Mining Sequential Patterns with Regular Expression Constraints

Minos Garofalakis
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
minos@bell-labs.com

Rajeev Rastogi
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
rastogi@bell-labs.com

Kyuseok Shim
KAIST[*] and AITrc[†]
373-1 Kusong-Dong, Yusong-Gu
Taejon 305-701, Korea
shim@cs.kaist.ac.kr

Abstract

Discovering sequential patterns is an important problem in data mining with a host of application domains including medicine, telecommunications, and the World Wide Web. Conventional sequential pattern mining systems provide users with only a very restricted mechanism (based on minimum support) for specifying patterns of interest. As a consequence, the pattern mining process is typically characterized by lack of focus and users often end up paying inordinate computational costs just to be inundated with an overwhelming number of useless results. In this paper, we propose the use of Regular Expressions (REs) as a flexible constraint specification tool that enables user-controlled focus to be incorporated into the pattern mining process. We develop a family of novel algorithms (termed SPIRIT – Sequential Pattern mIning with Regular expressIon consTraints) for mining frequent sequential patterns that also satisfy user-specified RE constraints. The main distinguishing factor among the proposed schemes is the degree to which the RE constraints are enforced to prune the search space of patterns during computation. Our solutions provide valuable insights into the tradeoffs that arise when constraints that do not subscribe to nice properties (like anti-monotonicity) are integrated into the mining process. A quantitative exploration of these tradeoffs is conducted through an extensive experimental study on synthetic and real-life data sets. The experimental results clearly validate the effectiveness of our approach, showing that speedups of more than an order of magnitude are possible when RE constraints are pushed deep inside the mining process. Our experimentation with real-life data also illustrates the versatility of REs as a user-level tool for focusing on interesting patterns.

Keywords: Data Mining, Constraints, Sequential Patterns, Regular Expressions, Finite Automata.

1 Introduction

Discovering *sequential patterns* from a large database of sequences is an important problem in the field of knowledge discovery and data mining. Briefly, given a set of data sequences, the problem is to discover subsequences that are *frequent*, in the sense that the percentage of data sequences containing them exceeds a user-specified minimum *support* [3, 12]. Mining frequent sequential patterns has found a host of potential application domains, including retailing (i.e., market-basket data), telecommunications, medicine, and, more recently, the World Wide Web (WWW). In market-basket databases, each data

[*]Korea Advanced Institute of Science and Technology
[†]Advanced Information Technology Research Center at KAIST

sequence corresponds to items bought by an individual customer over time and frequently occurring patterns can be very useful for predicting future customer behavior. In telecommunications, frequent sequences of alarms output by network switches capture important relationships between alarm signals that can then be employed for on-line prediction, analysis, and correction of network faults. In the medical field, frequent temporal patterns of symptoms and diseases exhibited by patients identify strong symptom/disease correlations that can be an invaluable source of information for medical diagnosis and preventive medicine. Finally, in the context of the WWW, server sites typically generate huge volumes of daily log data capturing the sequences of page accesses for thousands or millions of users[1]. Discovering frequent user access patterns in WWW server logs can help improve system design (e.g., better hyperlinked structure between correlated pages) and lead to better marketing decisions (e.g., strategic advertisement placement).

As a more concrete example, the `Yahoo!` Internet directory (`www.yahoo.com`) enables users to locate interesting WWW documents by navigating through large *topic hierarchies* consisting of thousands of different document classes [4]. These hierarchies provide an effective way of dealing with the abundance problem present in today's keyword-based WWW search engines. The idea is to allow users to progressively refine their search by following specific *topic paths* (i.e., sequences of hyperlinks) along a (predefined) hierarchy. Given the wide variety of topics and the inherently fuzzy nature of document classification, there are numerous cases in which distinct topic paths lead to different document collections on very similar topics. For example, starting from `Yahoo!`'s home page users can locate information on hotels in New York City by following either `Travel:Yahoo!Travel:North America:United States:New York:New York City:Lodging: Hotels` or `Travel:Lodging:Yahoo!Lodging:New York:New York Cities:New York City:Hotels and Motels`, where "`:`" denotes a parent-child link in the topic hierarchy. Mining user access logs to determine the most frequently accessed topic paths is a task of immense marketing value, e.g., for a hotel or restaurant business in New York City trying to select a strategic set of WWW locations for its advertising campaign.

