$. Note that we do not extract position lists for such frequent segments.

THEOREM 1. *The total length of position lists for base segments is no more than the total length of sequences in D.*

PROOF. Consider a single sequence. No two base segments occur at the same position in the sequence (otherwise, they are identical). Thus, the total length of the position lists involving the sequence is no more than the length of the sequence. The theorem follows from summing up over all sequences. ■

Below, we consider two methods for answering position queries $Q(X, s, i)$.

4.2 Index-based querying

In this method, we build an in-memory index for the positions of base segments. First, we rewrite each frequent segment X using base segments only. Consider two base segments B_1 and B_2, such that the last k items in B_1 are identical to the first k items in B_2, $k \geq 0$. The *k-join* of B_1 and B_2, denoted $B_1 \bowtie_k B_2$, is the segment obtained by overlapping the last k items of B_1 with the first k items of B_2.

COROLLARY 1. *A frequent segment X can be written into a sequence of k-joins of base segments, $0 \leq k < MinLen$:*

$$B_1 \bowtie_0 B_2 \bowtie_0 \cdots \bowtie_0 B_p \bowtie_k B_{p+1}.$$

■

EXAMPLE 4.1. *Table 1 shows a database of three sequences, with the alphabet $\{a, b, c, d\}$. Let $MinSup = 2/3$, and $MinLen = 2$. The following segments are frequent:*

$$ab(2),\ ac(3),\ acd(3),\ acda(2),\ cd(2),\ cda(2),\ da(2).$$

The integers in the brackets are support counts. The base segments and their position lists are given in Table 2. $ab * cda$ occurs in s_1 and s_2, so is a frequent pattern. We can write $ab * cda$ as $B_1 * (B_3 \bowtie_1 B_4)$ using only base segments. Similarly, $ab * acda$ is frequent and can be written as $B_1 * (B_2 \bowtie_0 B_4)$. ■

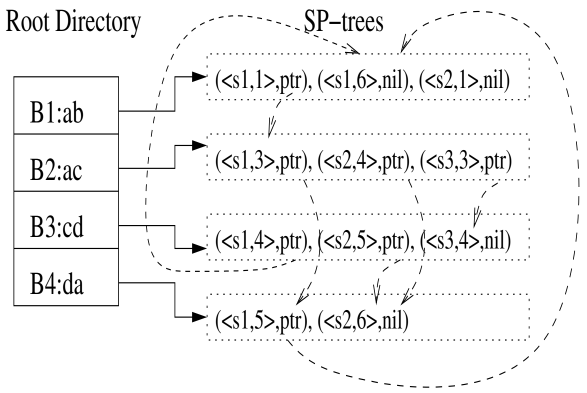
We build the following index using the position lists of base segments.

ID	Sequence
s_1	*abacdab*
s_2	*abcacda*
s_3	*baacdca*

Table 1: The database D

Base Segments	Position Lists
$B_1 = ab$	$(s_1 : 1, 6), (s_2 : 1)$
$B_2 = ac$	$(s_1 : 3), (s_2 : 4), (s_3 : 3)$
$B_3 = cd$	$(s_1 : 4), (s_2 : 5), (s_3 : 4)$
$B_4 = da$	$(s_1 : 5), (s_2 : 6)$

Table 2: The position lists



Root Directory
SP−trees
B1:ab
B2:ac
B3:cd
B4:da
(<s1,1>,ptr), (<s1,6>,nil), (<s2,1>,nil)
(<s1,3>,ptr), (<s2,4>,ptr), (<s3,3>,ptr)
(<s1,4>,ptr), (<s2,5>,ptr), (<s3,4>,nil)
(<s1,5>,ptr), (<s2,6>,nil)

Figure 1: The SP-index in Example 4.1

DEFINITION 4.1. *The* SP-index (Segment-to-Position index) *has two components, the* root directory *and the* SP-trees*. For each B_i, the root directory has an entry for the root of the SP-tree for B_i. The* SP-tree for B_i *is a B-tree for indexing the start positions $< s, p >$ of B_i in all sequences s. A leaf entry has the form $(< s, p >, ptr)$. Unlike the standard B-tree, ptr points to the leaf entry $(< s, p' >, ptr')$ for the next base segment in Corollary 1 if there is one, or else nil.* ■

EXAMPLE 4.2. *Figure 1 illustrates the sketch of SP-index for the base segments in Example 4.1. Only the leaf entries are shown. The entry $(< s_1, 3 >, ptr)$ links to entry $(< s_1, 5 >, ptr)$ because $B_2 \bowtie_0 B_4$ is a frequent segment. The entry $(< s_3, 3 >, ptr)$ links to entry $(< s_3, 4 >, ptr)$ because $B_2 \bowtie_1 B_3$ is a frequent segment.* ■

We explain how to compute a query $Q(X, s, i)$ using the SP-index. Let $X = B_1 \bowtie \cdots \bowtie B_m$, as in Corollary 1. We search the SP-tree for B_1 (like the B-tree) using the search key value $< s, i >$. Suppose that we reach a leaf entry $(< s, p >, ptr)$. Note that $< s, p >$ is the smallest start position of B_1 greater than i. To check if X actually occurs at this position, we follow the *ptr* link until $ptr = nil$. If the linked leaf entries represent the k-join for X, return the current position $< s, p >$ of B_1; otherwise, move to the next leaf entry for B_1 in sequence s, and repeat the above checking. This "move and check" is repeated until either the checking is successful, or either there is no more leaf entry for B_1 in sequence s, in which case we return nil.

