Sequence Mining in Categorical Domains: Incorporating Constraints

Mohammed J. Zaki
Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY 12180
zaki@cs.rpi.edu, http://www.cs.rpi.edu/~zaki

ABSTRACT

We present cSPADE, an efficient algorithm for mining frequent sequences considering a variety of syntactic constraints. These take the form of length or width limitations on the sequences, minimum or maximum gap constraints on consecutive sequence elements, applying a time window on allowable sequences, incorporating item constraints, and finding sequences predictive of one or more classes, even rare ones. Our method is efficient and scalable. Experiments on a number of synthetic and real databases show the utility and performance of considering such constraints on the set of mined sequences.

1. INTRODUCTION

This paper focuses on sequence data in which each example is represented as a sequence of "events", where each event might be described by a set of predicates, i.e., we are dealing with categorical sequential domains. Examples of sequence data include text, DNA sequences, web usage data, multi-player games, plan execution traces, and so on. The sequence mining task is to discover a sequence of attributes, shared across time among a large number of objects in a given database. For example, consider a web access database at a popular site, where an object is a web user and an attribute is a web page. The discovered patterns are the sequences of most frequently accessed pages at that site. This kind of information can be used to restructure the web-site, or to dynamically insert relevant links in web pages based on user access patterns. There are many other domains where sequence mining has been applied, which include discovering customer buying patterns in retail stores, identifying plan failures [12], finding network alarms [3], and so on.

The task of discovering all frequent sequences in large databases is quite challenging. The search space is extremely large. For example, with m attributes there are $O(m^k)$ potentially frequent sequences of length at most k. Many techniques have been proposed to mine temporal databases for the frequently occurring sequences. However, an unconstrained search can produce millions of rules or may even be intractable in some domains. Furthermore, in many domains, the user may be interested in interactively adding certain syntactic constraints on the mined sequences. For example the user may be interested in only those sequences that occur close together, those that occur far apart, those that occur within a specified time frame, those that contain specific items or those that predict a given attribute.

Mining sequences with such constraints has not received wide attention. In this paper we preset a new algorithm, called cSPADE, for discovering the set of all frequent sequences with the following constraints: 1) length and width restrictions, 2) minimum gap between sequence elements, 3) maximum gap between sequence elements, 4) a time window of occurrence of the whole sequence, 5) item constraints for including or excluding certain items, and forming superitems, and finally 6) finding sequences distinctive of at least one *class*, i.e., a special attribute-value pair, that we are interested in predicting. The approach that we take is that each of these constraints is fully integrated inside the mining process, with no post-processing step. We present experimental results on a number of synthetic and real datasets to show the effectiveness and performance of our approach. We would like to point out that even though the constraints we look at are not very complicated, they are representative of a much broader class of constraints, and the approach we develop can be applied to them without much change. Furthermore, the performance it delivers, combined with the ease with which it incorporates these constraints should be seen as a very attractive feature of cSPADE.

Related Work The problem of mining sequences has been studied in [1, 10, 6, 8, 11]. However, very little work has been done in constrained sequence mining. The GSP algorithm [10] was the first to consider minimum and maximum gaps, as well as time windows. GSP is an iterative algorithm; it counts candidate frequent sequences of length k in the k-th database scan. Special data structures like Hash Trees are used to speed up the frequency computation step. GSP requires as many full data scans as the longest frequent sequence. The problem of mining *generalized episodes* appeared in [6]. An episode is essentially a frequent sequence, but instead of being frequent across many input-sequences, an episode is frequent within one (long) sequence. The generalization consists of allowing one to express arbitrary unary conditions on individual episode events, or binary conditions on event pairs. An incremental and interactive sequence mining approach was proposed by us in [9]. The idea there was to cache previously mined results, which can then be used to answer user queries a lot faster than re-mining the entire dataset each time. Here our goal is to push the constraints inside the mining step itself.

Previous work has addressed constrained mining for association rules . A mine-and-examine paradigm for interactive

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2000 ACM 0-89791-88-6/97/05 ..$5.00

exploration of associations and sequence episodes was presented in [4]. The idea is to mine and produce a large collection of frequent patterns. The user can then explore this collection by the use of *templates* specifying what's interesting and what's not. They only consider inclusive and exclusive templates. A second approach to exploratory analysis is to integrate the constraint checking inside the mining algorithm. Recently [7] presented the CAP algorithm for extracting all frequent associations matching a rich class of constraints. However, the temporal nature of sequences introduces new kinds of constraints not considered by these works. Also, sequences may destroy the succinctness and *anti-monotone* properties of associations, i.e., when one incorporates constraints, we cannot guarantee that all subsequences will be frequent. Thus some of the insights from constrained associations [7] cannot be used for sequence mining.

The GSP method serves as a base for comparison against cSPADE. Another relevant work is the SPIRIT family of algorithms [2] for mining sequences that match user-specified regular-expression constraints. They present four methods which differ in the extent to which they push the constraints inside the mining method. The most relaxed is SPIRIT(N) which eliminates those items that do not appear in any RE. A post-processing step is then required to extract the exact answer. The most strict is SPIRIT(R), which applies the constraints while mining, and only outputs the exact set. It should be noted that if one specifies the most general RE (to contain any ordering among all items), then the SPIRIT(N/L) variants default to the GSP algorithm.

As such SPIRIT is complementary to our methods since we consider other kinds of constraints than regular-expressions. For example, a novel constraint we introduce in this paper is that of finding sequences predictive of at least one class for temporal classification problems. Second, as we shall show cSPADE (which is based on SPADE [11]) outperforms GSP anywhere from a factor of 2 to more than an order of magnitude. Since SPIRIT(N/L) is essentially the same as GSP, we expect cSPADE to outperform them as well. Additionally, the method we developed independently to deal with non-anti-monotone constraints is similar to SPIRIT(V/R). Once again we expect cSPADE to outperform them. cSPADE is also simple to implement, requiring no special internal data-structures, unlike many other approaches. Thus, the excellent performance and ease of incorporating constraints should be seen as the two great strengths of cSPADE.

2. SEQUENCE MINING

The problem of mining sequential patterns can be stated as follows: Let $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ be a set of m distinct *items* comprising the alphabet. An *event* is a non-empty unordered collection of items (without loss of generality, we assume that items of an event are sorted in lexicographic order). We also call an event an *itemset*. A *sequence* is an ordered list of events. An event is denoted as $(i_1 i_2 \cdots i_k)$, where i_j is an item. A sequence α is denoted as $(\alpha_1 \rightarrow \alpha_2 \rightarrow \cdots \rightarrow \alpha_q)$, where α_i is an event. The *length* of α is q and its *width* is the maximum size of any α_i for $1 \leq i \leq q$. A sequence with k items ($k = \sum_j |\alpha_j|$) is called a *k-sequence*. For example, $(B \rightarrow AC)$ is a 3-sequence.