The design of effective algorithms for mining frequent sequential patterns has been the subject of several studies in recent years [3, 5, 8, 9, 12, 14]. Ignoring small differences in the problem definition (e.g., form of input data, definition of a subsequence), a major common thread that runs through the vast majority of earlier work is the *lack of user-controlled focus in the pattern mining process.* Typically, the interaction of the user with the pattern mining system is limited to specifying a lower bound on the desired support for the extracted patterns. The system then executes an appropriate mining algorithm and returns a very large number of sequential patterns, only some of which may be of actual interest to the user. Despite its conceptual simplicity, this "unfocused" approach to sequential pattern mining suffers from two major drawbacks.

1. *Disproportionate computational cost for selective users.* Given a database of sequences and a fixed value for the minimum support threshold, the computational cost of the pattern mining process is fixed for any potential user. The problem here is that despite the development of efficient algorithms, pattern mining remains a computation-intensive task typically taking hours to complete. Thus, ignoring user focus can be extremely unfair to a highly selective user that is only interested in patterns of a very specific form.

2. *Overwhelming volume of potentially useless results.* The lack of tools to express user focus during the pattern mining process means that selective users will typically be swamped with a huge number of frequent patterns, most of which are useless for their purposes. Sorting through this

[1] In general, WWW servers only have knowledge of the IP address of the user/proxy requesting a specific web page. However, *referrers* and *cookies* can be used to determine the sequence of accesses for a particular user (without compromising the user's identity).

morass of data to find specific pattern forms can be a daunting task, even for the most experienced user.

The above discussion clearly demonstrates the need for novel pattern mining solutions that enable the incorporation of user-controlled focus in the mining process. There are two main components that any such solution must provide. First, given the inadequacy of simple support constraints, we need a *flexible constraint specification language* that allows users to express the specific family of sequential patterns that they are interested in. For instance, returning to our earlier "New York City hotels" example, a hotel planning its ad placement may only be interested in paths that (a) begin with `Travel`, (b) end in either `Hotels` or `Hotels and Motels`, and (c) contain at least one of `Lodging`, `Yahoo!Lodging`, `Yahoo!Travel`, `New York`, or `New York City`, since these are the only topics directly related to its line of business. Second, we need novel pattern mining algorithms that can exploit user focus by *pushing user-specified constraints deep inside the mining process*. The abstract goal here is to exploit pattern constraints to prune the computational cost and ensure system performance that is *commensurate* with the level of user focus (i.e., constraint selectivity). (Simply put, selective users should not be penalized for results that they did not ask for.)

We should note that even though recent work has addressed similar problems in the context of association rule mining [10, 11, 13], the problem of incorporating a rich set of user-specified constraints in sequential pattern mining remains, to the best of our knowledge, unexplored. Furthermore, as we will discover later in the paper, pattern constraints raise a host of new issues specific to sequence mining (e.g., due to the explicit ordering of items) that were not considered in the subset and aggregation constraints for itemsets considered in [10, 11, 13]. For example, our pattern constraints do not satisfy the property of *anti-monotonicity* [10]; that is, the fact that a sequence satisfies a pattern constraint does not imply that all its subsequences satisfy the same constraint. These differences mandate novel solutions that are completely independent of earlier results on constrained association rule mining [10, 11, 13].

