Under consideration for publication in Knowledge and Information Systems

Mining Minimal Distinguishing Subsequence Patterns with Gap Constraints

Xiaonan Ji*, James Bailey*, Guozhu Dong**

*NICTA Victoria Laboratory, Department of Computer Science and Software Engineering,
University of Melbourne, Victoria, Australia

**Department of Computer Science and Engineering, Wright State University, Ohio, USA

Abstract. Discovering contrasts between collections of data is an important task in data mining. In this paper, we introduce a new type of contrast pattern, called a **Minimal Distinguishing Subsequence (MDS)**. An MDS is a minimal subsequence that occurs frequently in one class of sequences and infrequently in sequences of another class. It is a natural way of representing strong and succinct contrast information between two sequential datasets and can be useful in applications such as protein comparison, document comparison and building sequential classification models. Mining MDS patterns is a challenging task and is significantly different from mining contrasts between relational/transactional data. One particularly important type of constraint that can be integrated into the mining process is the gap constraint. We present an efficient algorithm called *ConSGapMiner* (Contrast Sequences with Gap Miner), to mine all MDSs satisfying a minimum and maximum gap constraint, plus a maximum length constraint. It employs highly efficient bitset and boolean operations, for powerful gap based pruning within a prefix growth framework. A performance evaluation with both sparse and dense datasets, demonstrates the scalability of *ConSGapMiner* and shows its ability to mine patterns from high dimensional datasets at low supports.

Keywords: Data mining algorithm, sequential pattern, frequent pattern, emerging pattern, gap constraint, contrast pattern.

1. Introduction

Contrasting collections of data is an important objective in data mining and sequences are a particularly important form of data. In this paper, we introduce a new type of pattern that is useful for contrasting collections of sequences, called a *Minimal Distinguishing Subsequence (MDS)*. A distinguishing subsequence is a subsequence that

Received Nov 30, 2005
Revised Feb 1, 2006
Accepted April 1, 2006

appears frequently in one class of sequences, yet infrequently in another. A distinguishing subsequence is minimal if none of its subsequences is distinguishing. A key property of an MDS is that its items do not have to appear consecutively – there may be gaps between them. As mentioned in (Chan et al, 2003), in the analysis of purchase behaviours, web-logs and biochemical data (e.g. motifs research), sequence patterns with gaps are often much more useful than ones with no gaps.

There are many situations where MDSs are useful, such as the comparison of proteins, design of microarrays, characterisation of text and the building of classification models. We give two specific examples to highlight the idea.

Example 1.1. When comparing the two protein families[1] zf-C2H2 and zf-CCHC, we discovered a protein section *CLHH* appearing as a subsequence 141 times among a total of 196 protein sequences in zf-C2H2, but never appearing among the 208 sequences in zf-CCHC. This subsequence represents a very strong contrast feature, that is potentially interesting to biologists. From a classification perspective, an unknown protein sequence containing *CLHH* as a subsequence seems unlikely to be a member of the zf-CCHC family.

Indeed the potential usefulness of contrasts for protein datasets is highlighted by work in (She et al, 2003), where it is observed that biologists are very interested in identifying the most significant subsequences that discriminate between outer membrane proteins and non outer membrane proteins. Furthermore, the higher dimensional structure of proteins makes allowing gaps in a subsequence particularly important. Elements which have a gap between them in the sequence, may in fact be spatially very close in the 3-dimensional protein.

Example 1.2. Comparing the first and last books from the Bible, we found that the subsequences "having horns", "faces worship", "stones price" and "ornaments price" appear multiple times in sentences in the Book of Revelation, but never in the Book of Genesis. (The gap between the two words of each pair is ≤ 6 non trivial words.) Such pairs might be seen as a fingerprint associated with the Book of Revelation and may be of interest to Biblical scholars.

Items in an MDS do not necessarily have to appear immediately next to each other in the original sequences. However, subsequences in which items are far away from each other are likely to be less meaningful than those whose items are close in the original sequence. A key focus, therefore, is to set a gap constraint when mining the MDS set. This restricts the distance between neighbouring elements of the subsequence. The benefits are that the mining output is smaller and more intuitive and the mining process can be faster.

Challenges: Several challenges arise in the mining of MDS. The first is that the Apriori property does not hold for distinguishing subsequences, meaning that the subsequences of a distinguishing sequence are not necessarily distinguishing themselves. (In contrast, the property does hold for frequent subsequence patterns.) Hence, any bottom up mining strategy needs to employ extra techniques for pruning the search space. This is especially important, since the search space is exponential and the number of MDS patterns present in the data may also be very large.

The second challenge is that the MDS's frequency threshold cannot be set as high as it is in frequent subsequence mining. There, for some of the dense databases, the

[1] More information on the protein families can be found in Section 6.

thresholds may need to be set to at least 80% (Wang and Han, 2004). Using the same thresholds for MDS mining is likely to result in empty output. In MDS mining, thresholds below 30% are needed for dense databases.

The third challenge arises with respect to the gap constraint. Gap constraints have been considered in other contexts, such as episode pattern mining (Méger and Rigotti, 2004; Casas-Garriga, 2003; Zhang et al, 2005). Techniques there rely upon storing all possible occurrences in a list. For each candidate, a scan through the list is performed to test if it fulfills the gap constraint. This may be workable in pure frequent pattern mining under high frequency thresholds. However, since the gap constraint is not class preserved (see Section 4 for a brief explanation) (Zaki, 2000) and the search space is potentially larger in MDS mining, the occurrence list may be very large and thus such scans become very costly.

Our contributions: Besides introducing the concept of minimal distinguishing subsequences, we describe a new algorithm called *ConSGapMiner* (Contrast Sequences with Gap Miner), to efficiently mine the complete MDS set for a (minimum, maximum) gap constraint. We employ a novel technique using efficient bitset and boolean operations, to determine whether a candidate subsequence can satisfy the gap constraint. We also employ several other pruning strategies.

We also show how our general approach can easily accommodate the specification of a length constraint and how it can be easily modified to mine other kinds of patterns, such as frequent sequential patterns with gap constraints.

Experimental analysis shows that *ConSGapMiner* is able to efficiently mine MDSs from some very dense real-world databases, using a relatively low frequency threshold. Indeed, using the gap constraints, it is able to mine patterns for some very long proteins, in circumstances that would challenge the current generation of frequent subsequence miners.

Organization: The rest of the paper is organised as follows. Section 2 surveys related work in the area. Section 3 introduces the basic concepts used, as well as terminologies and notations used throughout the paper. Section 4 describes the basic *ConSGapMiner* algorithm. Discussion of extensions of the basic algorithm to include minimum gaps, length constraints and more complex minimization is provided in Section 5. Experimental and performance results are given in Section 6. Finally, future work is discussed in Section 7 and a summary of our results is given in Section 8.

2. Related Work

Emerging patterns, introduced by (Dong and Li, 1999; Dong and Li2, 2005), can be used to build high accuracy classification models in relational databases (Dong et al, 1999; Li et al, 2001). However it is difficult to translate the mining techniques for emerging patterns to sequential databases, since the order in which items occur in sequential data is significant and items may also occur multiple times. Contrasts for relational data have been considered in other work as well, see (Bay and Pazzani, 2001; Webb et al, 2003) for details.

In (Chan et al, 2003), the related concept of emerging substrings is introduced. These are strings of items used to differentiate between two classes of sequences. A suffix tree is used to store all the substrings. Version space trees have also been used to mine substring patterns satisfying a conjunction of constraints (Fischer and Raedt, 2004; Raedt and Kramer, 2001; Mitchell, 1982). Because substrings are a special case of subsequences using maximum gap as 0, our framework can also be used to mine minimal distinguishing substrings. However, since the items in subsequences may not necessarily appear consecutively, the use of substring data structures like suffix trees or version space trees is unsuitable for mining them. Also, the search space for subsequence patterns with gap constraints is larger and consequently the mining problem is more difficult.

An algorithm is given in (Hirao et al, 2003) to mine a single best subsequence pattern maximizing some function, which describes pattern goodness (and can describe contrasts). It does not produce a collection of patterns.

Work in (Lesh et al, 2000) examines the useful feature space for sequence databases. The algorithm used is SPADE (Zaki, 2001), relying on the Apriori property. Thus, any contrast patterns it finds must have all their subsequences being contrast as well. This assumption isn't true for MDS patterns.

References (Zaki, 2000; Méger and Rigotti, 2004) consider sequential pattern mining with gap constraints. However, these algorithms store all occurrences for a given candidate in a list, which needs to be scanned when checking the gap constraint. This idea becomes less effective in situations where the alphabet size and support thresholds are small and many long sequences need to be checked (such as in protein datasets). Gap constraints have also been considered for repetitive pattern mining with a single long sequence. Refer to (Méger and Rigotti, 2004; Casas-Garriga, 2003; Zhang et al, 2005) for details.

There exists a large body of work on finding motif patterns for protein sequences (see e.g. (Narasimhan et al, 2002)). Such patterns are related to MDSs, but are quite different and thus require different mining techniques. They also take into account various biological constraints and usually have 100% support.

Gap constraints are applied in alignment of genome or protein sequences (Gusfield, 1997). When computing the optimal alignment of two sequences, scoring functions can be adjusted to give penalties for insertions or deletions. Under this regime, alignments containing bigger gaps are less likely to be chosen. For this paper, we explicitly define the minimal and maximum gap, rather than allowing it to be automatically determined.