Example 4.3. *Let us answer $Q(acda, s_1, 1)$ using the SP-index in Figure 1. Note that $acda = B_2 \bowtie_0 B_4$. First, we search the SP-tree for B_2 using the search key value $< s_1, 1 >$. The search reaches the leaf entry $(< s_1, 3 >, ptr)$. Then, we check if acda occurs at the current start position $< s_1, 3 >$ by following the ptr link. In this case, $(< s_1, 3 >, ptr)$ links to $(< s_1, 5 >, ptr)$ in the SP-tree for B_4, which shows that $B_2 \bowtie_0 B_4$ indeed occurs at $< s_1, 3 >$. So we return the current position $< s_1, 3 >$ of B_2 as the answer to the query.* ■

4.3 Compression-based querying

This method compresses all positions in a *non-coding region* into a *new* item ϵ that matches no existing item except *. A non-coding region contains no part of a frequent segment. We can scan each original sequence once, identify each consecutive region not overlapping with any frequent segment, collapse it into the new item ϵ. For a long sequence and large $MinLen$ and $MinSup$, a compressed sequence is typically much shorter than the original sequence. To answer the query $Q(X, S, i)$ over a compressed sequence S, we scan S sequentially because S is short. Note that ϵ in S does not match any item in X.

Example 4.4. *For the database in Example 4.1, the compressed sequences for s_1, s_2, s_3 are:*
S_1 : abacdab.
S_2 : abϵacda (c collapses into ϵ).
S_3 : acd (ba and ca collapse into leading ϵ and ending ϵ, which are deleted) ■

The compression-based querying is amenable to approximate pattern matching. We will elaborate on this in Section 7.

5. PATTERN PHASE

This phase generates all frequent patterns $X_1 * \cdots * X_k$ using frequent segments X_i found in Segment Phase. The key is to organize the search space for patterns $X_1 * \cdots * X_k$ so that the Pattern Generation Pruning and Pattern Matching Pruning mentioned in Section 3 can be easily exploited. The segment tree and pattern tree defined below describe this organization.

Segment tree (ST). The ST organizes frequent segments X into a tree so that if X is a prefix of X', X is enumerated before X' in the depth-first enumeration of the tree. A terminal edge is labeled by an integer $k \geq 0$. A non-root node w is labeled by a base segment B_i, and represents the frequent segment $B_1 \bowtie_0 \cdots \bowtie_0 B_{p-1} \bowtie_k B_p$, where $B_1, \cdots, B_{p-1}, k, B_p$ are the labels on the path from the root to w. Let $seg(w)$ denote the frequent segment represented by w.

Example 5.1. *Figure 2 shows the ST for Example 4.1, with w_i denoting the ith node in the depth-first enumeration of the ST. w_3 represents the frequent segment $seg(w_3) = B_2 \bowtie_1 B_3 = acd$, where $B_2, 1, B_3$ are the labels on the path from the root to w_3. w_4 represents the frequent segment $seg(w_4) = B_2 \bowtie_0 B_4 = acda$, and w_6 represents the frequent segment $seg(w_6) = B_3 \bowtie_1 B_4 = cda$.* ■

Pattern tree (PT). The PT organizes patterns $X_1 * \cdots * X_k$ into a tree so that a super-pattern is enumerated after a sub-pattern in the depth-first enumeration of the tree. A non-root node v is labeled by a frequent segment $seg(w_i)$, where w_i is a node in ST, and represents the pattern $seg(w_1) * \cdots * seg(w_k)$, where $seg(w_1), \cdots, seg(w_k)$ are the labels on the path from the root to v. Let $pat(v)$ denote the pattern represented by v. Furthermore, if $v_1, \cdots, v_n$ are the child nodes from left to right, with the labels $seg(w'_1)$, $\cdots$, $seg(w'_n)$, $w'_1, \cdots, w'_n$ are in the order of depth-first enumeration of ST.

Therefore, if (non-root node) w is the parent of w' in ST (therefore, $seg(w)$ is a prefix of $seg(w')$), the node for $P = X_1 * \cdots * X_{k-1} * seg(w)$ is the immediate left sibling of the node for $P' = X_1 * \cdots * X_{k-1} * seg(w')$ in PT, therefore, P is enumerated before P' in the depth-first enumeration of PT. Below, we sketch our algorithm of using this property to perform Pattern Generation Pruning and Pattern Matching Pruning.