For a sequence α, if the event α_i occurs before α_j, we denote it as $\alpha_i < \alpha_j$. We say α is a *subsequence* of another sequence β, denoted as $\alpha \preceq \beta$, if there exists a one-to-one order-preserving function f that maps events in α to events in β, that is, 1) $\alpha_i \subseteq f(\alpha_i)$, and 2) if $\alpha_i < \alpha_j$ then $f(\alpha_i) < f(\alpha_j)$. For example the sequence $(B \rightarrow AC)$ is a subsequence of $(AB \rightarrow E \rightarrow ACD)$, since $B \subseteq AB$ and $AC \subseteq ACD$, and the order of events is preserved. For $k \geq 3$, the *generating subsequences* of a length k sequence are the two length $k-1$ subsequences of α obtained by dropping exactly one of its first or second items. By definition, the generating sequences share a common suffix of length $k-2$. For example, the two generating subsequences of $AB \rightarrow CD \rightarrow E$ are $A \rightarrow CD \rightarrow E$ and $B \rightarrow CD \rightarrow E$, and they share the common suffix $CD \rightarrow E$.

The database $\mathcal{D}$ for sequence mining consists of a collection of input-sequences. Each input-sequence in the database has an unique identifier called *sid*, and each event in an input-sequence also has a unique identifier called *eid*. We assume that no sequence has more than one event with the same time-stamp, so that we can use the time-stamp as eid.

An input-sequence $\mathcal{C}$ is said to *contain* another sequence α, if $\alpha \preceq \mathcal{C}$, i.e., if α is a subsequence of the input-sequence $\mathcal{C}$. The *support* or *frequency* of a sequence, denoted $\sigma(\alpha, \mathcal{D})$, is the total number of input-sequences in the database $\mathcal{D}$ that contain α. Given a user-specified threshold called the *minimum support* (*min_sup*), we say that a sequence is *frequent* if it occurs more than *min_sup* times. The set of frequent k-sequences is denoted as $\mathcal{F}_k$. A frequent sequence is *maximal* if it is not a subsequence of any other frequent sequence.

Given a database $\mathcal{D}$ of input-sequences and *min_sup*, the problem of mining sequential patterns is to find all frequent sequences in the database. For example, consider the database shown in Figure 1. It has three items (A, B, C), four input-sequences, and twelve events in all. The figure also shows all the frequent sequences with a minimum support of 75% (i.e., 3 out of 4 input-sequences). The maximal sequences are $A \rightarrow A$, $B \rightarrow A$, and $AB \rightarrow B$.

Our problem formulation is quite general since: 1) We discover sequences of *subsets* of items, and not just single item sequences. For example, the set AB in $(AB \rightarrow B)$. 2) We discover sequences with arbitrary *gaps* among events, and not just the consecutive subsequences. For example, the sequence $(AB \rightarrow B)$ is a subsequence of input-sequence 4, even though there is an intervening event A. The sequence symbol $\rightarrow$ simply denotes a *happens-after* relationship. 3) Our formulation is general enough to encompass almost any categorical sequential domain. For example, if the input-sequences are DNA strings, then an event consists of a single item (one of A, C, G, T), and the *eid* is the position rather than time. If input-sequences represent text documents, then each word (along with any other attributes of that word, e.g., noun, position, etc.) would comprise an event. Even continuous domains can be represented after a suitable discretization step.

3. THE SPADE ALGORITHM

In this section we describe SPADE [11], an algorithm for fast discovery of frequent sequences, which forms the basis for our constraint algorithm.

Sequence Lattice: SPADE uses the observation that the subsequence relation $\preceq$ defines a partial order on the set of sequences, i.e., if β is a frequent sequence, then all subsequences $\alpha \preceq \beta$ are also frequent. The algorithm systematically searches the sequence lattice spanned by the subsequence relation, from the most general (single items) to the most specific frequent sequences (maximal sequences) in a depth-first (or breadth-first) manner. For instance, Figure 1 shows the lattice for the example dataset.

Support Counting: Most of the current sequence mining algorithms [10] assume a *horizontal* database layout such as the one shown in Figure 1A. In the horizontal format, the database consists of a set of input-sequences of events (i.e.,

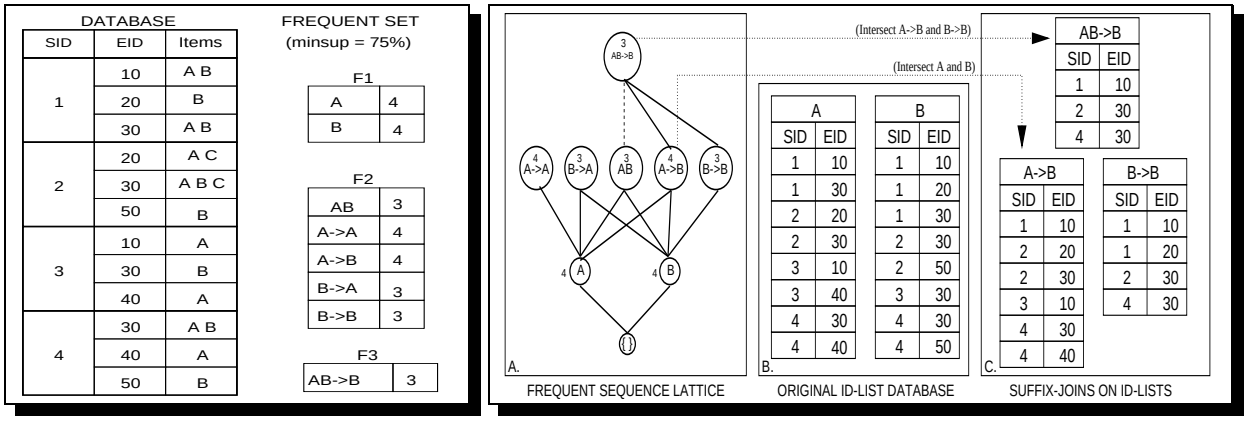

Figure 1: A) Original Database, B) Frequent Sequence Lattice, and Temporal Joins

item sets). In contrast, we use a *vertical* database layout, where we associate with each item X in the sequence lattice its *idlist*, denoted $\mathcal{L}(X)$, which is a list of all input-sequence (*sid*) and event identifier (*eid*) pairs containing the item. For example, the idlist for the item C in the original database (Figure 1) would consist of the tuples $\{\langle 2, 20\rangle, \langle 2, 30\rangle\}$. Given the per item idlists, we can iteratively determine the support of any k-sequence by performing a temporal join on the idlists of its two generating sequences (i.e., its $(k-1)$ length subsequences that share a common suffix). A simple check on the support (i.e., the number of distinct sids) of the resulting idlist tells us whether the new sequence is frequent or not. Figure 1B shows the initial vertical database with the idlist for each item. The intermediate idlist for $A \rightarrow B$ is obtained by a temporal join on the lists of A and B. Since the symbol $\rightarrow$ represents a temporal relationship, we find all occurrences of A before a B in an input-sequence, and store the corresponding time-stamps or eids, to obtain $\mathcal{L}(A \rightarrow B)$. We obtain the idlist for $AB \rightarrow B$ by joining the idlist of its two generating sequences, $A \rightarrow B$ and $B \rightarrow B$, but this time we are looking for *equality join*, i.e., instances where A and B co-occur before a B.