3. Definitions and Terminology

Let I be a set of distinct items. We call I the alphabet and $|I|$ the size of the alphabet. A sequence S over I is an ordered list of items, denoted as $e_1e_2e_3...e_n$, where $e_i \in I$ for $1 \leq i \leq n$. For example, DNA sequences are sequences over the alphabet of $\{A, C, G, T\}$, and the Declaration of Independence document is a sequence over the alphabet consisting of English words. We write $S_{[i]}$ to denote the i-th item of S, namely e_i. Note that the sequences we consider are univariate sequences, i.e. each element of the sequence is a single item. Although more general sequence definitions exist, the univariate representation is able to capture some of the most important and popular sequences, such as DNA, proteins, documents and Web-logs.

A sequence S' is a subsequence of a sequence $S = e_1e_2e_3...e_n$ (and S is a supersequence of S'), written as $S' \subseteq S$, if $S' = e_{i_1}e_{i_2}...e_{i_m}$ such that $1 \leq i_1 < i_2 < ... < i_m \leq n$. S' is a substring of S if $i_{j+1} = i_j + 1$ for all $1 \leq j < m$. For example, *AB* is a subsequence of *ACBC* but *BA* isn't, and *CBC* is a substring of *ACBC*.

Definition 3.1. (Max-Prefix) A sequence $e_1e_2e_3...e_n$'s max-prefix is $e_1e_2e_3...e_{n-1}$. The max-prefix is formed by removing the last item in S.

Example 3.1. *ABC* is the max-prefix of *ABCD* while *AB* isn't. According to our definition, a sequence has exactly one max-prefix.

Definition 3.2. (Subsequence Occurrence) Given a sequence $S = e_1e_2e_3...e_n$ and a subsequence $S' = e'_1e'_2...e'_m$ of S, a sequence of indices $\{i_1, i_2, ..., i_m\}$ is called an occurrence of S' in S if $1 \leq i_k \leq n$ and $e'_k = e_{i_k}$ for each $1 \leq k \leq m$, and $i_k < i_{k+1}$ for each $1 \leq k < m$.

Example 3.2. For the sequence *S=ACACBCB* and subsequence *S'=AB*, there are 4 occurrences of S' in S: $\{$*1,5*$\}$, $\{$*1,7*$\}$, $\{$*3,5*$\}$ and $\{$*3,7*$\}$.

We now define the gap constraints, which restrict the allowed distance between items of subsequences in sequences.

Definition 3.3. (Gap constraint and satisfaction) A (maximum) gap constraint is specified by a positive integer g. Given a sequence $S = e_1e_2...e_n$ and an occurrence $o_s = i_1i_2...i_m$ of a subsequence S', if $i_{k+1} - i_k \leq g + 1 \; \forall k \in \{1...m-1\}$, then we say the occurrence o_s fulfills the g-gap constraint. Otherwise we say o_s fails the g-gap constraint. If there is at least one occurrence of a subsequence S' fulfilling the g-gap constraint, we say S' fulfills the g-gap constraint. Otherwise S' fails the g-gap constraint. We will later consider the minimum gap constraint.

Example 3.3. In Example 3.2, only the occurrence $\{$*3,5*$\}$ fulfills the 1-gap constraint. Thus, the subsequence S' fulfills the 1-gap constraint since at least one of its occurrences does. No occurrence of S' fulfills the 0-gap constraint and so S' fails the 0-gap constraint.

Given a set of sequences D, a sequential pattern p and a gap constraint g, the count of p in D with g-gap constraint, denoted as $count_D(p, g)$, is the number of sequences in D in which p appears as a subsequence fulfilling the g-gap constraint. The (relative) support of p in D with g-gap constraint is defined as $supp_D(p, g) = \frac{count_D(p,g)}{|D|}$. Given a positive threshold δ, if $supp_D(p, g) \geq \delta$, we say p is frequent in D with g-gap constraint. Otherwise p is infrequent.

Definition 3.4. (g-MDS and the g-MDS mining problem) Given two classes of sequences pos (the positive) and neg (the negative), two support thresholds δ and α, and a maximum gap [2] g, a pattern p is called a Minimal Distinguishing Subsequence with g-gap constraint (g-MDS for short), if and only if the following conditions are true:

1. **Frequency condition:** $supp_{pos}(p, g) \geq \delta$;
2. **Infrequency condition:** $supp_{neg}(p, g) \leq \alpha$;
3. **Minimality condition:** There is no subsequence of p satisfying 1 and 2.

Given pos, neg, δ, α and g, the g-MDS mining problem is to find all the g-MDSs.

The minimality condition is important, since it both reduces output size and improves performance, as well as making patterns shorter (more succinct). This is especially important for datasets with long sequences, where the number of patterns output may be huge. Similar issues regarding concise representations arise in frequent and emerging pattern mining as well. In frequent pattern mining, mining closed or maximal patterns is

[2] We examine incorporation of a minimum gap constraint in Section 5.1. Also, in *ConSGapMiner*, the gap constraints for pos and neg do not necessarily have to be the same. In this paper, we use the same gap constraint for both, to make illustration easier. More discussion is provided in Section 5.

Table 1. A sequential database example

Sequence ID	Sequence	Class label
1	*CBAB*	pos
2	*AACCB*	pos
3	*BBAAC*	pos
4	*BCAB*	neg
5	*ABACB*	neg

popular (Yan et al, 2003; Wang and Han, 2004). In emerging pattern mining (Dong and Li, 1999; Dong and Li2, 2005), minimal and maximal borders are mined, rather than the entire space of emerging patterns.

To differentiate patterns which are guaranteed to be minimal, from those which *may* be minimal, we will use the following definition of Semi-Minimal Distinguishing Subsequence.

Definition 3.5. **(g-SMDS set)** Any super set of the g-MDS set containing patterns that satisfy the frequency and infrequency conditions, but not necessarily the minimality condition, is called a **Semi-Minimal Distinguishing Subsequence set** with g maximum gap constraint, g-SMDS set for short.

Example 3.4. Given the two sets of sequences shown in Table 1, suppose $\delta = 1/3$ (and $\alpha = 0$) and $g = 1$. The 1-MDSs are $\{$*BB, CC, BAA, CBA*$\}$. Notice that *BB* is a subsequence of all the negative sequences, if no gap constraint is used. However all the occurrences of *BB* in the negative fail the 1-gap constraint, so *BB* becomes a distinguishing subsequence when $g = 1$. Observe that every super sequence of an 1-MDS fulfilling the 1-gap constraint and support threshold is also distinguishing. However, these are excluded from the MDS set, since they are non-minimal and contain redundant information.

4. The *ConSGapMiner* Algorithm

We now introduce our algorithm known as *ConSGapMiner*, for solving the g-MDS mining problem. Extensions of this basic algorithm will be provided later in Section 5. It operates in three stages: i) candidate generation, ii) support and gap calculation, and iii) post processing (minimization). In the first stage, a candidate c is generated. In the next stage, its frequency support and gap satisfaction is computed for both the pos and neg. If $supp_{pos}(c, g) \geq \delta$ and $supp_{neg}(c, g) \leq \alpha$, then c is retained. Finally, in the third stage, post processing is used to remove all the non minimal patterns and yield the final g-MDS set. We now discuss each of these stages in turn.

4.1. Candidate generation

ConSGapMiner performs a depth-first search in a lexicographic sequence tree, similar to frequent subsequence mining techniques such as (Ayres et al, 2002; Yan et al, 2003). In the lexicographic sequence tree, each node contains a sequence s (we will interchangeably refer to nodes and the sequences they represent), a value for $count_{pos}(s, g)$ and a value for $count_{neg}(s, g)$. Each node is the max-prefix of each of its children. During the depth-first search, we extend the current node by a single item from the alphabet, according to a certain lexicographic order. For (the sequence of) each newly-generated node n, we calculate its supports from pos and from neg.

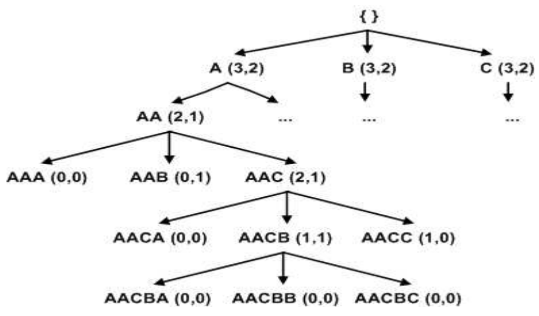


Fig. 1. The lexicographic tree.

Example 4.1. Part of the lexicographic tree for mining the database from Table 1 is given in Figure 1. Observe that the branches of the lexicographic tree terminate at nodes whose $count_{pos} = 0$.

Two basic pruning strategies can be applied to reduce the size of the search space of the tree. These will be applied in the candidate generation process.

Non-Minimal Distinguishing Pruning: This strategy is based on the fact that any supersequence of a distinguishing sequence cannot be a minimal one. Suppose we encounter a node representing sequence s, where c is the last item in s and $supp_{pos}(s, g) \geq \delta$ and $supp_{neg}(s, g) \leq \alpha$. Then i) we need never extend s and ii) need never extend any of the sibling nodes of s by the item c. Such an extension would lead to a supersequence of s and wouldn't be an MDS.