Algorithm. We enumerate patterns $P = X_1 * \cdots * X_k$ in the order of depth-first enumeration of PT. At the current node v in PT, we maintain the set, $v.dead$, of highest nodes w in ST that failed to extend $P = pat(v)$, i.e., $pat(v) * seg(w)$ is not frequent. Following the Pattern Generation Pruning, for each node w in $v.dead$, the subtree at w are pruned from extending P. We also maintain the smallest position i of each sequence at which $pat(v)$ occurs. We use such positions i in the query $Q(X, s, i)$ to restrict the region in a sequence s when matching $pat(v) * X$ against s. As we advance from v to the immediate right sibling or the first child node v' in the depth-first enumeration of PT, $P' = pat(v')$ is a super-pattern of $P = pat(v)$. Following Pattern Generating Pruning and Pattern Matching Pruning, we obtain $v'.dead$ as $v.dead$ and obtain the smallest position at v' as the position returned by $Q(X, s, i)$ at v.

Example 5.2. *Figure 3 shows a part of PT generated using the ST in Figure 2. $v_i : seg(w)$ denotes the ith node in the depth-first enumeration of PT, with $seg(w)$ being the label. Initially, $v_1.dead = \emptyset$. At v_2, extending v_1 by w_1, i.e., $pat(v_1) * seg(w_1)$, is not successful, indicated by a dashed line, so $v_1.dead = \{w_1\}$. At v_3, $pat(v_1) * seg(w_2)$ (= $ab * ac$) is frequent. We set $v_3.dead$ to $v_1.dead = \{w_1\}$, At v_4, extending v_3 by w_2 (= $ab * ac * ac$) is not successful, so $v_3.dead = \{w_1, w_2\}$. At v_5, extending v_3 by w_5 (= $ab*ac*cd$) is not successful, so $v_3.dead = \{w_1, w_2, w_5\}$. At v_6, extending v_3 by w_7 (= $ab * ac * da$) is successful. At v_7, the label of v_3, i.e., $seg(w_2) = ac$, is a prefix of the label of v_7, i.e., $seg(w_3) = acd$, so $v_7.dead = v_3.dead = \{w_1, w_2, w_5\}$.* ■

6. EXPERIMENTS

We evaluated the proposed two-phase depth-first enumeration algorithm, denoted 2PDF. 2PDF-Index denotes the index-based method and 2PDF-Compression denotes the compression-based method. We compared these algorithms with two sequential pattern mining algorithms developed in the data mining field, PrefixSpan [15] [2] and SPAM [6], which have shown superior performance on transaction sequences compared to earlier algorithms such as [18, 17, 16]. Our purpose is to see how these algorithms, primarily designed for market-basket analysis, would respond to the new type of explosion in biosequences. Since our algorithm finds the

[2]For PrefixSpan, we used the *pseudo-projection* technique as suggested in [15], which makes PrefixSpan faster than SPAM.

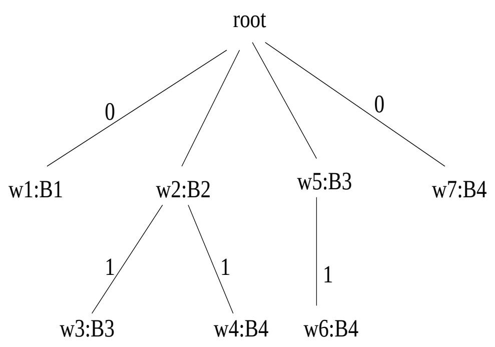


Figure 2: The segment tree in Example 5.1

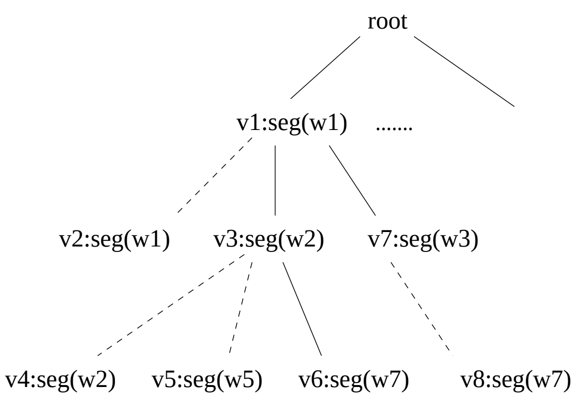


Figure 3: The pattern tree in Example 5.2

Symbol	Meaning in [18]
D	Number of customers (number of sequences)
C	Average number of transactions per customer (length of sequences)
T	Average number of items per transaction (=1)
S	Average length of maximal potentially frequent sequences
I	Average size of itemsets in maximal potentially frequent sequences (=1)
N_S	Number of maximal potentially frequent sequences
N_I	Number of maximal potentially frequent itemsets (=N)
N	Number of items (=4 or 20)

Table 3: Parameters of the data generator

Simulated category	Name	C	S	N	D	MinLen
DNA sequences	$C256S64N4D100K$	256	64	4	100K	7
	$C128S32N4D100K$	128	32	4	100K	5
Protein sequences	$C256S64N20D100K$	256	64	20	100K	3
	$C128S32N20D100K$	128	32	20	100K	3
Transaction sequences	$C20S8N10000D100K$	20	8	10,000	100K	1

Table 4: Synthetic data sets

complete set of frequent patterns, we did not compare with methods that find a subset of frequent patterns. All experiments were conducted on a PC with 2GHZ CPU and 1GB memory running the Windows 2000 Professional.