If we had enough main-memory, we could enumerate all the frequent sequences by traversing the lattice, and performing joins to obtain sequence supports. In practice, however, we only have a limited amount of main-memory, and all the intermediate idlists will not fit in memory. SPADE breaks up this large search space into small, manageable chunks that can be processed *independently* in memory. This is accomplished via *suffix-based equivalence classes* (henceforth denoted as a class). We say that two k length sequences are in the same class if they share a common $k-1$ length suffix. The key observation is that each class is a sub-lattice of the original lattice and can be processed independently. Each suffix class is independent in the sense that it has complete information for generating all frequent sequences that share the same suffix, i.e., the generating subsequences for any sequence must also be in the same class. SPADE recursively decomposes the sequences at each new level into even smaller independent classes. For instance, at level one it uses suffix classes of length one (X,Y), at level two it uses suffix classes of length two (X $\rightarrow$ Y, XY) and so on. We refer to level one suffix classes as *parent* classes. These suffix classes are processed one-by-one using depth-first search. Figure 2 shows the pseudo-code (simplified for exposition, see [11] for exact details) for the main procedure of the SPADE algorithm. The input to the procedure `Enumerate-Frequent` is a suffix class, along with the idlist for each of its elements. Frequent sequences are generated by joining [11] the idlists of all distinct pairs of sequences in each class and checking the support of the resulting idlist against *min_sup*. The sequences found frequent at the current level form classes for the next level. This process is recursively repeated until all frequent sequences have been enumerated. It is easy to see that we need memory to store intermediate idlists for classes along one path in the search lattice. Classes are deleted on return from each recursive call.

SPADE (min_sup):
1. $\mathcal{P} = \{$ parent classes $P_i\}$;
2. **for** each parent class $P_i \in \mathcal{P}$ **do** Enumerate-Frequent(P_i);

Enumerate-Frequent(S):
1. **for** all sequences $A_i \in S$ **do**
2. **for** all sequences $A_j \in S$, with $j > i$ **do**
3. $\mathcal{L}(R)$ = Temporal-Join($\mathcal{L}(A_i), \mathcal{L}(A_j)$);
4. **if** ($\sigma(R) \geq$ min_sup) **then** $T = T \cup \{R\}$; print R;
5. Enumerate-Frequent(T);
6. delete S;

Figure 2: Pseudo-code for SPADE

Temporal Joins: We now describe how we perform the temporal idlist joins for two sequences. Note that support is incremented only once per input-sequence. Given a suffix equivalence class $[S]$, it can contain two kinds of elements: an itemset of the form XS or a sequence of the form $Y \rightarrow S$, where X and Y are items, and S is some (suffix) sequence. Let's assume without loss of generality that the itemsets of a class always precede its sequences. To extend the class for the next level it is sufficient to join the idlists of all pairs of elements. However, depending on the pairs being joined, there can be up to three possible resulting frequent sequences: 1) *Itemset vs Itemset*: If we are joining XS with YS, then we get a new itemset XYS. 2) *Itemset vs Sequence*: If we are joining XS with $Y \rightarrow S$, then the only possible outcome is new sequence $Y \rightarrow XS$. 3) *Sequence vs Sequence*: If we are joining $X \rightarrow S$ with $Y \rightarrow S$, then there are three possible outcomes: a new itemset $XY \rightarrow S$, and two new sequences $X \rightarrow Y \rightarrow S$ and $Y \rightarrow X \rightarrow S$. A special case arises when we join $X \rightarrow S$ with itself, which can only produce the new sequence $X \rightarrow X \rightarrow S$.

Consider the idlist for the items A and B shown in Figure 1 B). These are sequence elements $A \rightarrow \emptyset$ and $B \rightarrow \emptyset$ for the class $[\emptyset]$. To get the idlist for the resultant itemset AB, we need to check for *equality* of sid-eid pairs. In our example, $\mathcal{L}(AB) = \{\langle 1, 10\rangle, \langle 1, 30\rangle, \langle 2, 20\rangle, \langle 4, 30\rangle\}$. It is frequent at 75% minimum support level (i.e., 3 out of 4 input-sequences). To compute the idlist for the sequence $A \rightarrow B$, we need to check for a *follows* temporal relationship, i.e., for a given pair (c, t_1) in $\mathcal{L}(A)$, we check whether there exists a pair (c, t_2) in $\mathcal{L}(B)$ with the same sid c, but with $t_2 > t_1$. If this is true, it means that the item B follows the item A for sequence c, and we add (c, t_1) to the

idlist of $A \rightarrow B$. The final idlist for $A \rightarrow B$ is shown in Figure 1 B). We call $A \rightarrow B$ the *forward* follows join. The idlist of $B \rightarrow A$ is obtained by reversing the roles of A and B. We call $B \rightarrow A$ the *reverse* follows join. As a further optimization, we generate the idlists of the three possible new sequences in just one join.

At first thought, one might think that we need to maintain more information for the temporal join. For example, how come the idlist $\mathcal{L}(A \rightarrow B)$ is not $\{\langle 1, 10, 20\rangle, \langle 1, 10, 30\rangle, \langle 2, 20, 30\rangle, \langle 2, 20, 50\rangle, \langle 2, 30, 50\rangle, \langle 3, 10, 30\rangle, \langle 4, 30, 50\rangle, \langle 4, 40, 50\rangle\}$. If we consult the original database in Figure 1A, we find that these are exactly the time-stamps or eids where A is followed by B for a given *sid*. However, since we know that all members of a class share the same suffix (and consequently the same eids for the suffix), it is sufficient to keep only the time-stamp for the first item. For example, consider the temporal join $\mathcal{L}(A \rightarrow B \rightarrow B)$ obtained from $\mathcal{L}(A \rightarrow B)$ and $\mathcal{L}(B \rightarrow B)$. All that needs to be done is to find instances where A precedes B, and one can simply ignore the common suffix B. In this case the resulting idlist turns out to be $\{\langle 1, 10\rangle, \langle 2, 20\rangle\}$. One can verify, by looking at Figure 1, that there are only two instances where $A \rightarrow B \rightarrow B$ occur in the data, once in $sid = 1$ with the starting $eid = 10$, and once in $sid = 2$ with the starting $eid = 20$.

