Mining Minimal Distinguishing Subsequence Patterns with Gap Constraints

Xiaonan Ji* James Bailey* Guozhu Dong†

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 maximum gap constraint. We present an efficient algorithm called ConSGapMiner, to mine all MDSs according to a maximum gap 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.*

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 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 [4], 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.

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

†Department of Computer Science and Engineering, Wright State University, USA

There are many situations where MDSs can be applied, 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 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 [12], where it is observed that biologists are very interested in identifying 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 *maximum 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 MDSs. The first is that the Apriori property does not hold for distinguishing subsequences (unlike it does for frequent subsequences), meaning that the subsequences of a distinguishing sequence are not necessarily distinguishing themselves. 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 thresholds may need to be set to at least 80% [13]. Using the same thresholds for MDS mining is likely to result in empty output. In MDS mining, thresholds usually 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 [9, 3]. 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 3 for a brief explanation) [16] and the search space is potentially larger in MDS mining, the position 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 (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.

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.

Related Work: Emerging patterns, introduced by [5], can be used to build high accuracy classification models in relational databases ([8]). 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 [2] and [14] for details.

In [4], 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. 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 a suffix tree is unsuitable for mining them. Also, the search space is larger and consequently the mining problem is more difficult.

An algorithm is given in [6] to mine a single best subsequence pattern maximising some function, which describes pattern goodness (and could describe a contrast). It does not enumerate a collection patterns.

Work in [7] examines the useful feature space for sequence databases. The algorithm used is SPADE [17], 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 [16] and [9] consider sequential pattern mining with gap constraints. However, their algorithm stores 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).

There exists a large body of work on finding motif patterns for protein sequences (see e.g. [10]). Such patterns are related to MDSs, but are far more general and thus much more difficult to mine. They also take into account various biological constraints and usually have 100% support.

Organisation: Section 2 introduces the basic concepts used and Section 3 describes the *ConSGapMiner* algorithm. Experimental results are given in Section 4, followed by discussion in Section 5 and conclusion in Section 6.

2. Problem definition

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 2.1 ***(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 2.1 *ABC is the max-prefix of ABCD while AB isn't. According to our definition, a sequence has exactly one max-prefix.*

Definition 2.2 ***(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 2.2 *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 2.3 ***(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.*

Example 2.3 *In Example 2.2, 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 2.4 ***(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 [1] 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 very important, because it both reduces output size and improves performance, as well as making patterns more succinct.

Similar to JEPs ([8]) we will place special emphasis on those g-MDSs satisfying $\alpha = 0$ (never appearing in the negative class). In the experimental section, we focus on the g-MDS mining problem with $\alpha = 0$. Our techniques are of course applicable for any value of α.

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*

Example 2.4 *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.*

3. The *ConSGapMiner* Algorithm

We now introduce our algorithm known as *ConSGapMiner*, for solving the g-MDS mining problem. It operates in three stages. 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 answers and yield the final g-MDS set. We now discuss each of these stages in turn.

3.1. Candidate generation

ConSGapMiner performs a depth-first search in a lexicographic sequence tree, similar to frequent subsequence

[1] 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.

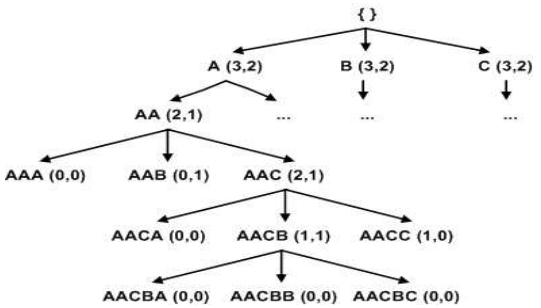


Figure 1. The lexicographic tree.

mining techniques such as [1, 15, 11]. 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*.

Example 3.1 *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 3.2 *In Figure 1, because* $supp_{neg}(AACC) = 0$*, AACC must be distinguishing and we know in the subtree of its sibling AACB,* $supp_{neg}(AACBC)$ *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 generalise 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 [16]. This means that an infrequent sequence's supersequence is not always necessarily infrequent and 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.