6.1 Synthetic data sets

The first set of experiments was conducted on the synthetic data sets generated in [18]. Table 3 shows the parameters used and Table 4 shows the names of data sets. The alphabet size N and sequence length C characterizes the explosion of search space. The data sets with $N = 4$ simulate DNA sequences, the data sets with $N = 20$ simulate protein sequences, and the data set with $N = 10,000$ simulates transaction sequences. The DNA or protein sequences have significantly longer average length C, i.e., 128 or 256, than transaction sequences, i.e., 20. In general, for larger C and smaller N, we use a larger $MinLen$ due to more expected local similarity. Like in [18], N_S was set to 5000.

Execution time. Figures 4-7 show the result on biosequences. The first column shows the execution time in *logarithm scale* (i.e., $log_{10}T$ for execution time T) against $MinSup$. For 2PDFs, this includes the time in Segment Phase (i.e., computing frequent segments, building the SP-index or compressing sequences) and Pattern Phase. Both 2PDF-Index and 2PDF-Compression are several orders of magnitude faster than PrefixSpan and SPAM on long sequences of a small alphabet. Several factors contributed to this speedup: the reduced frequency of pattern matching because of "one segment at a time" pattern growth, the more aggressive Pattern Generation Pruning and the Pattern Matching Pruning. This experiment also shows that 2PDF-Index is more scalable for a small $MinSup$ than 2PDF-Compression. Table 6 shows the number of base segments, frequent segments, and frequent patterns for

$$C128S32N4D100K \text{ at } MinLen = 5.$$

Figure 8 shows the result on the transaction data set with the execution time in the first figure. For this dataset, we set $MinLen = 1$, which is the worst case for the 2PDF methods. There are two findings. First, mining biosequences is much more difficult than mining transaction sequences, as indicated by the much smaller execution time and minimum support here compared to Figures 4-7. Note that the actual time is used in Figure 8 instead of the logarithm scale. Second, the 2PDFs, though designed for long sequences of a small alphabet, are also highly competitive on short sequences of a large alphabet.