4. INCORPORATING CONSTRAINTS

Here we look at different syntactic constraints on the mined sequences, which include length and width restrictions, minimum gap, maximum gap, total time window of validity of the sequence, item constraints, and enumerating sequences predictive of a given class among a set of class values.

DEFINITION 1. *We say that a constraint is* class-preserving *if in the presence of the constraint a suffix-class retains it's self-containment property, i.e., support of any k-sequence can be found by joining the idlists of its two generating subsequences of length* $(k-1)$ *within the same class.*

Whether a constraint is class-preserving or not has a direct effect on the cost of incorporating it into cSPADE. We shall see that only maximum gap is not class-preserving. It thus requires a different method of enumeration, which can be costly, since it needs more global information. All other constraints are class-preserving, and thus the frequent sequences can be listed using local suffix class information only. The class-preserving property is a new way of specifying whether a constraint is anti-monotone or not.

Handling Length and Width Restrictions: In many practical domains one has to restrict the maximum allowable length or width of a pattern just to make the task tractable, since otherwise one gets a combinatorial blowup in the number of frequent sequences. This is especially true for highly structured data, where the item frequency is relatively high. Incorporating restrictions on the length and width of the sequences is straightforward. In Figure 2, at line 3, we simply add a check to see if $width(R) <= max_w$ and if $length(R) <= max_l$, where max_w and max_l are the user-specified restrictions on the maximum allowable width and length of a sequence, respectively. Length and width have no effect on the idlists, and thus they are class-preserving.

Handling Minimum Gaps: In many domains one would like to see sequences that occur after some given interval. For example, finding all DNA subsequences that occur more than 20 positions apart, which can be useful for identifying non-local patterns. Minimum gaps can be incorporated into cSPADE without much trouble. The first observation is to note that minimum gap is a class-preserving constraint. Let $X \rightarrow S$, and $Y \rightarrow S$ be two sequences in the class $[S]$. Now if, say $X \rightarrow Y \rightarrow S$, is frequent with min_gap at least δ, then clearly Y and S must be δ apart, and also X and S must be δ or more apart. In other words one can determine if $X \rightarrow Y \rightarrow S$ by joining the idlists of $X \rightarrow S$, and $Y \rightarrow S$.

Since each suffix class remains self-contained, the only thing that needs changing is adding a *min_gap* check in the join operation. Assume that we would like to compute $\mathcal{L}(A \rightarrow B)$ from the generating items A and B in Figure 1. Let's assume that min_gap is 20. Given a pair $\langle c, t_a\rangle$ in $\mathcal{L}(A)$, we check if there exists a pair $\langle c, t_b\rangle$ in $\mathcal{L}(B)$ such that $t_b \neq t_a$ and $t_b - t_a \geq min\_gap$. If so we add the pair $\langle c, t_a\rangle$ to the idlist of $A \rightarrow B$. For example, the pair $\langle c = 1, t_a = 10\rangle$ from A's idlist is added to $\mathcal{L}(A \rightarrow B)$, since there exists the pair $\langle c = 1, t_b = 30\rangle$ in B's idlist, with $t_b - t_a = 30 - 10 = 20 \geq min\_gap$. After doing this for all sids we get $\mathcal{L}(A \rightarrow B) = \{\langle 1, 10\rangle, \langle 2, 20\rangle, \langle 2, 30\rangle, \langle 3, 10\rangle, \langle 4, 30\rangle\}$. Minimum gaps have no effect on the equality join.

Handling Maximum Gaps: Unlike minimum gap, maximum gap is not class-preserving. For example, let max_gap be δ, and let $X \rightarrow S$, and $Y \rightarrow S$ be two sequences in the class $[S]$. Let $X \rightarrow Y \rightarrow S$ be frequent with max_gap=δ. Clearly, Y and S must be at most δ apart, thus $Y \rightarrow S$ is frequent. On the other hand, we cannot claim the same thing for $X \rightarrow S$. All we can claim is that X and S are at most 2δ apart, and it may happen that there is no instance where X and S are within δ distance. In other words, $X \rightarrow S$ may be infrequent with max_gap=δ, yet $X \rightarrow Y \rightarrow S$ can be frequent. Unfortunately, this destroys the equivalence class self-containment. It requires a different process for enumerating the candidate frequent sequences. As such *max_gap* represents many constraints that are non-class-preserving, i.e., the new enumeration method we develop should work for many other constraints that destroys the class self-containment.

The first change required is that we need to augment the temporal join method to incorporate the *max_gap* constraint. This is easy to implement. Assume that we would like to compute $\mathcal{L}(A \rightarrow B)$ from the generating items A and B in Figure 1. Let's assume that *max_gap* is 15. Given a pair $\langle c, t_a\rangle$ in $\mathcal{L}(A)$, we check if there exists a pair $\langle c, t_b\rangle$ in $\mathcal{L}(B)$ such that $t_b \neq t_a$ and $t_b - t_a \leq max\_gap$. If so we add the pair $\langle c, t_a\rangle$ to the idlist of $A \rightarrow B$. For example, the pair $\langle c = 1, t_a = 10\rangle$ from A's idlist is added to $\mathcal{L}(A \rightarrow B)$, since there exists the pair $\langle c = 1, t_b = 20\rangle$ in B's idlist, with $t_b - t_a = 20 - 10 = 10 \leq 15 = max\_gap$. After doing this for all sids we get $\mathcal{L}(A \rightarrow B) = \{\langle 1, 10\rangle, \langle 2, 20\rangle\}$. Like minimum gaps, maximum gaps also have no effect on the equality join.

We now look at the new approach needed for enumerating sequences with maximum gap. Since a class is no longer self-contained, we cannot simply perform a self-join of the class members against itself. Instead, we can do a join with $\mathcal{F}_2$, the set of frequent 2-sequences, which is already known. For example, let's consider the suffix class $[B] = \{AB, A \rightarrow B, B \rightarrow B\}$, and $[A] = \{A \rightarrow A, B \rightarrow A\}$. Let's look at the element $A \rightarrow B$ in class $[B]$. To extend this sequence by one more item, we perform a temporal join with all members of $[A]$, since A is the first item of $A \rightarrow B$. This produces the new candidate class $[A \rightarrow B] = \{A \rightarrow A \rightarrow B, B \rightarrow A \rightarrow B\}$. If any of the candidates are frequent, the same process is repeated for the class $[A \rightarrow B]$. We recursively carry out joins with $\mathcal{F}_2$ for each new class until no extension is found to be frequent. In other words, to extend a sequence $X \rightarrow S$ in class $[S]$, we join it with all 2-sequences in the suffix class $[X]$. For each $Z \rightarrow X$ (or ZX) in $[X]$ we generate the candidate $Z \rightarrow X \rightarrow S$ (or $ZX \rightarrow S$).