Example 4.2. In Figure 1, because $supp_{pos}(\mathit{AACC}) > 0$ and $supp_{neg}(\mathit{AACC}) = 0$, *AACC* must be distinguishing. Moreover, we know in the subtree of its sibling *AACB*, $supp_{neg}(\mathit{AACBC})$ must be 0, too. So *AACBC* can't be an MDS.

Max-Prefix Infrequency Pruning: Whenever a candidate isn't frequent in pos, then none of its descendants in the tree can be frequent. Thus, whenever we come across a node s, where $supp_{pos}(s, g) < \delta$, we don't need to extend this node any further. For example, in Figure 1, it is not necessary to extend *AAB* (which has support zero in pos), since no frequent sequence can be found in its subtree.

It is worth noting that this technique does not generalize to full a-priori like pruning – "if a subsequence is infrequent in pos, then no supersequence of it can be frequent". Such a statement is not true, because the gap constraint is not class preserved (Zaki, 2000). This means that an infrequent sequence's supersequence is not necessarily infrequent; this consequently increases the difficulty of our problem. Indeed, extending an infrequent subsequence by appending will not lead to a frequent sequence, but extensions by inserting items in the middle of the subsequence may lead to a frequent subsequence. An example situation is given next.

Example 4.3. For Figure 1, suppose $\delta = 1/3$ and $g = 1$. Then *AAB* is not a frequent pattern because $count_{pos}(\mathit{AAB}, 1) = 0$. But looking at *AAB*'s sibling, the subtree rooted at *AAC*, we see that $count_{pos}(\mathit{AACB}, 1) = 1$. So a supersequence *AACB* is frequent, but its subsequence *AAB* is infrequent.

The algorithm for candidate generation is given in Algorithm 1. Assume MDS is set to empty initially. It is called at the top level by Candidate_Gen($\{\}, g, I, \delta, \alpha$).

Algorithm 1 Candidate_Gen(c,g,I,δ,α): Generate new candidates from sequence c

```
Require: $c$:sequence, $g$:maximum gap, $I$:alphabet, $\delta$: minimal support in $pos$,
    $\alpha$:maximum support in $neg$.
Ensure: $MDS$ is a global variable containing all distinguishing subsequences generated from the entire tree.
 1: $ds = \emptyset$ {to contain all distinguishing children of $c$}
 2: for all $i \in I$ do
 3:   if $c + i$ is not a supersequence of any sequence in $ds$ then
 4:     $nc = c + i$
 5:     $supp_{pos}$=Support_Count($nc$,$g$,$pos$)
 6:     $supp_{neg}$=Support_Count($nc$,$g$,$neg$)
 7:     if $supp_{pos} \geq \delta$ AND $supp_{neg} \leq \alpha$ then
 8:       $ds = ds \cup nc$ {$nc$ is distinguishing}
 9:     else if $supp_{pos} \geq \delta$ then
10:       Candidate_Gen($nc$,$g$,$I$,$\delta$,$\alpha$)
11:     end if
12:   end if
13: end for
14: $MDS = MDS \cup ds$
```

4.2. Support Calculation and Gap Checking

For each newly-generated candidate c, $count_{pos}(c, g)$ and $count_{neg}(c, g)$ must be computed. The main challenge comes in checking satisfaction of the gap constraint. A candidate can occur many times within a single positive sequence. A straightforward idea for gap checking would be to record the occurrence of each candidate in a separate list. When extending the candidate, a scan of the list determines whether or not the extension is legal, by checking whether the gap between the end position and the item being appended is smaller than the (maximum) gap constraint value for each occurrence. This idea becomes ineffective in situations with small alphabet size and small support threshold and many long sequences needing to be checked, since the occurrence list becomes unmanageably large. Instead, we use a new method for gap checking, based on a bitset representation of subsequences and the use of boolean operations. This technique is described next.

Definition 4.1. (Bitset) A bitset is a sequence of bits which each takes the value 0 or 1. An n-bitset X contains n bits, and $X[i]$ refers to the i-th bit of X.

We use a bitset to describe how a sequence can occur within another sequence. Suppose we have a sequence $S = e_1e_2e_3...e_n$, and another sequence S', which is no longer than S. The occurrence(s) of S' in S can be represented by an n-bitset. This n-bitset BS is defined as follows: If both i) there exists a supersequence of S' of the form $e_1e_2e_3...e_i$ and ii) e_i is the final item of S', then $BS_{[i]}$ is set to 1; otherwise it is set to 0. For example, if *S=BACACBCCB*, the 9-bitset representing S' =*AB* is 000001001. This indicates how the subsequence *AB* can occur in *BACACBCCB*, with a '1' being turned on in each final position where the subsequence *AB* could be embedded. If S' isn't a subsequence of S, then the bitset representing the occurrences of S' consists of all zeros.

For the special case where S' is a single item, i.e. $S' = e$, then $BS_{[i]}$ is set to 1 if $e_i = e$. In the last example, the 9-bitset representing the single item C is 001010110.

It will be necessary to compare a given subsequence against multiple other sequences. In this case, the subsequence will have associated with it an array of bitsets, where the k-th bitset describes the occurrences of S' in the k-th sequence.

Initial Bitset Construction: Before mining begins, it is necessary to construct the bitsets that describe how each item of the alphabet occurs in each sequence from the pos and neg datasets. So, each item i has associated with it an array of $|pos| + |neg|$ bitsets. For a given item, the number of bitsets in its array which contain one or more 1's, is equal to $count(i, g)$.

Example 4.4. Consider the database in Table 1. The bitset array for A contains 5 elements, namely $[0010, 11000, 00110, 0010, 10100]$. Also, $count_{pos}(A, g) = 3$ and $count_{neg}(A, g) = 2$.

Bitset Checking: Each candidate node c in the lexicographic tree has a bitset array associated with it, which describes how the sequence for that node can occur in each of the $|pos|+|neg|$ sequences. This bitset array can be directly used to compute $count_{pos}(c, g)$ and $count_{neg}(c, g)$ (i.e. $count_{pos}(c, g)$ is just the number of bitsets in the array not equal to zero, that describe positive sequences). During mining, we extend a node c to get a new candidate c', by appending some item i. Before we can compute $count_{pos}(c', g)$ and $count_{neg}(c', g)$, we first need to compute the bitset array for c'. The bitset array for c' is calculated using the bitset array for c and the bitset array for item i, and is done in two stages.

Stage 1: Using the bitset array for c, we generate another array of corresponding mask bitsets. Each mask bitset captures all the valid extensions of c, with respect to the gap constraint, for a particular sequence in $pos \cup neg$. Suppose the maximum gap is g, for a given bitset b in the bitset array of c. We perform $g + 1$ times of right shift of it by distance 1, with 0s filling the leftmost bits. This results in $g + 1$ intermediate bitsets, one for each stage of the shift. By ORing together all the intermediate bitsets, we obtain the final mask bitset m derived from b. The mask bitset array for c consists of all such mask bitsets.

Example 4.5. Taking the last bitset 10100 in the previous example and setting $g = 1$, the process is:

$$\begin{array}{ccc} 10100 & >> & 01010 \\ 01010 & >> & 00101 \\ \hline \text{OR} & & 01111 \end{array}$$

01111 is the mask bitset derived from bitset 10100.

Intuitively, a mask bitset m generated from a bitset b, closes all 1s in b (by setting them to 0) and opens the following $g + 1$ bits (by setting them to 1). In this way, m can accept only 1s within a $g + 1$ distance from the 1s in b.

Stage 2: We use the mask bitset array for c and the bitset array for item i, to calculate the bitset array for c' which is the result of appending i to c. Consider a sequence s in $pos \cup neg$ and suppose the mask bitset describing it is m and the bitset for item i is t. The bitset describing the occurrence of c' in s, is equal to m AND t. If the bitset of the new candidate c' doesn't contain any 1, we can conclude that this candidate is not a subsequence of s with g-gap constraint.

Example 4.6. ANDing 01111 (the mask bitset for sequence A) from the last example with C's bitset 00010, gives us *AC*'s bitset 00010.

Taking the last sequence in Table 1, *ABACB*, B's 5-bitset is 01001 and its mask 5-bitset is:

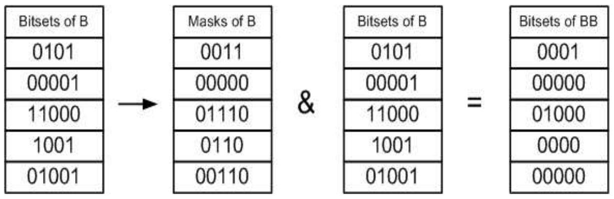


Fig. 2. The generation of *BB*'s bitset array.

$$\begin{array}{ccc} 01001 & >> & 00100 \\ 00100 & >> & 00010 \\ \hline \text{OR} & & 00110 \end{array}$$

So *BB*'s bitset is: 00110 AND 01001 $= 00000$. This means *BB* is not a subsequence of *ABACB* with 1-gap constraint.

Example 4.7. Figure 2 shows the process of getting the bitset array for *BB* from that for *B*. From the figure we can see $count_{pos}(BB, 1) = 2$ and $count_{neg}(BB, 1) = 0$.

The task of computing bitset arrays can be done very efficiently. Modern computer architectures have very fast implementations of shift operations and logical operations. Since the maximum gaps are usually small (e.g. less than 20), the total number of right shifts and logical operations needed is not too large. Consequently, calculating $supp_{pos}(c, g)$ and $supp_{neg}(c, g)$ can be done extremely quickly. The algorithm for support counting is given in Algorithm 2.