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

Require: c:sequence, g:maximum gap, I:alphabet, δ: minimal support in *pos*, α:maximum support in *neg*.
Ensure: *DS* is a global variable containing all candidate distinguishing subsequences generated from the tree.

```
1: ds = ∅ {to contain all distinguishing children of c}
2: for all i ∈ 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 ≥ δ AND supp_neg ≤ α then
8:       ds = ds∪nc {nc is distinguishing}
9:     else if supp_pos ≥ δ then
10:      Candidate_Gen(nc,g,I,δ,α)
11:    end if
12:  end if
13: end for
14: DS=DS ∪ ds
```

Example 3.3 *For Figure 1, suppose* $\delta = 1/3$ *and* $g = 1$*. Then AAB is not a frequent pattern because* $count_{pos}(AAB,1) = 0$*. But looking at AAB's sibling, the subtree rooted at AAC, we see that* $count_{pos}(AACB,1) = 1$*. So here, a supersequence (AACB) is frequent, but its subsequence (AAB) is infrequent.*

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

3.2. 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 occurrences 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 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 3.1 ***(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 $e_1e_2e_3...e_i$ ($i \leq n$) 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 3.4 *Consider the database in Table 1. A's array of bitsets contains 5 elements and is* $[0010, 11000, 00110, 0010, 10100]$. *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 3.5 *Taking the last bitset* 10100 *in the previous example and setting* $g = 1$*, the process is:*

10100	>>	01010
01010	>>	00101
OR		01111

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 3.6 *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:*

01001	>>	00100
00100	>>	00010
OR		00110

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

Example 3.7 *Figure 2 shows the process of getting the bitset array BB from 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

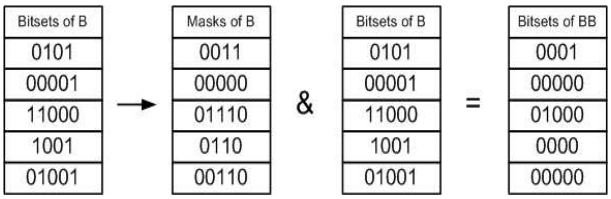


Figure 2. The generation of *BB*'s bitset array ($g = 1$)

Procedure 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 item i of c'.

Ensure: : return $supp_D(c', g)$ and the bitset array for c'

```
1: count ← 0
2: for all s ∈ D do
3:    p ← BARRAY_c[s]
4:    i ← IARRAY_i[s]
5:    m = m XOR m {bitset m contains all zeros}
6:    c ← 0 {loop counter}
7:    repeat
8:       p = (p >> 1)
9:       m = m OR p
10:      c++
11:   until c = g + 1
12:   m = m AND i
13:   if (m ≠ 0) then
14:      count++
15:   end if
16:   BARRAY_c'[s] = m
17: end for
18: return count/|D|
```

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 Procedure 2.

3.3. Minimization

The patterns returned by Procedure 1 are not necessarily all minimal. 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, post-processing minimization is needed.

A naive idea for removing non-minimal sequences from a set, is to check each one 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. We improve on this basic idea by making use of two properties of non-minimal sequences.

Theorem 3.1 *Let S and S' be two distinguishing sequences returned by Procedure 1. If S' is a subsequence of S, then*

Table 2. Sizes of protein families.

Id	*Pos*(num)	*Neg*(num)	Avg. Len.(*Pos*,*Neg*)
1	DUF1694(16)	DUF1695(5)	(123,186)
2	SrfB(5)	Spheroidin(4)	(1025,932)
3	TatC(74)	TatD_DNase(119)	(205,262)

Property 1 $|S'| \leq |S|$ *and*

Property 2 *S and S' must share the same final item.*

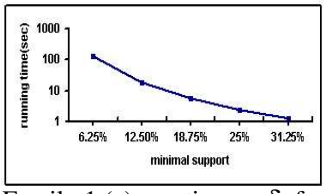
The proof is omitted here for space considerations and will be provided in a companion full paper.