Handling Time Window: A time window indicates

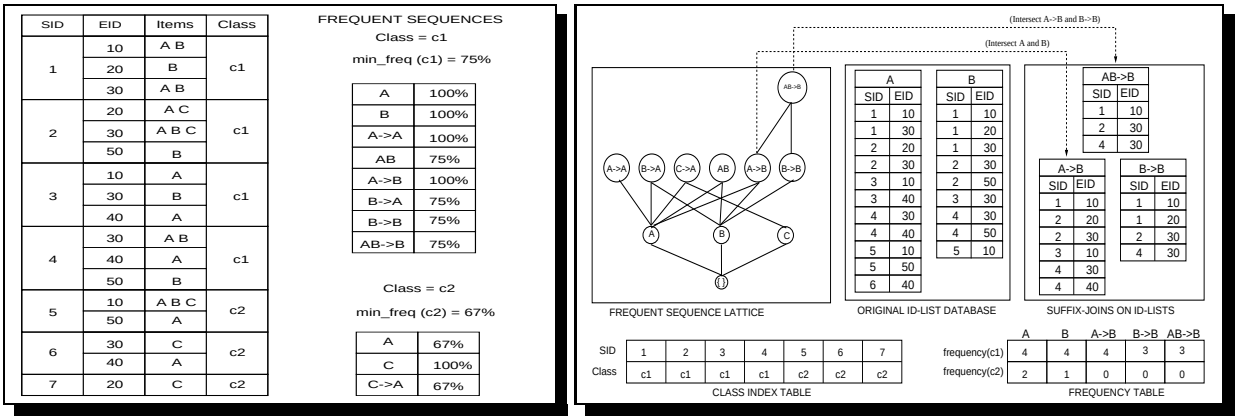

Figure 3: A) Database with Classes, B) Frequent Sequence Lattice and Frequency Computation

that we are interested in patterns that occurred within that time-frame, i.e., instead of minimum and maximum gap constraints that apply between sequence elements, the time window restriction applies to the entire sequence. Fortunately, time window is a class-preserving operation, requiring no change to the enumeration, but only a small check while performing joins. For example, if the sequence $X \rightarrow Y \rightarrow S$ is within time-window δ, then clearly any subsequence must also be within the same widow-size.

Incorporating time windows is not altogether easy due to our choice of idlist structure. It stores only the *eid* associated with the prefix item, but now we need to know the time between the first and last items in a sequence. For example, let window be 20, and let's assume that X occurs at time 20, XY at 30, and Y at time 50, for an input-sequence c. Then the idlist for $X \rightarrow Y$ has the pairs $\{\langle c, 20\rangle, \langle c, 30\rangle\}$ and for $Y \rightarrow Y$ the pair $(c, 30)$. Then we don't have enough information to say if $X \rightarrow Y \rightarrow Y$ occurs within a window of 20. If we look only at the eid for the first items, the difference is $30 - 20 = 10$. But clearly $X \rightarrow Y \rightarrow Y$ occurs in a window given by $50 - 20 = 30$, and not within a $window = 20$. Thus using the prefix *eids* alone one cannot determine the window length. We need to somehow know the time between the first and the last eid for a sequence (call it $diff$), but this information is not available in the idlist format. Our solution is to add an extra column to the idlist, so that we have triples of the form $\langle sid, eid, diff\rangle$, where initially $diff = 0$ for a single item. This is sufficient to check if a sequence is within a given window. For our example above, the idlist of $X \rightarrow Y$ now has $\{\langle c, 20, 10\rangle, \langle c, 30, 20\rangle\}$ and of $Y \rightarrow Y$ has $\langle c, 30, 20\rangle$. The gap between $\langle c, 20, 10\rangle$ and $\langle c, 30, 20\rangle$ is 10, and if we add this to the $diff = 20$ of $\langle c, 30, 20\rangle$, we get a number greater than our window size. Thus $X \rightarrow Y \rightarrow Y$ doesn't occur within a window of 20.

Handling Item Constraints: An advantage of the vertical format and equivalence classes is the ease with which one can incorporate constraints on the items that can appear in a sequence. Assume the task is to return sequences containing certain user-specified items, i.e., an *inclusion* constraint. We can examine each suffix class as it is generated (starting from the parent classes), and generate a new class from a sequence only if it contains any one of the specified items. Due to the self contained nature of equivalence classes, all possible frequent sequences containing that item will eventually be generated. Consider a related task of *excluding* certain items. In this case we can simply remove that item from the parent class. Thereafter, the item will never appear in any sequence. Another example of an item constraint is to consider a group of items or a sequence as a "super-item". All we have to do is compute the idlist for the super-item and thereafter treat it as a single item. All sequences containing the super-item can then be found.

Handling Classes: This constraint is applicable for classification datasets, i.e., where each input-sequence has a class label. Let β be a sequence and c be a class label. The *confidence* of the rule $\beta \Rightarrow c$, denoted $conf(\beta, c)$, is the conditional probability that c is the label of a sequence given that it contains sequence β. That is, $conf(\beta, c) = fr(\beta, \mathcal{D}_c)/fr(\beta, \mathcal{D})$ where $\mathcal{D}_c$ is the subset of the database $\mathcal{D}$ with class label c. Our goal is to find all frequent sequences with high confidence. Figure 3 shows an example database with labels. There are 7 sequences, 4 belonging to class c_1, and 3 belonging to class c_2. In general there can be more than two classes. We are looking for different *min_sup* in each class. For example, while C is frequent for class c_2, it's not frequent for class c_1. The rule $C \Rightarrow c_2$ has confidence $3/4 = 0.75$, while the rule $C \Rightarrow c_1$ has confidence $1/4 = 0.25$.

As before, we use temporal joins to enumerate the sequences. The join is essentially the same as in the non-class case, except we now also maintain the *class index table* indicating the classes for each input-sequence. Using this table we are able to determine the frequency of a sequence in all the classes at the same time. For example, A occurs in sids $\{1, 2, 3, 4, 5, 6\}$. However sids $\{1, 2, 3, 4\}$ have label c_1 and $\{5, 6\}$ have label c_2. Thus the frequency of A is 4 for c_1, and 2 for c_2. The class frequencies for each pattern are shown in the *frequency table* in Figure 3.

cSPADE (min_sup):
1. $\mathcal{P} = \{$ parent classes $P_i\}$;
2. **for** each parent class $P_i \in \mathcal{P}$ **do** Enumerate-Frequent(P_i);

Enumerate-Frequent(S):
1. **for** all sequences $A_i \in S$ **do**
2. **if** (maxgap) *//join with* $\mathcal{F}_2$
3. p = Prefix-Item(A_i);
4. N = { all 2-sequences A_j in class $[p]$}
5. **else** *// self-join*
6. N = { all sequences $A_j \in S$, with $j \geq i$}
7. **for** all sequences $\alpha \in N$ **do**
8. **if** (length(R) <= max_l and width(R) <= max_w and accuracy(R) $\neq$ 100%)
9. $\mathcal{L}(R)$=Constrained-Temporal-Join($\mathcal{L}(A_i)$, $\mathcal{L}(\alpha)$, min_gap, max_gap, $window$);
10. **if** ($\sigma(R, c_i) \geq$ min_sup(c_i)) **then**
11. $T = T \cup \{R\}$; print R;
12. Enumerate-Frequent(T);
13. delete S;

Figure 4: Pseudo-code for cSPADE

Frequent sequences are enumerated in a depth-first manner, however, this time sequences found to be frequent for *any* class c_i at the current level, form classes for the next

level, i.e., instead of a global minimum support value, we use a class specific support value min_sup(c_i). This process is repeated until all frequent sequences have been enumerated. In recent work [5] we showed how to use the mined sequences for feature extraction and selection in temporal domains, by selecting the sequences highly predictive of a class. These features were input to standard classification algorithms, improving classification accuracy by 10-50%.