Algorithm 2 Support_Count(c',g,D): calculate $supp_D(c', g)$

```
Require: $g$:maximum gap and $BARRAY_c$: the bitset array for max-prefix $c$ of $c'$ and
    $IARRAY_i$, the bitset array for the final element $i$ of $c'$.
Ensure: : return $supp_D(c', g)$ and the bitset array for $c'$
count ← 0
for all $s \in D$ do
    $p \leftarrow BARRAY_c[s]$
    $i \leftarrow IARRAY_i[s]$
    $m = m$ XOR $m$ {bitset $m$ contains all zeros}
    $c \leftarrow 0$ {loop counter}
    repeat
        $p = (p >> 1)$
        $m = m$ OR $p$
        $c++$
    until $c = g + 1$
    $m = m$ AND $i$
    if ($m \neq 0$) then
        $count++$
    end if
    $BARRAY_{c'}[s] = m$
end for
return $count/|D|$
```

4.3. Minimization

We have already seen how non-minimal distinguishing pruning eliminates non-minimal candidates during tree expansion. However, the pattern set returned by Algorithm 1 is only semi-minimal, i.e. an SMDS set. For example, in Figure 1, we will get *ACC*, which is a supersequence of the distinguishing sequence *CC*. Thus, in order to get the g-MDS set, a post-processing minimization step is needed.

A naive idea for removing non-minimal sequences, is to check each against all the others, removing it if it is a supersequence of at least one other. For n sequences, this leads to an $O(n^2)$ algorithm, which is expensive if n is large.

Firstly, we observe that it is not necessary to check if a sequence is a supersequence of any longer sequence. So first of all, we ensure that the sequential patterns are clustered according to their length, when they are output during mining.

Secondly, we use a prefix tree for carrying out minimization. Sequences are inserted into the tree in ascending order of length. Each sequence p to be inserted into the tree is compared against the sequences already there. This is easily done by stepping through each prefix of p, at each stage identifying the nodes of the tree which are subsequences of the prefix so far. The process terminates when a leaf node or the end of p is reached. If a subsequence of p in the tree is found, then p is discarded. Otherwise, p must be minimal and it is inserted.

Compared to the naive $O(n^2)$ method, using a prefix tree can help avoid some duplicate comparisons, particularly for situations where there is substantial similarity between the sequential patterns, since each sequential pattern prefix is only stored once. For example, consider two shorter patterns P_1 =*ABCC*, P_2 =*ABCF* and a longer pattern P_3 =*ABCDE*. To check whether P_3 is minimal by using the naive way, we compare P_3 with P_1 itemwise for 5 comparisons and with P_2 itemwise for 5 comparisons to conclude that P_3 is minimal. By using the prefix tree, *ABC* is built once and compared once, which takes itemwise 3 comparisons and then another 2 comparisons to check the other two items *D* and *E*. Finally we know that P_3 is minimal, because no leaf is found. This takes itemwise 5 comparisons total, rather than 10 comparisons using the naive way.

The algorithm for minimization is shown in Algorithm 3 and the complete algorithm of *ConSGapMiner* is provided in Algorithm 4.

Algorithm 3 Minimization(ds): minimize the distinguishing subsequence set ds

```
Require: $ds$ stored in main memory
1: $pt \leftarrow NIL$ {empty the prefix tree}
2: for all $s \in ds$ in ascending order of $|s|$ do
3:     if $\exists s' \in pt$ that $s' \subseteq s$ then
4:         remove $s$ from $ds$
5:     else
6:         $pt$.add($s$)
7:     end if
8: end for
```

5. Extending the Basic ConsGapMiner Algorithm

We now examine several extensions to the basic approach just described. These involve handling minimum gaps, length constraints and performing more complex types

```
Algorithm 4 ConSGapMiner($pos$,$neg$,$g$,$\delta$,$\alpha$): mine all the $g$-MDS from $pos$ to $neg$
Require: $I$: alphabet list, $g$: maximum gap constraint, $\delta$: minimal support in $pos$, $\alpha$:
    maximal support in $neg$. A global set SMDS is used to contain the patterns gener-
    ated by Candidate_Gen
Ensure: $g$-MDS set $mds$
1: $SMDS \leftarrow \{\}$ {$ds$ contains nothing at the beginning}
2: $c \leftarrow \{\}$
3: Candidate_Gen($c$,$g$,$I$,$\delta$,$\alpha$)
4: $mds$ =Minimization($SMDS$)
5: return $mds$
```

of minimization. We also discuss the applicability of our framework for mining frequent sequential patterns.

5.1. Allowing Minimum Gap Constraints

The first extension is to add a minimum gap constraint along with the maximum one.

Definition 5.1. (Minimum Gap Constraint) A minimum gap constraint is specified by a positive integer q. Given a sequence $S = e_1e_2...e_n$ and an occurrence $o_s = (i_1, i_2, ..., i_m)$ of a subsequence S', if $i_{k+1} - i_k \geq q + 1\ \forall k \in \{1...m-1\}$, then we say the occurrence o_s fulfills the q-minimum gap constraint. Otherwise we say o_s fails the q-minimum gap constraint. If there is at least one occurrence of a subsequence S' fulfilling the q-minimum gap constraint, we say S' fulfills the q-minimum gap constraint. Otherwise S' fails it.

Example 5.1. Consider a sequence S =*ACEBE*. If a maximum gap $g = 3$ is given, *AE* is a subsequence with 2 occurrences in S. The first is $(1, 3)$ and the other is $(1, 5)$. If a minimum gap $q = 2$ is given, *AE* is a subsequence with one occurrence $((1, 5))$. Considering only the above maximum gap constraint, *AC* is a subsequence of S, but in conjunction with the above minimum gap constraint, it isn't.

The preceding definition of minimum gap constraint is quite similar to that for maximum gap. Minimum gaps can be useful for applications where we require items in a sequence to be at least certain distance apart from one another. For example, in scenarios where the items in the sequence represent values being sampled over time, such as a waveform, items that are too close to each other may represent information that is overly similar. Minimum gaps may then be specified to help remove potential redundancy in the discovered patterns. An interesting special case is when the value of the minimum gap is specified to equal the value for the maximum gap. This will result in patterns whose items are distributed in equal distance in the original dataset. This could be viewed as a kind of quasi-periodicity.

Suppose we have a set of sequences D, a sequential pattern p and two gap constraints g and q. The count of p in D with g as the maximum gap constraint and q as the minimum gap constraint, denoted as $count_D(p, g, q)$, is the number of sequences in D in which p appears as a subsequence fulfilling both the g-gap constraint and the q-minimum gap constraint. The (relative) support of p in D with g-gap constraint and q-minimum gap constraint is defined as $supp_D(p, g, q) = \frac{count_D(p,g,q)}{|D|}$.

We now redefine the mining problem to include both the minimum and maximum gap constraints:

Definition 5.2. (Extended MDS mining problem) Given two classes of sequences pos (the positive) and neg (the negative), two support thresholds δ and α, a maximum gap g and a minimum gap q, a pattern p is called a Minimal Distinguishing Subsequence with (g, q)-gap constraint ((g, q)-MDS for short), if and only if the following conditions are true:

1. **Frequency condition:** $supp_{pos}(p, g, q) \geq \delta$;
2. **Infrequency condition:** $supp_{neg}(p, g, q) \leq \alpha$;
3. **Minimality condition:** There is no subsequence of p satisfying 1 and 2.

Given pos, neg, δ, α, g and q, the (g, q)-MDS mining problem is to find all the (g, q)-MDSs.

It is easy to extend the bit operations used in *ConSGapMiner* to handle this minimum gap constraint. The only part requiring modification is the construction of the mask bitset. Recall that for a maximum gap constraint g, to construct the mask bitset, we perform $g + 1$ times of right shift by distance 1 and OR the $g + 1$ intermediate bitsets together. The resulting bitset is the mask of the given bitset. When a minimum gap constraint q is added, we initially right shift the given bitset q times and discard the intermediate bitsets. Then, we perform another $g + 1 - q$ right shift operations and OR the $g + 1 - q$ intermediate bitsets to obtain the mask bitset.

Example 5.2. Consider the last sequence in Table 1 as an example. We know that *B*'s bitset is 00111 w.r.t. $g = 2$. So *BC*'s bitset should be: 00111 AND 00010=00010. Consider $q = 2$, we discard the first 2 intermediate bitsets which are 00100 and 00010. Then we right shift $g + 1 - q = 1$ times and the mask bitset is 00001. From this bitset we can see that the adjacent two positions, which are 3 and 4, are closed and the third position 5 is open. This mask bitset expresses the minimum gap constraint. By ANDing this mask bitset with the single item C's bitset 00010, we get 00000; so *BC* isn't a subsequence of *ABACB*, because it fails the minimum gap constraint $q = 2$.

5.2. Sequence Length Constraints

The use of a maximum length constraint is a popular way of reducing the search space and size of the output for sequential pattern mining algorithms (e.g. (Zaki, 2000)). *ConSGapMiner* can be easily extended to mine patterns with lengths less than a given threshold l. Here, by length we mean the number of items that appear in the pattern (as distinct from window size, which refers to the maximum gap between the first and last item in the pattern). We supplement the existing pruning strategies with another called max-length pruning.