Property 1 means that it is not necessary to check if a sequence is a superset of any longer sequence. This property is in fact true for any pair of subsequences, not just the distinguishing ones returned by Procedure 1. Property 2 means that each sequence need only be compared with those which share a common final item. This property is only true for the sequences returned by Procedure 1. It isn't true for arbitrary sequences. The actual minimization is performed as follows. We use the well-known prefix tree structure. For each item i in the alphabet, a prefix tree pt_i is built. Sequences having final item i are inserted into pt_i, in order of length. At each insertion, the sequence being inserted is checked to see whether it is a supersequence of any sequence in the prefix tree and discarded if this is the case.

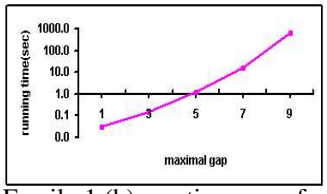
4. Performance Study

In this section, we study the performance of *ConSGapMiner*, as well as analysing some of the properties of the g-MDSs that we mine. No comparison is made against other systems, since we are not aware of any other work that is suitable for mining g-MDSs. A number of experiments on both protein families and Bible books have been carried out. 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`. In all experiments, α (the maximum frequency in threshold for *neg*) was set to zero and the experiments were run on a 3.0GHz Intel Xeon PC, with 4 gigabytes of main memory, running UNIX.

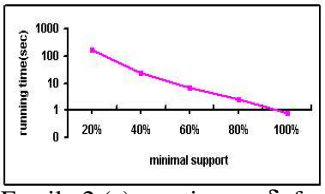
Protein Families: The protein families that we used are listed in Table 2. These represent some challenging situa-



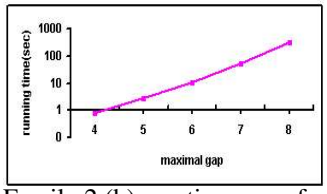
Family 1 (a): runtime vs δ, for $g = 5$, $\alpha = 0$.



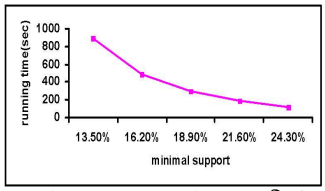
Family 1 (b): runtime vs g, for $\delta = 5(31.25\%)$, $\alpha = 0$.



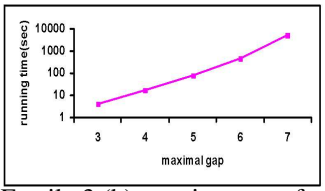
Family 2 (a): runtime vs δ, for $g = 4$, $\alpha = 0$.



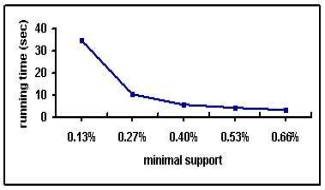
Family 2 (b): runtime vs g, for $\delta = 5(100\%)$, $\alpha = 0$.



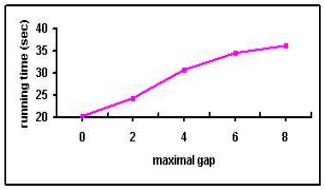
Family 3 (a): runtime vs δ, for $g = 5$, $\alpha = 0$.



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



Bible: runtime vs δ for $g = 6$.



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

Figure 3. Experimental results.

tions and their sizes are representative for protein families. In Figure 3, we give the running time for varying frequency thresholds(refer to a) and (maximum) gap size (refer to b). 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. e.g. Take 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. 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.

We also examined the distribution of the 5-MDS for the TatC vs. TatD_DNase. They are approximately normally distributed around a mean length (i.e. the number of items in a pattern) of $7-8$. There were some patterns with lengths 11, despite the gap size being 5. This reflects the ability of g-MDS to capture long patterns. Indeed for these length 11 patterns, it would be permitted for there to be an occurrence where the first item's position and last item's position are separated by distance of as much as $(11-1)*5 = 50$.