Here we briefly review two effective pruning techniques that produce sequences that are not redundant and are distinctive of at least one class. Let $M(s, \mathcal{D})$ be the sequences in $\mathcal{D}$ that contain sequence s. We say that sequence s_1 *subsumes* sequence s_2 with respect to predicting class c iff $M(s_2, \mathcal{D}_c) \subseteq M(s_1, \mathcal{D}_c)$ and $M(s_1, \mathcal{D}_{\neg c}) \subseteq M(s_2, \mathcal{D}_{\neg c})$. Intuitively, if s_1 subsumes s_2 for class c then s_1 is superior to s_2 for predicting c because s_1 covers every example of c in the training data that s_2 covers and s_1 covers only a subset of the non-c examples that s_2 covers. The first pruning rule is that we do not extend any sequence with 100% accuracy, i.e., it occurs in only one class. Let s_1 be a sequence contained by examples of only one class. Extensions of s_1 will be subsumed by s_1, and thus need not be examined. The second pruning rule is as follows: We say that $A \rightsquigarrow B$ in a dataset if B occurs in every event in every sequence in which A occurs. If $A \rightsquigarrow B$ then any feature containing an event with both A and B will be subsumed by one of its subsets, and thus we can prune it. The pseudo-code for cSPADE incorporating the above constraints is shown in Figure 4.

5. EXPERIMENTAL RESULTS

Experiments were done on a 450Mhz Pentium II with 256MB memory running Linux 6.0.

Synthetic Datasets We used the publicly available dataset generation code from IBM (http://www.almaden.ibm.com/cs/quest/syndata.html). These datasets mimic real-world transactions, where people buy a sequence of sets of items. Some customers may buy only some items from the sequences, or they may buy items from multiple sequences. The customer sequence size and transaction size are clustered around a mean and a few of them may have many elements. The datasets are generated using the following process. First N_I maximal itemsets of average size I are generated by choosing from N items. Then N_S maximal sequences of average size S are created by assigning itemsets from N_I to each sequence. Next a customer sequence of average C transactions is created, and sequences in N_S are assigned to different customer elements, respecting the average transaction size of T. The generation stops when D customers have been generated. Like [10] we set $N_S = 5000$, $N_I = 25000$ and $N = 10000$. Table 1 shows the datasets with their parameter settings. We refer the reader to [1] for additional details on the dataset generation.

Dataset	C	T	S	I	D	Size
C10T5S4I2.5D200K	10	5	4	2.5	200K	76MB
C20T5S8I2.5D200K	20	20	8	2.5	200K	151MB
EvacuationPlan	4.1	7.6	-	-	202K	40MB
FireWorld	36.3	4.6	-	-	1000	1.3MB
Spelling	27.3	3	-	-	2917	2.17MB

Table 1: Synthetic and Real Dataset Parameters

EvacuationPlan This real-life dataset was obtained from a evacuation planning domain. The planner generates plans for routing commodities from one city to another. Each plan is an input-sequence, with the plan identifier as the sid. An event consists of an identifier, an outcome (such as "success", "late", or "failure"), an action name (such as "move", or "load"), and a set of additional parameters specifying things such as origin, destination, vehicle type ("truck", or "helicopter"), weather conditions, and so on. The mining goal is to identify the causes of plan failures.

FireWorld We obtained this dataset from simple forest-fire domain [5]. We use a grid representation of the terrain. Each grid cell can contain vegetation, water, or a base. We label each instance with `SUCCESS` if none of the locations with bases have been burned in the final state, or `FAILURE` otherwise. Thus, the goal is to predict if the bulldozers will prevent the bases from burning, given a partial execution trace of the plan. For this data, there were 38 items. In the experiments reported below, we used $min\_sup = 20\%$, $max_w = 3$, and $max_l = 3$, to make the problem tractable.

Spelling To create this dataset, we chose two commonly confused words, such as "there" and "their", and searched for sentences in the 1-million-word Brown corpus containing either word [5]. We removed the target word and then represented each word by the word itself, the part-of-speech tag in the Brown corpus, and the position relative to the target word. For "there" vs. "their" dataset there were 2917 input-sequences, and 5663 feature/value pairs or items. In the experiments reported below, we used a $min\_sup = 5\%$, $max_w = 3$, and $max_l = 2$.

Number of Constrained Frequent Sequences: Figure 5 shows the total number of frequent sequences found for different databases when we incorporate the minimum gap, maximum gap, time window, and maxmin constraints. The maxmin constraint only allows sequences that have elements exactly δ apart, i.e.., with $min\_gap = max\_gap = \delta$. The 'Base' line is the total number of unconstrained sequences. As expected the number of sequences decrease with increasing minimum gap, while they increase with increasing maximum gap and window size. The total number of frequent sequences when one imposes gap or window constraints can be much smaller than the base case. In fact, in many domains it is infeasible to mine all the frequent sequences without any constraints due to combinatorial blowup.

Running Time with Gaps: Figure 6 shows the effect of incorporating minimum gaps, maximum gaps and maxmin gaps on the running time of cSPADE. In the figure a constraint value of 1 always denotes the unconstrained case.

For each database, we compare the performance of cSPADE with GSP [10] for the various gap constraints. Let's consider the unconstrained case (i.e., x-axis=1). We find that cSPADE outperforms GSP by factor of 2 (to almost 10 on other datasets not shown here). When we consider minimum and maxmin gaps, we find that the difference between the two methods narrows. This is because the number of frequent sequences decreases drastically with increasing gap values. In other words, since minimum gap (and time windows as well) preserve the suffix class structure, we find that the time to incorporate these constraints is a fraction of the cost of mining everything, varying with the selectivity of the gap. Larger the minimum gap (and smaller the window), the fewer are the patterns found, and vice versa.