Space consumption. The second column of Figures 4-8

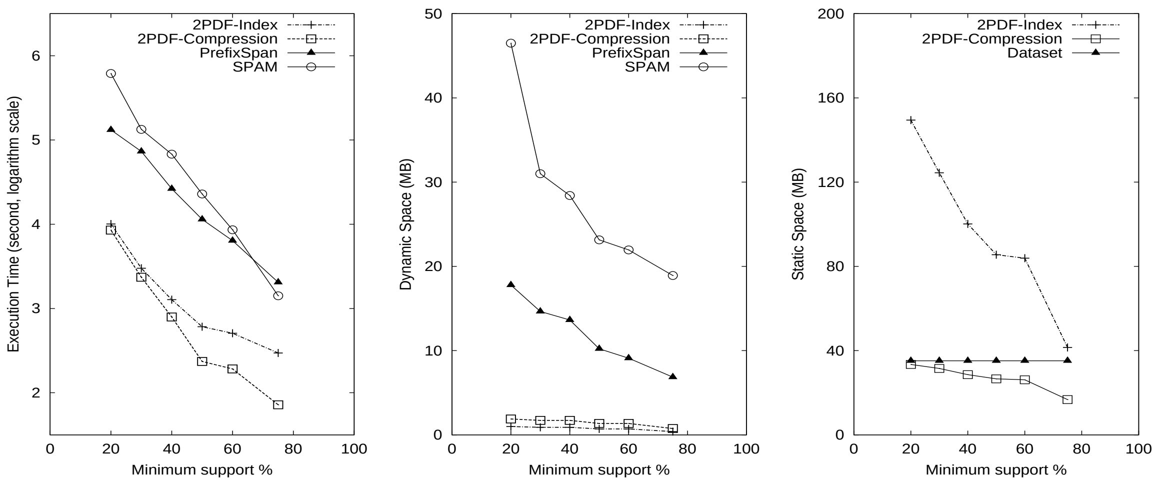


Figure 4: C128S32N4D100K, MinLen = 5

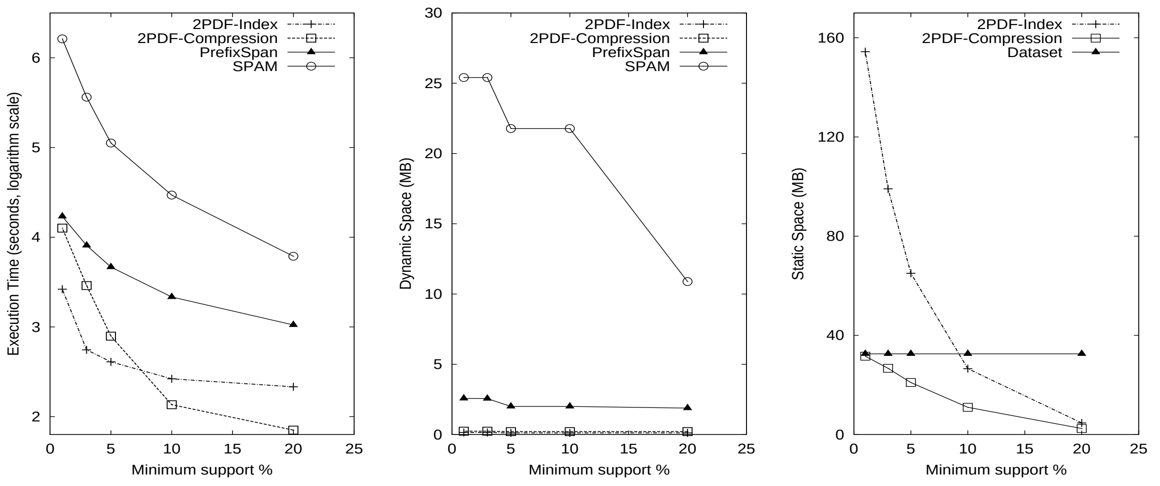


Figure 5: C128S32N20D100K, MinLen = 3

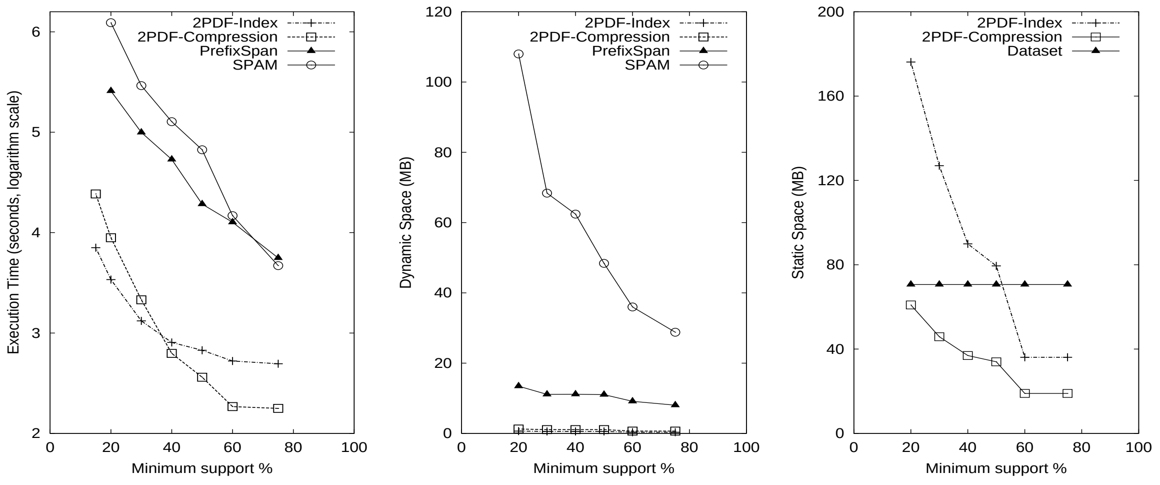


Figure 6: C256S64N4D100K, MinLen= 7

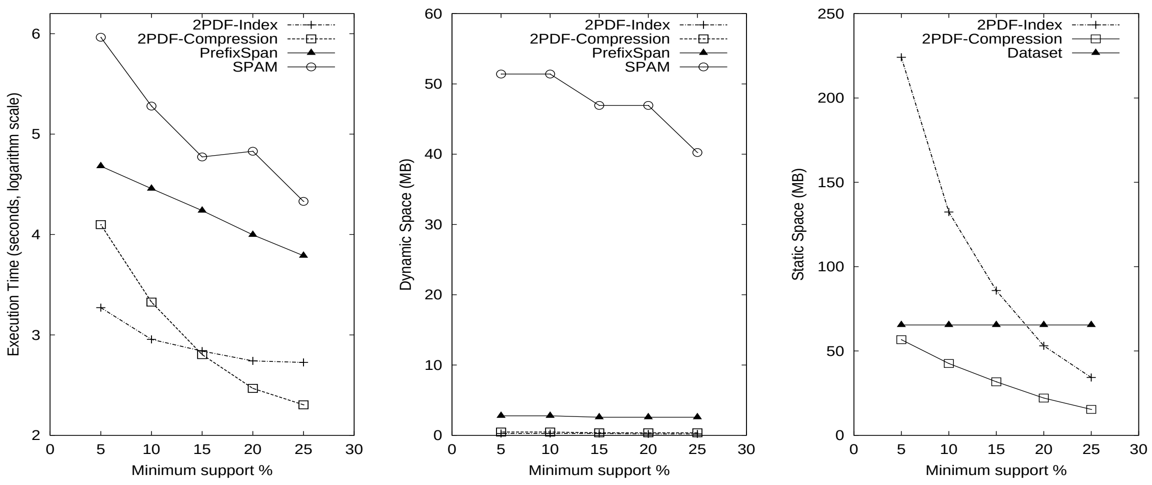


Figure 7: C256S64N20D100K, MinLen= 3

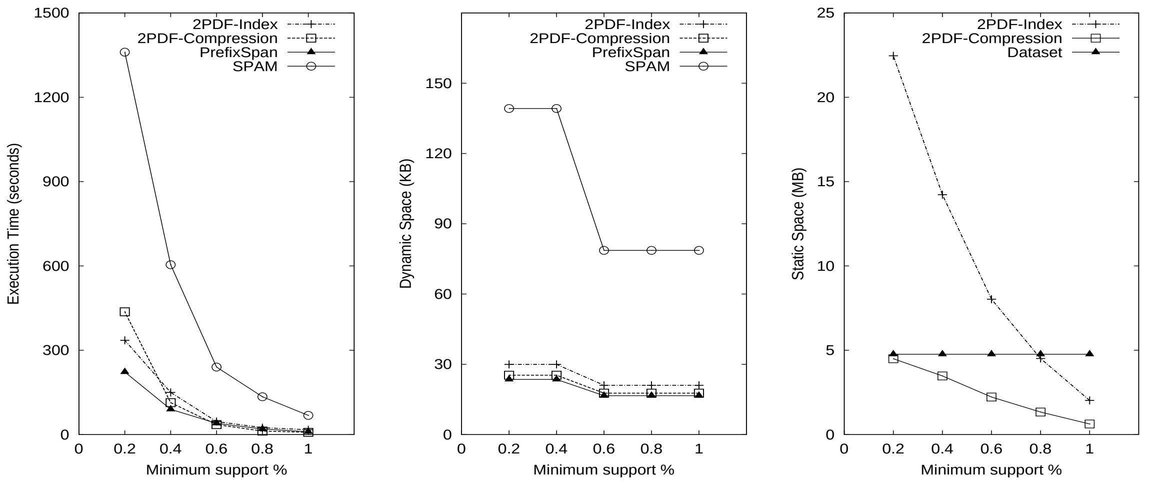


Figure 8: C20S8N10000D100K, MinLen = 1

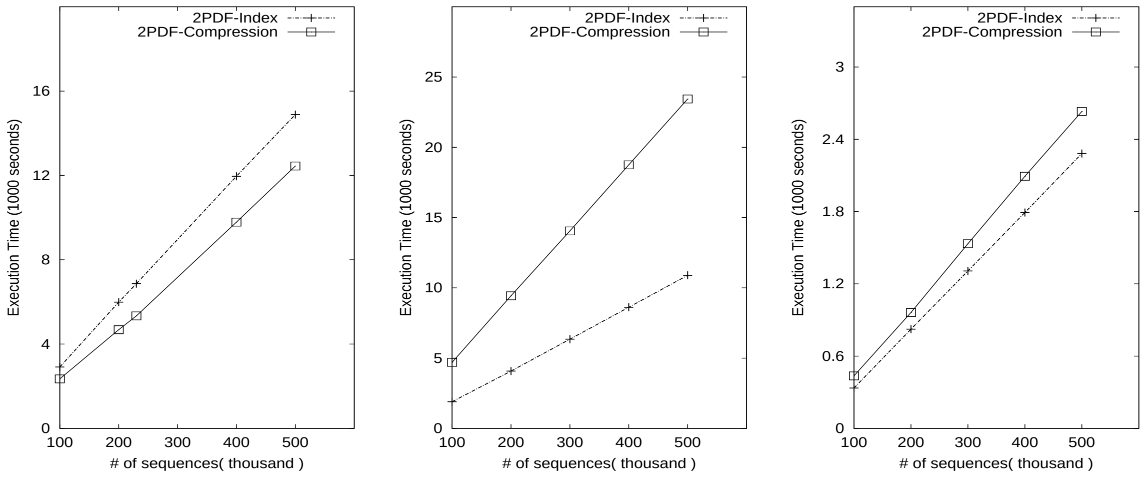


Figure 9: Scalability wrt the database size

Data set	# sequences	Min length	Max length	Avg length
DNA data set	122,855	200	300	254
Protein data set	192,497	150	250	198

Table 5: Biological data sets

MinSup	# base segment	# segment	# pattern
5%	223	1288	557722
10%	142	602	471015
15%	101	313	18502
20%	91	240	6131

Table 6: Statistics for C128S32N4D100K, MinLen=5

shows the maximum space used by the current depth-first path in all algorithms, called "Dynamic Space". The two 2PDFs required little dynamic space because only one position is kept for each sequence. The last column of Figures 4-8 shows the space for storing the index in 2PDF-Index and compressed database in 2PDF-Compression, called "Static Space", compared to the input database size denoted by "Dataset". For a small $MinSup$, the static space of 2PDF-Index is high. Our current implementation stores a sequence id repeatedly in every entry for the sequence in the SP-trees. This static space can be significantly reduced by eliminating this repeated store using the partial-key technique [5]. The static space for 2PDF-Compression is much less than the database size.