Max-Length Pruning: Whenever a candidate whose length is equal to l is generated, then it is never extended.

By keeping a counter of the depth of the tree within each node, it is then straightforward to determine the current length of each candidate for performing the comparison. Using length constraint pruning is likely to result in a much shallower lexicographic tree and thus faster mining time.

5.3. Coverage and Prefix-Based Pattern Minimization

In order to discover patterns which satisfy the minimality condition from Definitions 3.4 and 5.2, we have described strategies for pattern minimization which aim to determine whether a pattern (or candidate) is a supersequence of some other pattern (or candidate). For some situations, pursuing this kind strategy may be too aggressive and useful patterns may be eliminated, since the gap constraint is not class preserved. Consider the following example:

Example 5.3. Suppose *pos*=$\{$*ACCBD*, *ACDBD*, *ABD*$\}$ and two distinguishing patterns *ACBD* and *ABD* have been found using $\delta = 1/2$, $g = 1$, $q = 0$. We observe that $count_{pos}(ACBD, 1, 0) = 2$ and $count_{pos}(ABD) = 1$. If we remove *ACBD* because it is a super-sequence of *ABD*, we will lose a pattern which has higher frequency than its subpattern and thus is arguably a more important feature.

To address this problem, we now describe two alternate minimization techniques. The first is based on comparisons using coverage in pos, the second is based on prefix comparisons. Both are less aggressive and remove fewer patterns than the minimization we have previously described (which will be called *basic minimization* from now on).

In essence, we need to to use stricter conditions for minimality than that used in Definitions 3.4 and 5.2. Given two MDSs, p_1 and p_2 and a reference dataset D, we wish to remove p_2 due to p_1, if p_1 occurs in every sequence from D that p_2 does ($p_2 \Rightarrow_D p_1$) . If p_1 occurs in every sequence from pos that p_2 does, then it is guaranteed that ($p_2 \Rightarrow_D p_1$) only if $D = pos$. This is *coverage based minimization*. If p_1 is prefix of p_2, then it is guaranteed that $p_2 \Rightarrow_D p_1$, with respect to any D. This is *prefix based minimization*.

5.3.1. Coverage Based Minimization

To describe the implementation of coverage based minimization, we first begin by formally defining the notion of coverage.

Definition 5.3. (Coverage Set) Given a sequence p, gap parameters g and q and the datasets *pos* and *neg*, the coverage set of p is equal to the set of sequences from pos in which p appears as a subsequence fulfilling both the $g-$gap constraint and the q-minimum gap constraint. The coverage set can be represented by a bitset, containing as many bits as there are sequences in pos. Bit i is turned on if p appears as a subsequence in sequence i from pos. Otherwise it is set to zero.

When performing coverage based minimization, a sequence p_1 can be eliminated by a sequence p_2 iff:

1. $p_2 \subseteq p_1$ and
2. the coverage set of p_1 is a subset of the coverage set of p_2.

To adjust the basic *ConSGapMiner* algorithm from Section 4 we have to change two techniques, the first is the non-minimal distinguishing pruning step and the other is the post-processing minimization.

For each candidate in the lexicographic tree shown in Figure 1, a coverage bitset is attached. This coverage bitset contains as many bits as the total of sequences in pos. A newly-generated candidate's coverage bitset can be set by its bitset array. If the i-th bitset in the array contains at least one 1, this candidate's $count_{pos}$ is increased and the i-th bit in the coverage bitset is set to 1. The rule for the pruning is then changed to:

Non-Minimal Distinguishing Pruning (adjusted): Suppose we encounter a node representing sequence s, where c is the last item in s and $supp_{pos}(s,g) \geq \delta$ and $supp_{neg}(s,g) \leq \alpha$. Then i) we need never extend s and ii) for any of the sibling nodes s', we AND s''s coverage bitset with s's coverage bitset and then XOR the resulting bitset with s''s coverage bitset. If the resulting bitset doesn't contain any 1, then we never need extend s' by the item c. If the resulting bitset contains at least one 1, we must extend s' by the item c.

Boolean operations can be used to test whether s's coverage is a superset of s''s. If this is the case, then no candidate in the subtree of s' has a coverage which is not a subset of s's. In this situation, any extension of the nodes in the subtree of s', with item c can give a super-sequence of s with a subset of s's coverage, which means it isn't minimal. If s' has a non-subset coverage of s, then there may be an extension with item c, which gives a distinguishing subsequence with a non-subset coverage of s's and this may be minimal, so we need to generate and keep it.

For the post-processing minimization, we still order the patterns in descending order of their lengths. For each pattern we still keep a coverage bitset with the same meaning as described above. For each pattern p, we find all the patterns which have shorter lengths and with a coverage that is a superset of that of p. If such a pattern is found, the standard way of checking whether this pattern is a subsequence of p or not is performed. If it is a subset, p is eliminated. If no pattern can be found to eliminate p, then we retain it in the MDS set.

5.3.2. Prefix Based Minimization

Performing prefix based minimization is substantially simpler than performing coverage based minimization. Two modifications are needed to the basic ConsSGapMiner algorithm. First of all, the non-minimal distinguishing pruning step is adjusted to the following:

Non-Minimal Distinguishing Pruning (adjusted): Suppose we encounter a node representing sequence s, where c is the last item in s and $supp_{pos}(s,g) \geq \delta$ and $supp_{neg}(s,g) \leq \alpha$. Then we need never extend s.

Secondly, no post processing minimization step is needed, since all distinguishing sequences produced are guaranteed to be prefix minimal.

5.3.3. Comparison of the Three Minimization Techniques

We now compare the three minimization techniques we have described: basic minimization, prefix based minimization and coverage minimization. We compare them in terms of output size, running time and classification capability.

Suppose we denote the MDS set obtained using each of these strategies as MDS, MDS_{pre} and MDS_{cov}. Observe that $MDS \subseteq MDS_{cov} \subseteq MDS_{pre}$, hence $|MDS| \leq |MDS_{cov}| \leq |MDS_{pre}|$. In terms of running time, the search space for mining MDS will be the smallest and the search space to mine MDS_{pre} will be the largest. With regard to classification ability, we would expect this to be correlated with the number of patterns that are contained in an unknown test sequence (according to the gap constraints). Thus, MDS_{pre} would have the most useful classification capability, since it i) includes all the patterns that MDS and MDS_{cov} do, ii) may include some patterns which do not have a subsequence that both exists in MDS or MDS_{cov} and is contained in the test sequence. This means MDS_{pre} will contain the most patterns which are contained in an unknown test sequence during classification. In a similar way, we would expect MDS to have the least powerful classification capability.

5.4. Applicability of the Approach to Frequent Sequential Patterns

Frequent sequential pattern mining with gap constraints has been identified as a useful task (Zaki, 2000). *ConSGapMiner* can be adapted to mine this kind of pattern, too. For this scenario, support only needs to be checked with reference to a single dataset. The method for checking maximum and/or minimum gap constraints remains the same. The main difference is that only max-prefix infrequency pruning can be used. The non-minimal distinguishing pruning and post processing minimization techniques are no longer needed, since a succinct representation is not required.

6. Performance Study

We have evaluated the performance of *ConSGapMiner* in a number of ways. Section 6.1 focuses on mining a special kind of MDS which has zero support in the *neg* dataset. This kind of pattern is similar to the jumping emerging pattern (Dong and Li, 1999; Dong and Li2, 2005). It captures very sharp contrasts between the datasets. Section 6.2 then examines the mining of patterns with non-zero support in the *neg*. In Section 6.3 we then show the distribution of the MDSs and look at the effect of using a maximum length constraint. We use a minimum gap $q = 0$ for all experiments.

No comparison is made against other systems, since we are not aware of any other work that is suitable for mining g-MDSs. We use two kinds of sequences, one from protein families and the other from books of the Bible. These two sequence types represent some interesting real-world applications. On the one hand, protein families use a relatively small alphabet (20 amino acids), each containing relatively few sequences with long average length. On the other hand, books of the Bible are built on a large alphabet (several thousand words), and have thousands of sentences of small average length. The protein families were selected from PFam: Protein Family Database (`http://www.sanger.ac.uk/Software/Pfam/`) and the Bible books were downloaded from `http://www.o-bible.com/dlb.html`. All the experiments were run on a 3.0GHz Intel Xeon PC,with 4 gigabytes of main memory, running UNIX.

We now give some more specific details on the datasets. (1) The pairs of protein families that we used are listed in Table 2. These represent some challenging situations and the dataset sizes are representative for protein families. (2) We extracted sequences from the books of the Bible. This kind of sequential data differs from protein data, due to its large alphabet size, much smaller sequence length and larger number of sequences. We used all sentences in the first four books of the New Testament (Matthew, Mark, Luke and John) as the positive class and all sentences in the first four books of the Old Testament (Genesis, Exodus, Leviticus and Numbers) as the negative class. In order to obtain meaningful patterns, we removed all the punctuation and frequently appearing words such as ”and”, ”the”, ”of”. Each sentence corresponds to a separate sequence. There are 3768 sequences in pos, 4893 sequences in neg, and a total (alphabet size) of 3344 unique words. Average sentence length is 7 words and the maximum is 23.

Table 2. Protein family pairs.