On the other hand we find that the time difference between the two algorithms widens with increasing maximum gap (and can be more than a factor of 10). Handling maximum gaps destroys the self-sufficiency of a suffix class, and thus the performance drops because the number of frequent sequences increases with increasing max_gap. It should also be noted that the running time of cSPADE is less sensitive to an increase in the max_gap than GSP, since the running time only gradually increases with increasing values.

Figure 6 also shows the performance of two variants of cSPADE and GSP. For these two variants, called pre-cSPADE

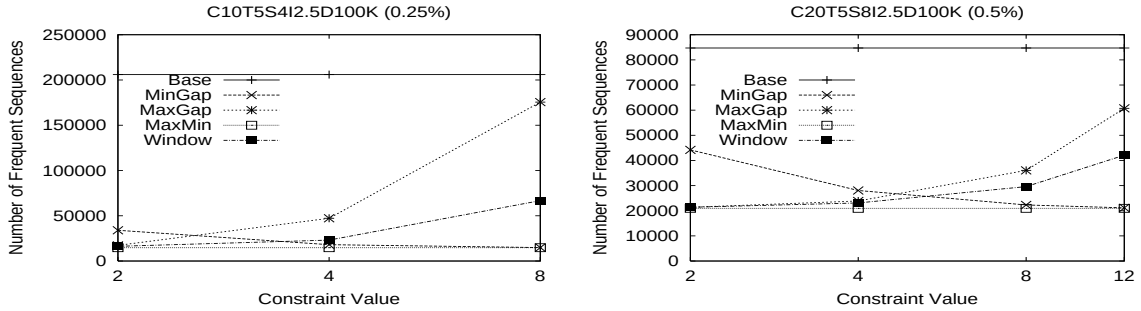

Figure 5: Number of Constrained Frequent Sequences

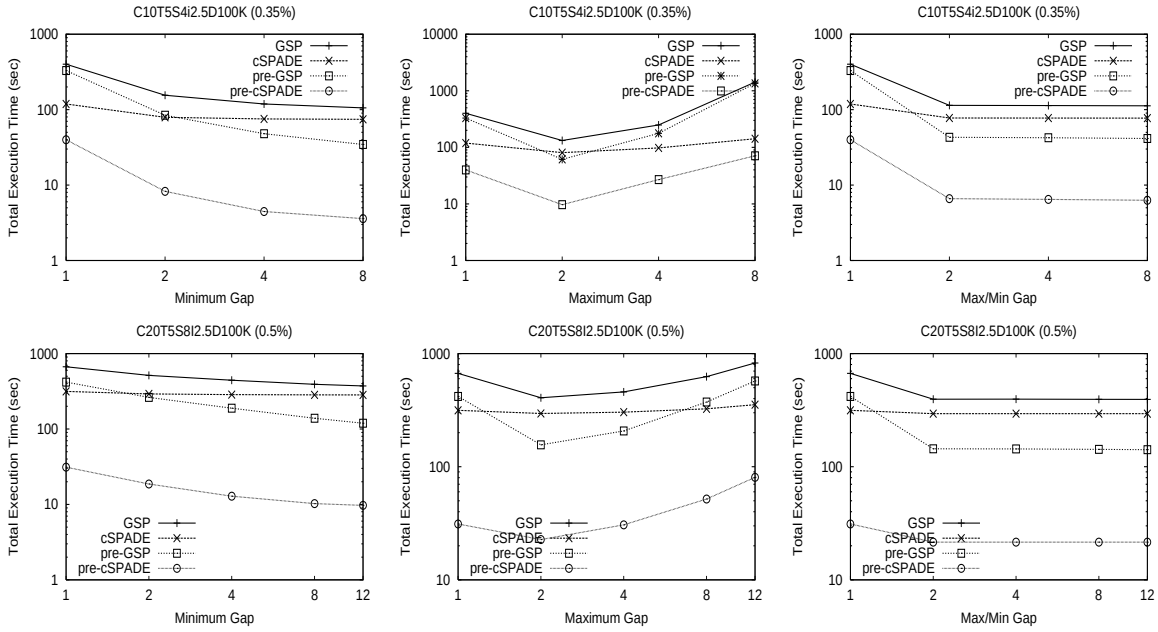

Figure 6: Effect of Minimum Gap, Maximum Gap, and MaxMin Gap

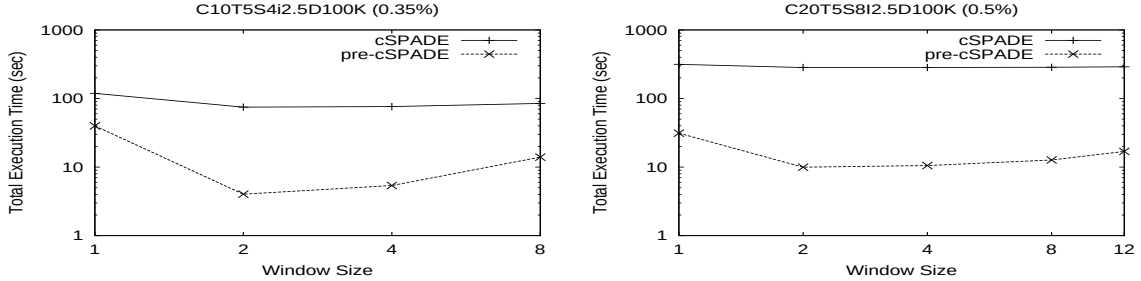

Figure 7: Effect of Window Size

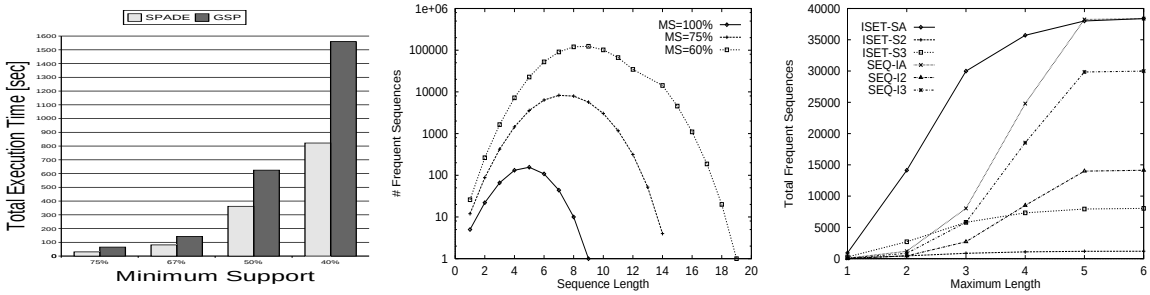

Figure 8: A) cSPADEvs. GSP, B) Sequences Mined at Various Levels of Minimum Support (MS), C) Number of Sequences Discovered with Length Restrictions (75% Minimum Support). ISET-SA shows all sequences plotted against limits on the itemset length; ISET-S2(3) the sequences with 2(3) itemsets per sequence plotted against the itemset length; SEQ-IA all sequences against varying #itemsets/sequence; and SEQ-I2(3) the sequences with fixed itemset length of 2(3) against varying #itemsets/sequence.