Scalability. Figure 9 shows, from left to right, the execution time after scaling up the database size up to 500K for three challenging data sets:

$$C128S32N4D100K\ (MinLen = 5,\ MinSup = 30\%),$$
$$C256S64N4D100K\ (MinLen = 7,\ MinSup = 25\%),$$
$$C20S8N10000D100K\ (MinLen = 1,\ MinSup = 0.2\%).$$

The 2PDFs show a linear scalability with respect to the database size. On the most time-consuming $C256S64N4D100K$ (the center figure), 2PDF-Index is superior to 2PDF-Compression. This confirms the intuition that the index method has a better scalability for large data sets.

6.2 Biological data sets

The second set of experiments was conducted on real life DNA and protein sequences extracted from the web site of National Center for Biotechnology Information [3]. The DNA data set was extracted by the conjunction of (1) search category ="Nucleotide", (2) sequence length range=[200:300], and (3) data submission period=[2002/12, 2003/02]. The protein data set was extracted by the conjunction of (1) search category="Protein", (2) sequence length range=[150:250], and (3) data submission period=[2002/12, 2003/02]. The statistics of these data sets is given in Table 5.

Figures 10-11 show the execution time, dynamic space and static space. The comparison of 2PDFs with PrefixSpan and SPAM is similar to that for synthetic data sets in Section 6.1. This experiment confirmed the superiority of the proposed methods on real life biosequence data.

[3]http://www.ncbi.nlm.nih.gov

7. EXTENSION

So far, approximate matching is allowed through variable length gaps $*$ in a pattern $X_1 * \cdots * X_k$. We can extend approximate matching to segments X_i using the standard *edit distance* measured by the number of insertion, deletion and mutation on items to achieve exact matching. A segment X *approximately matches* a segment Y if the edit distance between X and Y is no more than the specified maximum tolerance. A sequence s *approximately contains* a pattern $X_1 * \cdots * X_k$ if for $1 \leq i \leq k$, X_i approximately matches a segment Y_i in s and $*$ matches zero or more item, such that the total edit distance is no more than the specified maximum tolerance.

The index-based method no longer works for the edit distance based approximate matching because it has to index all "approximate base segments", which can be too large. The compression-based method is amenable to approximate matching. A *non-coding region* now is defined as a region that approximately matches no part of any frequent segment. The approximate matching will reduce the effectiveness of compression. Such a complexity increase is expected for any method because approximate matching leads to a larger solution space. On the other hand, approximate matching has no impact on Pattern Generation Pruning and Pattern Matching Pruning observed in Section 3. Therefore, the compression-based method continues to work for approximate matching, provided that the compression and pattern matching take into account of this change.

8. CONCLUSION

Biosequences experience a different type of explosion of search space from that for classic transaction sequences, and traditional pruning techniques are not effective for mining biosequences. Particularly, growing a pattern $X_1 * \cdots * X_k$ one item at a time requires many database scans and high frequency of pattern matching. We proposed a two-phase algorithm to address this problem. The novelty is using the information obtained about local patterns X_i in the first phase to help reduce the search of global patterns $X_1 * \cdots * X_k$ in the second phase. Experiments on both synthetic and real life data sets demonstrated significant speed up over sequential pattern mining methods.

9. REFERENCES

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad, *Depth first generation of long patterns*, SIGKDD, 2000.

[2] A. Brazma, I. Jonassen, I. Eidhammer, and D. Gilbert, *Approaches to the automatic discovery of patterns in biosequences*, Technical report, Department of Informatics, University of Bergen, Norway, 1995.

[3] D. Burdick, M. Calimlim, and J. Gehrke, *MAFIA: A maximal frequent itemset algorithm for transactional databases*, ICDE, 2001.