Pair Id	Pos	#sequences	Neg	#sequence	avg. len. (*pos*)	avg. len. (*neg*)
1	DUF1694	16	DUF1695	5	123	186
2	CbiA	80	CbiX	76	195	106
3	SrfB	5	Spheroidin	4	1025	932
4	TatC	74	TatD_DNase	119	205	262

Table 3. Some 6**-MDSs from the Bible Books**

substrings (support)	subsequences (support)
unclean spirit (13)	seated right (10)
eternal life (24)	seated hand (10)
good news (23)	answer truly (10)
forgiveness sin (22)	full wonder (24)
chief priests (53)	kingdom heaven (33)

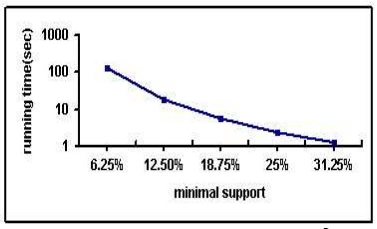
6.1. Mining MDSs with zero support in neg

Protein Families: In Figure 3, we give the running time for varying frequency thresholds (refer to a) and (maximum) gap size (refer to b). α is set to 0 for all cases. We can see that as the maximum gap becomes larger, or as the frequency threshold δ becomes lower, more time is required for mining. An important reason for this is that the MDS output size increases dramatically in both situations. For example, consider the final pair of protein families in Table 2. When $g = 5$ and $\delta = 24.3\%$, there are 20936 5-MDSs output. Changing δ to 5.4%, the output size jumps to 3600822. For the same dataset with $\delta = 27\%$ and $g = 3$, the output size is 536, whereas for $\delta = 27\%$ and $g = 7$, it is 314791. Some explanations for these are the following: The smaller the maximum gap is, the earlier a candidate is likely to become distinguishing (being less likely to appear in the neg) and so earlier pruning of the search space is possible. Similarly, earlier pruning is possible for high values of the frequency constraint δ, since it is more difficult to satisfy for longer (and thus lower) sequence nodes in the tree. Furthermore, the longer sequences in the pos and neg are, the more time *ConSGapMiner* needs, since it has to search to deeper levels in the lexicographic tree

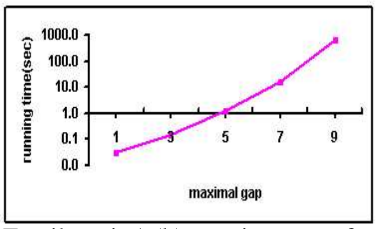
Books of the Bible: Experimental results concerning the Bible books are shown in Figure 3. Looking at these figures, we can see that *ConSGapMiner* operates much faster on this kind of data. The larger alphabet means that non-minimal distinguishing pruning happens very early in the lexicographic tree, while the small average length means the tree cannot become too deep. Table 3 lists some of the patterns returned when mining the 6-MDS. Both contiguous patterns (substrings) and non-contiguous patterns (subsequences) are shown, with the number of times they occur. Obviously, for human understanding of the patterns, the meaning of the substrings is more straightforward than subsequences. However, subsequence contrasts can sometimes capture combinations of interesting words that are not found by substrings.

6.2. Mining MDSs with non-zero support in negative

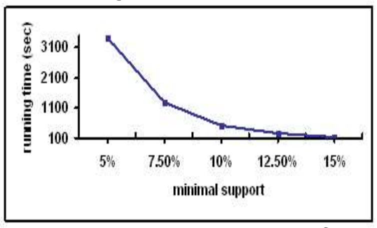
Figure 4 shows the experiments of the four protein family pairs with varying α, fixed δ and g. *ConSGapMiner* runs faster when α is bigger. This is because, as α increases, it becomes easier for a candidate pattern to become distinguishing, because it is easier for its support in *neg* to be below α. When this happens, the subtree of the candidate pattern



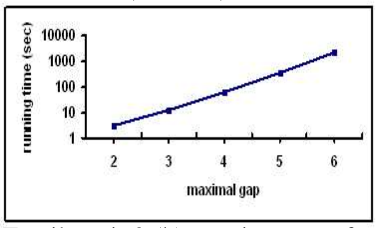
Family pair 1 (a): runtime vs δ, for $g = 5, \alpha = 0$.



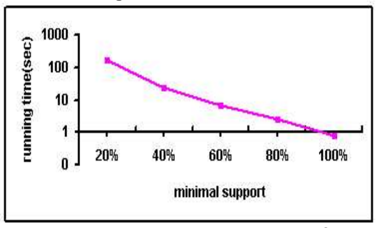
Family pair 1 (b): runtime vs g, for $\delta = 5(31.25\%), \alpha = 0$.



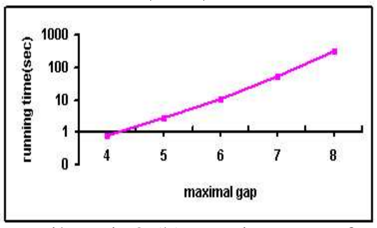
Family pair 2 (a): runtime vs δ, for $g = 5, \alpha = 0$.



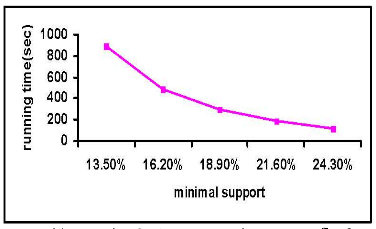
Family pair 2 (b): runtime vs g, for $\delta = 8(10\%), \alpha = 0$.



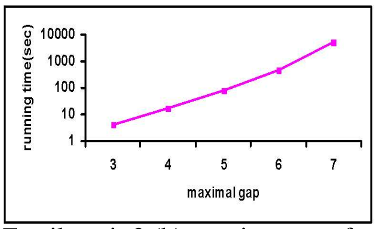
Family pair 2 (a): runtime vs δ, for $g = 4, \alpha = 0$.



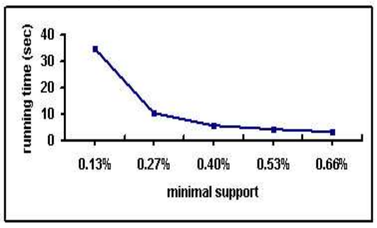
Family pair 2 (b): runtime vs g, for $\delta = 5(100\%), \alpha = 0$.



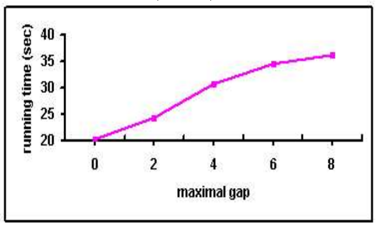
Family pair 3 (a): runtime vs δ, for $g = 5, \alpha = 0$.



Family pair 3 (b): runtime vs g, for $\delta = 20(27\%), \alpha = 0$.

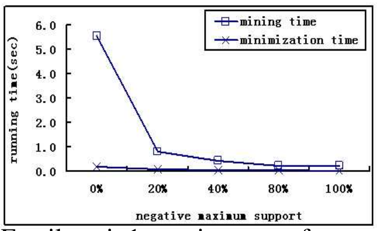


Bible: runtime vs δ for $g = 6$.

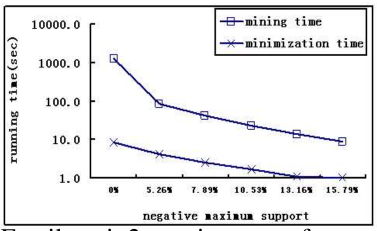


Bible: runtime vs g for $\delta = 0.13\%$.

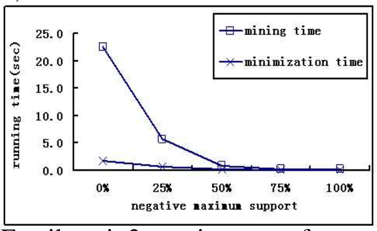
Fig. 3. Zero α experiments for protein families and Bible books.



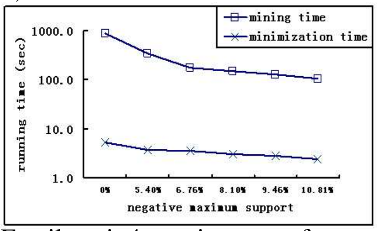
Family pair 1: runtime vs α, for $g = 5$, $\delta = 18.75\%$.



Family pair 2: runtime vs α, for $g = 5$, $\delta = 7.5\%$.



Family pair 3: runtime vs α, for $g = 5$, $\delta = 40\%$.



Family pair 4: runtime vs α, for $g = 5$, $\delta = 13.5\%$.

Fig. 4. Non-zero α experiments for protein families.

can be pruned and the search space becomes smaller. But as α increases, the contrast information carried by the mined patterns may be weaker because potentially more negative sequences can match them as well. From Figure 4 we can also see the time used by minimization procedure. This post-processing step only takes a small proportion of the total running time.

6.3. Distributions of the MDS patterns and effect of the length constraint

We also examined the distribution of the MDSs mined from the four protein family pairs. From Figure 6 we can see that the number of patterns drops while the support increases. The patterns with high support in *pos* and low (even 0) support in *neg* are considered to capture the most significant contrast information and considered to be more valuable.

We now look at the patterns by considering their median lengths. Candidates with smaller lengths are more easily found in *neg*; so a pattern needs to be reasonably long to capture enough information to distinguish *pos* from *neg*. Also the minimization post-process removes a lot of long patterns because some shorter subsequences are already contained.

The speedups that result from using a maximum length constraint for the second pair of the protein families in 2 are now shown. From Figure 5 we can see that as the maximum length goes down, the mining process achieves a considerable speedup, due to the depth of the lexicographic tree being no more than the maximum length parameter.