Experiment	CPU seconds with no pruning	CPU seconds with only $A \rightsquigarrow B$ pruning	CPU seconds with all pruning	# Sequences with no pruning	#Sequences with only $A \rightsquigarrow B$ pruning	# Sequences with all pruning
FireWorld	5.8 *hours*	560	559	25,336,097	511,215	511,215
Spelling	490	407	410	1,126,114	999,327	971,085

Table 2: Mining with Classes: with and without the $A \rightsquigarrow B$ pruning.

and pre-GSP, we assume some one-time pre-processed information is available. Namely that an invariant set of frequent 2-sequences (above a minimum threshold) has been precomputed. Thus pre-cSPADE and pre-GSP can use these values to directly start computing k-sequences with $k \geq 3$. Here we find that pre-cSPADE typically is 10 times better than pre-GSP, and can be as high as 50 times better (the gap is expected to widen if lower values of support are considered). There are several reasons why cSPADE (pre-cSPADE) outperforms GSP (pre-GSP): 1) cSPADE uses only simple join operation on idlists. As the length of a frequent sequence increases, the size of its idlist decreases, resulting in very fast joins. 2) No complicated hash-tree structure is used, and no overhead of generating and searching of subsequences is incurred. These structures typically have very poor locality. On the other hand SPADE has excellent locality, since a join requires only a linear scan of two lists. 3) As the minimum support is lowered, more and larger frequent sequences are found. GSP makes a complete dataset scan for each iteration. cSPADE on the other hand restricts itself to only a few scans. It thus cuts down the I/O costs. Since SPIRIT [2] and its variants (using the most general regular expression) default to GSP we expect cSPADE to outperform it as well.

An very interesting property of the non-class-preserving *max_gap* constraint is that as one increases the gap size, the time for mining exceeds the base (unconstrained) time, sometimes by as much as a factor of 2 or 3 for pre-cSPADE, and a factor of 3 to 4 for GSP-variants. cSPADE on the other hand mined all the sequences within a factor of 1.2 of the base time, showing excellent resilience to changes in *max_gap*. For cSPADE variants the time increases since the algorithms perform joins of $\mathcal{F}_k$ with $\mathcal{F}_2$ (instead of $\mathcal{F}_k$) to obtain the new candidates $\mathcal{F}_{k+1}$, which leads to more unnecessary intersections. For GSP variants the time increases due to the complexity of incorporating the *max_gap* check during support counting. Since *max_gap* is representative of many of the other non-class-preserving constraints, we expect this behavior to apply to those cases as well. Incidentally, a similar observation was made in SPIRIT [2].

Effect of Window Size: Figure 7 shows how the running time of cSPADE changes with window size (we did not implement the window constraint for GSP). Since the window constraint is class-preserving it runs very fast compared to the base unconstrained case.

	MS=100%	MS=75%	MS=60%
#Sequences	544	38386	642597
Time	0.2s	19.8s	185.0s

Table 3: EvacuationPlan

Effect of Length and Width Constraints: For EvacuationPlan data, the number of frequent sequences of different lengths for various levels of minimum support, as well as a comparison of cSPADE and GSP (unconstrained case), are plotted in Figure 8, while the running times and the total number of frequent sequences is shown in Table 3. We can also reduce the number of patterns generated by putting limits on the maximum number of itemsets or events per sequence or the maximum length of an itemset. Figure 8 plots the total number of frequent sequences discovered under length restrictions. For example, there are 38386 total sequences at 75% *min_sup* (ISET-SA). But if we restrict the maximum itemset length to 2, then there are only 14135 sequences. If we restrict the maximum number of itemsets per sequence to 3, then we discover only 8037 sequences (ISET-S3), and so on. Due to the high frequency character of this domain, it was essential to put these restrictions, especially on the maximum length of an itemset to be able to use a low minimum support value, and to discover long sequences.

Handling Classes: Table 2 shows the impact on mining time of the $A \rightsquigarrow B$ pruning rule described in Section 4 when mining for sequences that are predictive of a class. The pruning rule did not make a great difference for Spelling, but made a tremendous difference in the FireWorld, where the same event descriptors often appear together. Without $A \rightsquigarrow B$ pruning, the FireWorld problem is essentially unsolvable because cSPADE finds over 20 million frequent sequences.

In this paper we presented a new algorithm, called cSPADE, for discovering the set of all frequent sequences with the following constraints: length and width restrictions, minimum and maximum gap between sequence elements, time window of occurrence of the whole sequence, item constraints for including or excluding certain items, and finding sequences distinctive of at least one *class*, i.e., a special attribute-value pair, that we are interested in predicting. We presented experimental results on a number of synthetic and real datasets to show the effectiveness and performance of our approach. The two main strengths of cSPADE are that it delivers performance far superior to existing approaches to constrained sequences, and that it incorporates the constraints with relative ease. As part of future work we plan to introduce a much wider class of constraints within cSPADE, such as regular expressions, relational or aggregate constraints on items, etc. Another direction is to directly search for "interesting" sequences in the presence of background information (such as user-specified beliefs), i.e., we would like to mine sequence that significantly bolster the belief or contradict it.

6. REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *11th Intl. Conf. on Data Engg.*, 1995.

[2] M. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In *25th Intl. Conf. Very Large Databases*, 1999.

[3] K. Hatonen et al. Knowledge discovery from telecommunication network alarm databases. In *12th Intl. Conf. Data Engineering*, February 1996.

[4] M. Klemettinen et al. Finding interesting rules from large sets of discovered association rules. In *3rd Intl. Conf. Information and Knowledge Management*, pages 401–407, November 1994.

[5] N. Lesh, M. J. Zaki, and M. Ogihara. Mining features for sequence classification. In *5th Intl. Conf. Knowledge Discovery and Data Mining*, August 1999.

[6] H. Mannila, H. Toivonen, and I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery: An International Journal*, 1(3):259–289, 1997.

[7] R. T. Ng et al. Exploratory mining and pruning optimizations of constrained association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, June 1998.

[8] T. Oates et al. Algorithms for finding temporal structure in data. In *6th Intl. Workshop on AI and Statistics*, March 1997.

[9] S. Parthasarathy et al. Incremental and interactive sequence mining. In *8th ACM CIKM Conference*, November 1999.

[10] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *5th Intl. Conf. Extending Database Technology*, March 1996.

[11] M. J. Zaki. Efficient enumeration of frequent sequences. In *7th Intl. Conf. Info. and Knowledge Management*, Nov 1998.

[12] M. J. Zaki, N. Lesh, and M. Ogihara. PLANMINE: Sequence mining for plan failures. In *4th Intl. Conf. Knowledge Discovery and Data Mining*, August 1998.