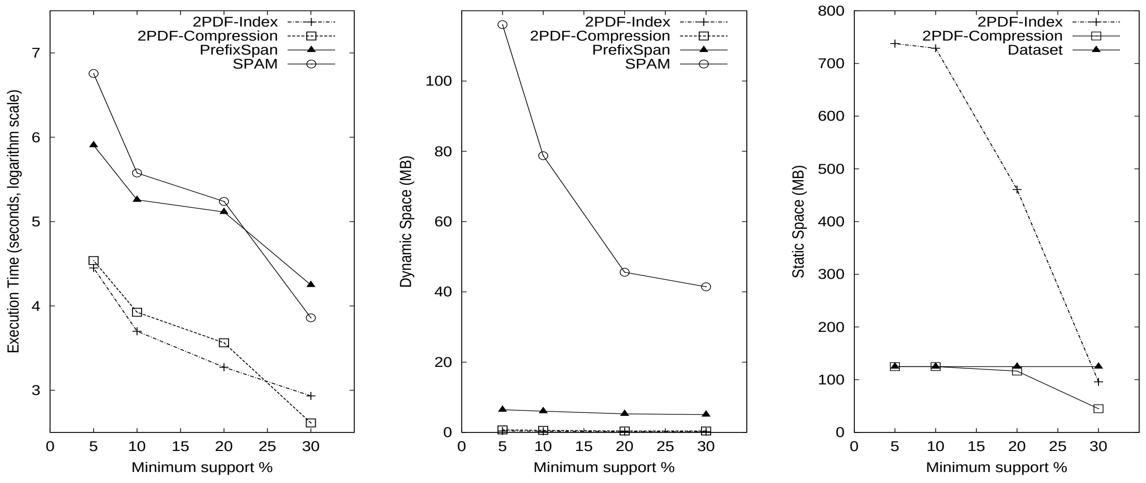


Figure 10: The real life DNA dataset, MinLen = 5

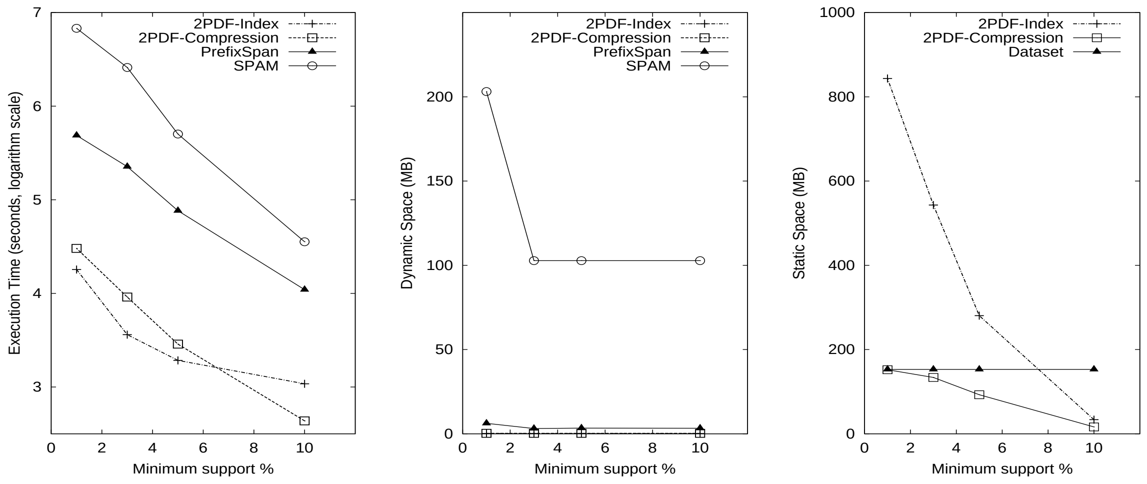


Figure 11: The real life Protein dataset, MinLen = 3

[4] *BLAST*, http://www.ncbi.nlm.nih.gov/Education/BLASTinfo/information3.html.

[5] P. Bohannon, P. Mcllroy, and R. Rastogi, *Main-memory index structures with fixed-size partial keys*, SIGMOD, 2001.

[6] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick, *Sequential pattern mining using a bitmap representation*, SIGKDD, 2002, pp. 215–224.

[7] R. J. Bayardo, *Efficiently mining long patterns from databases*, SIGMOD, pp. 85–93, 1998.

[8] Hirosawa et at, *Comprehensive study on iterative algorithms of multiple sequence alignment*, Comput. Applic. Biosci, Vol. 11, pp. 13–18, 1995.

[9] I. Jonassen, J.F. Collins, and D.G. Higgins, *Finding flexible patterns in unaligned protein sequences*, Protein Sci., Vol. 4, pp. 1587–1595, 1995.

[10] H. Mannila, H. Toivonen, and A. I. Verkamo, *Discovery of frequent episodes in event sequences*, Journal of Data Mining and Knowledge Discovery, Vol. 1, pp. 259–289, 1997.

[11] A.F. Neuwal and P. Green, *Detecting patterns in protein sequences*, J. Mol. Biol., Vol. 239, pp. 698–712, 1994.

[12] S. Sinha and M. Tompa, *A statistical method for finding transcription factor binding sites*, Intelligent Systems for Molecular Biology, 2000.

[13] M.F. Sagot and A. Viari, *A double combinatorial approach to discovering patterns in biological sequences*, Symposium on Combinatorial Pattern Matching, pp. 186–208, 1996.

[14] J.T.L. Wang, G.W.chirn, T.G. Marr, B. Shapiro, D. Shasha, and K. Zhang, *Combinatorial pattern discovery for scientific data: some preliminary results*, SIGMOD, 1994.

[15] J. Pei, J. Han, B. Asl, Q. Chen, U. Dayal, and M. Hsu, *Prefixspan: mining sequential patterns efficiently by prefix-projected pattern growth*, ICDE, 2001.

[16] M. J. Zaki, *SPADE: An efficient algorithm for mining frequent sequences*, Machine Learning Journal, Special Issue on Unsupervised Learning, Vol. 42, No. 1/2, pp. 31–60, 2001.

[17] R. Srikant and R. Agrawal, *Mining Sequential Patterns: Generalizations and Performance*

Improvements, Proc. 5th Int. Conf. Extending Database Technology, EDBT, Vol. 1057, Springer-Verlag, pp. 3–17, 1996.

[18] R. Agrawal and R. Srikant, *Mining Sequential Patterns*, ICDE, 1995.

[19] J. Yang, W. Wang, P.S. Yu, and J. Han, *Mining long sequential patterns in a noisy environment*, SIGMOD, 2002.

[20] H. Toivonen, *Sampling large databases for association rules*, VLDB, 1996.