From a classification point of view, the best patterns are those with high support in *pos*, low support in *neg* and small lengths. These patterns capture strong contrast information and are more likely to be contained in an unknown (test) sequence and contribute to the identification of the most likely class (in a classification scenario). But as we can see from the diagrams, these "useful" patterns are relatively few. In order

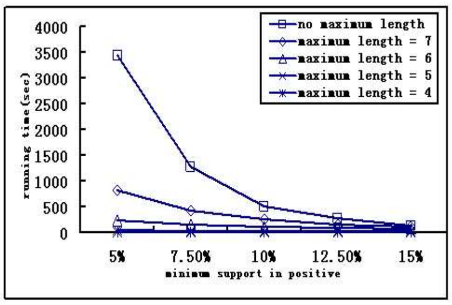


Fig. 5. Varying the maximum length constraint ($g = 5$, $\alpha = 0$).

to get enough "useful" patterns, parameters need to be carefully set. The relationship between parameter settings and the accuracy of classifiers is likely to depend on specific dataset properties, but is an interesting topic for future work.

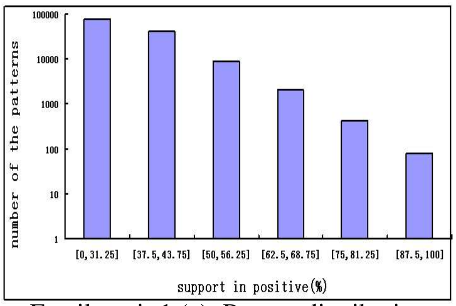
7. Discussion and Future Work

The results in the previous section are only a snapshot of the experiments we performed. We also tested *ConSGapMiner* on a number of other protein datasets, with overall performance being similar and pleasing overall. We now look at the complexity of the algorithm, discuss some limitations and consider future work.

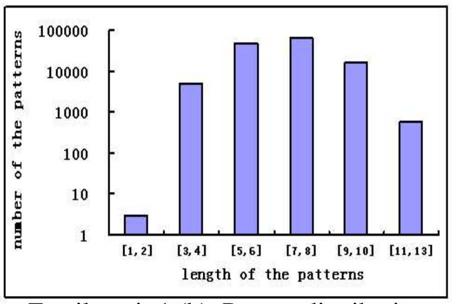
7.1. Complexity analysis

We analyze the time and space complexity of ConSGapMiner. Let L be the average length of the sequences; A be the size of the alphabet; N be the total number of the sequences in *pos* and *neg*.

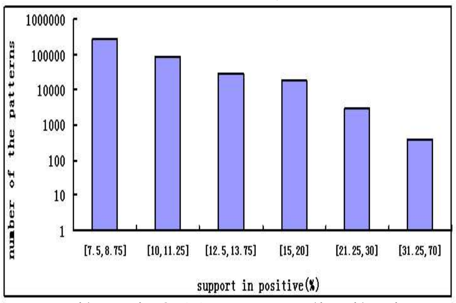
Space complexity: Each bitset takes $L/8$ bytes to store and each bitset array takes up $(L * N)/8$ bytes. Initially all the single-item bitset arrays use up to $(A * L * N)/8$ bytes. During the candidate generation, a depth-first search is performed in the lexicographic tree. Suppose the longest pattern's length is l, then the upper bound of the memory usage is (without any pruning of the search space): $(l * A * L * N)/8$ bytes. At runtime, the deeper the candidate is located in the tree, the more branch pruning is possible so the actual usage of memory is always less than $(l * A * L * N)/8$. For the minimization, suppose the average length of the patterns is l_{avg} and each item in the sequence can be stored in 1 byte (that means A is less than 256 which is sufficient in most bio-sequence mining problems) and the total number of minimal distinguishing subsequences is T_{pat}. In the prefix tree, each node contains 1 byte for the item and $4A$ bytes for the children links (suppose the machine is 32bit one). The upper bound memory used to store the tree is $l_{avg} * T_{pat} * (4A + 1)$ considering no sharing of the prefix. So the memory usage by *ConSGapMiner* is $max(l * A * L * N)/8, l_{avg} * T_{pat} * (4A + 1)$. Suppose we have 1GB of memory, a reasonable assumption for today's hardware; suppose A is 20, which is the case for amino acids in proteins; and suppose the longest pattern contains no more than 20 amino acids and that the average length of the MDSs is 10. Then *ConSGapMiner* can handle a dataset having an average length L of 1000 and the number of sequences N of 20000, within $(20 * 20 * 1000 * 20000)/8 = 953MB$, assuming an



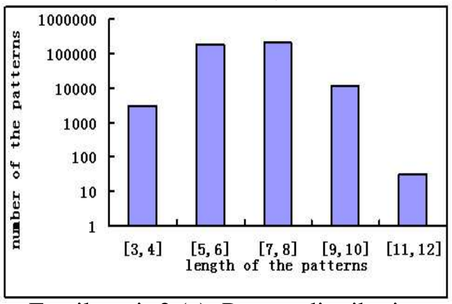
Family pair 1 (a): Pattern distribution according to supports ($g = 9$, $\delta = 31.25\%$ and $\alpha = 0$).



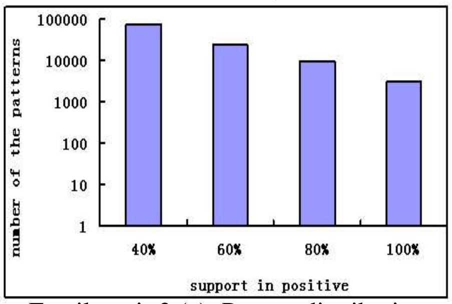
Family pair 1 (b): Pattern distribution according to lengths ($g = 9$, $\delta = 31.25\%$ and $\alpha = 0$).



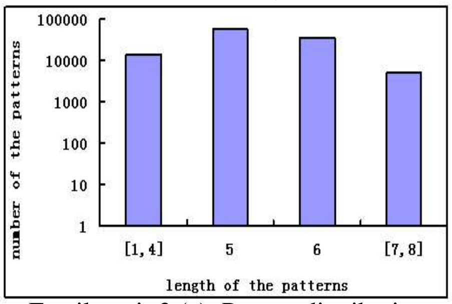
Family pair 2 (a): Pattern distribution according to supports ($g = 5$, $\delta = 7.5\%$ and $\alpha = 0$).



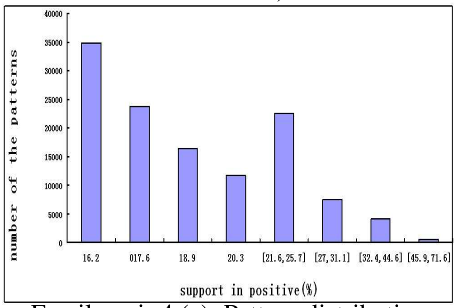
Family pair 2 (a): Pattern distribution according to lengths ($g = 5$, $\delta = 7.5\%$ and $\alpha = 0$).



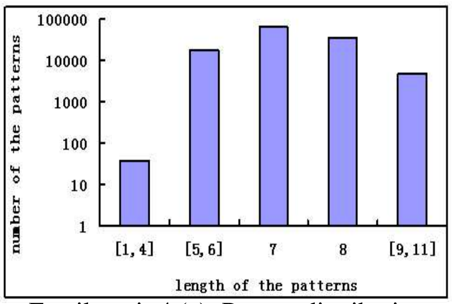
Family pair 3 (a): Pattern distribution according to supports ($g = 4$, $\delta = 40\%$ and $\alpha = 0$).



Family pair 3 (a): Pattern distribution according to lengths ($g = 4$, $\delta = 40\%$ and $\alpha = 0$).



Family pair 4 (a): Pattern distribution according to supports ($g = 5$, $\delta = 16.20\%$ and $\alpha = 0$).



Family pair 4 (a): Pattern distribution according to supports ($g = 5$, $\delta = 16.20\%$ and $\alpha = 0$).

Fig. 6. Pattern distributions

output size 1000000 of MDSs. *ConSGapMiner* is thus likely to be practical for many bioinformatic applications.

Time complexity Suppose N_{exp} nodes in the lexicographic tree are explored by *ConSGapMiner* and the maximum gap constraint is g. A pre-condition of the following analysis is that byte-wise bit operations can be finished in constant time and checking for whether a bitset contains 1 or not can also be done in constant time. For each bitset, $g+1$ right shifts and ORing the intermediate bitsets require $2(g+1)L/8$ time to finish. The last AND operation takes another $L/8$ time. So for each node, $2*N*(g+2)*L/8+N$ time is used to generate the bitset array and determine $Count_{pos}$ and $Count_{neg}$. For the total N_{exp} nodes, the time used to mine all the MDSs is roughly $N_{exp}*(2*N*(g+2)*L/8+N)$. For short protein segments with maximum length no more than 64 or 32, all bit operations can be finished in constant time and so*ConSGapMiner* will be extremely fast.

7.2. Limitations

We now discuss some of the limitations of *ConSGapMiner*. Firstly, for very large maximum gap constraint, the shift operation of the bitsets may become costly. Secondly, for extremely large datasets with many sequences longer than 10K, there may be insufficient main memory to use *ConSGapMiner*. From this point of view, it may be worthwhile to examine schemes for bitset compression. Finally, the parameters of δ, α, g and q all have to be chosen by the user in order to give useful patterns. For users who lack knowledge of the structure of the datasets, this may be a very difficult task. Hence, it may be worthwhile to instead focus on “parameter free” mining (Tzvetkov et al, 2003) mining the top k most distinguishing sequences.