Books of the Bible: We conducted experiments using sentences from the books of the Bible as sequences. 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 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 25. The Experimental results 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.

Effect of Pruning: We have seen the effect of varying the parameters g and δ. We also conducted a number of other experiments (not shown due to lack of space) to evaluate the power of the different pruning techniques. As expected, the Max-Prefix infrequency pruning gives substantial savings, similar to using the frequency constraint for frequent subsequence mining. By employing the non-minimal distinguishing pruning strategy, *ConSGapMiner* usually gains a speedup of factor two and the pattern size

Table 3. Some 6-MDSs from the Bible Experiment

substrings(support)	subsequences(support)
unclean spirit(13)	cakes fishes(10)
eternal life(24)	seated hand(10)
good news(23)	answer truly(10)
forgiveness sin(22)	question saying(13)
chief priests(53)	truly kingdom(12)

before minimization shrinks by a factor of four. The time taken for the minimization post-processing is insignificant.

5. 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.

The performance of *ConSGapMiner* is very pleasing overall. However, as mentioned, 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, but it would be useful to employ other constraints as well. Using a length constraint (restricting the maximum number of items in an MDS) is straightforward within our framework. Using a window size constraint (limiting the maximum gap between the first and last item in an MDS) is more difficult. Employing bitset operations to maintain this constraint requires much more information to be maintained for each node in the lexicographic tree.

This paper has focused on presenting an efficient algorithm for mining g-MDS patterns. There is also the 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. Similar to emerging patterns ([8]), we believe g-MDSs will be useful for building classifiers. Studying classification may also allow examination of the tradeoffs between choosing an appropriate *pos* frequency threshold δ and gap size, versus the quality of the g-MDS patterns that are mined.

6. Concluding Remarks

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

We studied the efficient mining of minimal distinguishing subsequences and made the following major contributions: (a) A prefix growth framework for mining g-MDSs, utilising a number of pruning techniques. (b) A bitset operation based technique for checking gap constraints. Analysis and experiments show that our approach works well for a number of datasets, particularly high dimensional proteins.

Acknowledgements: This work is 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

[1] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *KDD*, pages 429–435, 2002.

[2] S. D. Bay and M. J. Pazzani. Detecting group differences: Mining contrast sets. *Data Mining and Knowledge Discovery*, 2001.

[3] Gemma Casas-Garriga. Discovering unbounded episodes in sequential data. In *PKDD*, pages 83–94, 2003.

[4] Sarah Chan, Ben Kao, Chi Lap Yip, and Michael Tang. Mining emerging substrings. In *DASFAA*, pages 119–, 2003.

[5] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *KDD*, pages 43–52, 1999.

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

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

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

[9] Nicolas Méger and Christophe Rigotti. Constraint-based mining of episode rules and optimal window sizes. In *PKDD*, pages 313–324, 2004.

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

[11] J. Pei, J. Han, B. Mortazavi-Asl, H. Wang, J. Pinto, Q. Chen, U. Dayal, and M. Hsu. Mining sequential patterns by pattern-growth: The prefixspan approach. *IEEE TKDE*, 16(10), 2004.

[12] R. She, F. Chen, K. Wang, M. Ester, J. L. Gardy, and F. Brinkman. Frequent-subsequence-based prediction of outer membrane proteins. In *KDD*, pages 436–445, 2003.

[13] Jianyong Wang and Jiawei Han. Bide: Efficient mining of frequent closed sequences. In *ICDE*, pages 79–90, 2004.

[14] G. I. Webb, S. Butler, and D. Newlands. On detecting differences between groups. In *Proceedings of KDD*, pages 256–265, 2003.

[15] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large databases. In *SDM*, 2003.

[16] M. Zaki. Sequence mining in categorical domains: Incorporating constraints. In *CIKM*, pages 422–429, 2000.

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