7.3. Window Size Constraint

As mentioned earlier, the number of MDS patterns present for high dimensional datasets can be very large. Gap size is certainly an important way of reducing this output size. A window size constraint (limiting the maximum gap between the first and last item in an MDS) appears to be more difficult to deal with and to require the maintenance of more involved data than bitsets.

7.4. Suitability for Classification

The focus of this paper has been presenting an efficient algorithm for mining MDS patterns. Of course, there is the important related question of how such patterns may be used. We believe these patterns are interesting, due to their intuitive, human understandable form and ability to capture strong contrasts. Such contrast patterns can be used directly by humans, and they can be used to build accurate classifiers. We have carried out some preliminary experiments (not included in this paper) which indicate that a simple classifier model built from these patterns is able to give very promising predictive power for determining the correct protein family of an unknown sequence, an important research topic in bioinformatics.

Evaluating a range of classifiers built using MDSs to select an optimal one is an interesting direction. One possibility is to form a high dimension feature space with the

MDSs and then use an SVM to predict in this space dimension. This kind of direction has been followed in (She et al, 2003), although only frequent substrings were used.

7.5. Using Different Gaps for pos and neg Datasets

As mentioned earlier, *ConSGapMiner* can be easily modified to mine patterns using a gap constraint for the neg dataset which is different than that used for the pos dataset. For example, mine all patterns using $g = 0$ in pos and $g = 20$ in neg. This will yield MDSs which appear as substrings in pos and which have a very strict definition of non appearance in neg - they must not appear as a substring, or as any subsequence with gap less than 20. These can be thought of as being contrasts that are much sharper than those that would be obtained if $g = 0$ were used for both datasets.

8. Concluding Remarks

We have introduced the data mining problem of *minimal distinguishing subsequences*. These patterns can capture essential contrast information between different classes of sequences.

Algorithmically, we studied the efficient mining of minimal distinguishing subsequences and made the following major contributions: (a) A prefix growth framework for mining MDSs, utilizing a number of pruning techniques. (b) A bitset-manipulation based technique for checking gap constraints. Analysis and experiments show that our approach works well for a number of datasets, particularly high dimensional proteins. (c) Some extensions of our algorithm show that it is flexible and can be extended easily to deal with the mining of some other pattern types. (d) A thorough study of the performance of the *ConSGapMiner* and the properties of the MDSs.

Acknowledgements: This work was partially supported by National ICT Australia. National ICT Australia is funded by the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

Agrawel R and Srikant R(1995) Mining sequential patterns. In *Proceedings of ICDE*, pp 3–14

Antunes C and Antunes AL(2003) Generalization of Pattern-Growth Methods for Sequential Pattern Mining with Gap Constraints. In *Proceedings of MLDM*, pp 239–251

Ayres J, Flannick J, Gehrke J, Yiu T(2002) Sequential pattern mining using a bitmap representation. In *Proceedings of KDD*, pp 429–435

Bailey J, Manoukian T, Ramamohanarao K(2003) Classification using constrained emerging patterns. In *Proceedings of WAIM*, pp 226–237

Bay SD and Pazzani MJ(2001) Detecting group differences: Mining contrast sets. *Data Mining and Knowledge Discovery*

Casas-Garriga G(2003) Discovering unbounded episodes in sequential data. In *Proceedings of PKDD*, pp 83–94

Chan S, Kao B, Yip CL, Tang M(2003) Mining emerging substrings. In *Proceedings of DASFAA*, pp 119–126

Das G, Fleischer R, Gasieniec L, Gunopulos D, Kärkkäinen J(1997) Episode matching. In *Proceedings of CPM*, pp 12–27

Dong G and Li J(1999) Efficient mining of emerging patterns: Discovering trends and differences. In *Proceedings of KDD*, pp 43–52

Dong G and Li J(2005) Mining Border Descriptions of Emerging Patterns from Dataset Pairs. In *Knowledge and Information Systems* 8(2), 178–202.

Dong G, Zhang X, Wong L, Li J(1999) CAEP: Classification by Aggregating Emerging Patterns. In *Proceedings of Int'l Conf. on Discovery Science*, pp 30–42

Fischer J and Raedt LD(2004) Towards Optimizing Conjunctive Inductive Queries. In *Proceedings of PAKDD*, pp 625–637

Gusfield D(1997) Algorithms on strings, trees and sequences, computer science and computational biology. *Cambridge*

Han J, Pei J, Mortazavi-Asl B, Chen Q, Dayal U, Hsu M(2000) Freespan: frequent pattern-projected sequential pattern mining. In *Proceedings of KDD*, pp 355–359

Hirao M, Hoshino H, Shinohara A, Takeda M, Arikawa S(2003) A practical algorithm to find the best subsequence patterns. *Theor. Comput. Sci.*, 292(2):465–479

Ji X, Bailey J, Dong D(2005) Mining minimal distinguishing subsequence patterns with gap constraints. In *Proceedings of ICDM*, pp 194–201

Lesh N, Zaki MJ, Ogihara M(2000) Scalable feature mining for sequential data. *IEEE Intelligent Systems*, 15(2):48–56

Li J, Dong G, Ramamohanarao K(2001) Making use of the most expressive jumping emerging patterns for classification. *Knowl. and Inf. Syst.*, 3(2):131–145

Maier D(1978) The complexity of some problems on subsequences and supersequences. *Journal of ACM*, 25(2):322–336

Mannila H, Toivonen H, Verkamo AI(1995) Discovering frequent episodes in sequences. In *Proceedings of KDD*, pp 210–215

Méger N, Rigotti C(2004) Constraint-based mining of episode rules and optimal window sizes. In *Proceedings of PKDD*, pp 313–324

Mitchell TM(1982) Generalization as Search. *Journal of Atrifical Intelligence*, 18(2):203–226

Narasimhan G, Bu C, Gao Y, Wang X, Xu N, Mathee K(2002) Mining protein sequences for motifs. *Journal of Computational Biology*, 9(5):707–720

Pei J, Han J, Mortazavi-Asl B, Pinto H, Chen Q, Dayal U, Hsu M(2001) Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proceedings of ICDE*, pp 215–224

Raedit LD, Kramer S(2001) The Levelwise Version Space Algorithm and its Application to Molecular Fragment Finding. In *Proceedings of IJCAI*, pp 853–862

She R, Chen F, Wang K, Ester M, Gardy JL, Brinkman F(2003) Frequent-subsequence-based prediction of outer membrane proteins. In *Proceedings of KDD*, pp 436–445

Tronícek Z(2001) Episode matching. In *Proceedings of CPM*, pp 143–146

Tzvetkov P, Yan X, Han J(2003) TSP: Mining Top-K Closed Sequential Patterns. In *Proceedings of ICDM*, pp 347–354

Wang J, Han J(2004) BIDE: Efficient mining of frequent closed sequences. In *Proceedings of ICDE*, pp 79–90

Webb GI, Butler S, Newlands D(2003) On detecting differences between groups. In *Proceedings of KDD*, pp 256–265

Yan X, Han J, Afshar R(2003) Clospan: Mining closed sequential patterns in large databases. In *Proceedings of SDM*

Zhang M, Kao B, Cheung D, Yip K(2005) Mining Periodic Patterns with Gap Requirement from Sequences. In *Proceedings of SIGMOD*

Zaki MJ(2000) Sequence mining in categorical domains: Incorporating constraints. In *Proceedings of CIKM*, pp 422–429

Zaki MJ(2001) Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1/2):31–60

Author Biographies

Xiaonan Ji received a B.E. degree from San Yat-sun University, Guang Dong, China, in 2003. He is currently a Phd student at the Department of Computer Science and Software Engineering, University of Melbourne, Australia. His research interests include data mining especially sequence data mining with application in bioinformatics.

James Bailey received the Ph.D. degree from University of Melbourne in 1998. He is currently a Senior Lecturer at the Department of Computer Science and Software Engineering, University of Melbourne, Australia. He has also worked at both Kings College and Birkbeck College, University of London. His main research interests are in data mining, agent systems and XML technologies and he has over 50 scientific publications. He has served on numerous international program committees and was program co-chair of the Australasian Database Conference in 2006 and 2007.

Guozhu Dong received the Ph.D. degree from the University of Southern California in 1988. He is currently an associate professor at Wright State University. He also taught at the University of Melbourne and Flinders University, and was a visiting scientist at Lucent Bell Labs, KRDL (I2R) Singapore, RIKEN Japan, UCSB, Simon Fraser U, U of Georgia, and UIUC. His main research interests are in the areas of databases, knowledge bases, data mining, and bioinformatics. He has over 90 scientific publications and 3 US patents. He is a senior member of IEEE and a member of ACM. He has served on program committees of numerous major database and data mining conferences, including IEEE ICDE, IEEE ICDM, ICDT, ACM KDD, ACM PODS, VLDB, etc. He was a Program Committee co-chair of WAIM 2003. He has served on the international editorial board of International Journal of Information Technology.

Correspondence and offprint requests to: James Bailey, Department of Computer Science and Software Engineering, University of Melbourne, Victoria 3053, Australia. Email: jbailey@csse.unimelb.edu.au