In this paper, we formulate the problem of mining sequential patterns with *regular expression constraints* and we develop novel, efficient algorithmic solutions for pushing regular expressions inside the pattern mining process. Our choice of regular expressions (REs) as a constraint specification tool is motivated by two important factors. First, REs provide a simple, natural syntax for the succinct specification of families of sequential patterns. Second, REs possess sufficient expressive power for specifying a wide range of interesting, non-trivial pattern constraints. These observations are validated by the extensive use of REs in everyday string processing tasks (e.g., UNIX shell utilities like `grep` or `ls`) as well as in recent proposals on query languages for sequence data (e.g., the Shape Definition Language of Agrawal et al. [1]). Returning once again to our "New York City hotels" example, note that the constraint on topic paths described earlier in this section can be simply expressed as the following RE:

```
Travel ( Lodging | Yahoo!Lodging | Yahoo!Travel | New York | New York City )
                                                ( Hotels | Hotels and Motels ),
```

where "`|`" stands for disjunction. We propose a family of novel algorithms (termed SPIRIT – Sequential Pattern mIning with Regular expressIon consTraints) for mining frequent sequential patterns that also belong to the language defined by the user-specified RE. Our algorithms exploit the equivalence of REs to deterministic finite automata [7] to push RE constraints deep inside the pattern mining computation. The main distinguishing factor among the proposed schemes is the *degree* to which the RE constraint is enforced within the generation and pruning of candidate patterns during the mining process. We observe that, varying the level of user focus (i.e., RE enforcement) during pattern mining gives rise to certain interesting tradeoffs with respect to computational effectiveness. Enforcing the RE constraint at each phase of the mining process certainly minimizes the amount of "state" maintained after each phase, focusing only on patterns that could potentially be in the final answer set. On the other hand, minimizing this maintained state may not always be the best solution since it can, for example, limit our ability to do effective support-based pruning in later phases. Such tradeoffs are obviously related to our previous observation that RE constraints are *not* anti-monotone [10]. We believe that our results provide useful insights into the more general problem of constraint-driven, ad-hoc data mining, showing that there can be a whole spectrum of choices for dealing with constraints, even when they do not subscribe to nice properties like anti-monotonicity or succinctness [10]. An extensive experimental study with synthetic as well as real-life data sets is conducted to explore the tradeoffs involved and their impact on the overall effectiveness of our algorithms. Our results indicate that incorporating RE constraints into the pattern mining computation can some times yield more than an order of magnitude improvement in performance, thus validating the effectiveness of our approach. Our experimentation with real-life WWW server log data also demonstrates the versatility of REs as a user-level tool for focusing on interesting patterns.

The remainder of this paper is organized as follows. Section 2 reviews related research in the area of data mining and knowledge discovery. In Section 3, we present the formulation of our constrained pattern mining problem along with the necessary definitions and notation. The SPIRIT family of algorithms is presented in Section 4 and extensions to deal with sequences of itemsets and distance constraints are proposed in Section 5. Section 6 discusses the findings of an extensive experimental study of the SPIRIT algorithms using both synthetic and real-life data sets. Finally, Section 7 concludes the paper and identifies directions for future research. The work reported in this paper has been done in the context of the $\mathcal{SERENDIP}$ data mining project at Bell Laboratories (`www.bell-labs.com/projects/serendip`).

2 Related Work

Agrawal and Srikant [3] introduce the problem of mining sequential patterns from a set of market-basket data sequences, where each sequence element is a set of items purchased in the same transaction. They propose and experimentally evaluate three algorithms based on the general Apriori framework [2]. In more recent work, Srikant and Agrawal [12] generalize their earlier problem definition to allow for (a) *time-gap constraints*, placing bounds on the time separation between adjacent elements in a pattern, (b) *sliding time windows*, permitting elements of a pattern to span a set of transactions within a user-specified window, and (c) *item taxonomies*, enabling the discovery of patterns across different levels of a user-defined taxonomy. They propose GSP (Generalized Sequential Patterns), an Apriori-style algorithm that efficiently handles all three extensions to the basic pattern mining problem and, at the same time, offers much better performance than their earlier schemes. Chen et al. [5] study the problem of mining access patterns in a hyperlinked information-providing environment. Essentially, they consider a simpler version of the pattern mining problem described by Agrawal and Srikant, where (a) elements of a data sequence are assumed to be *single items*, and (b) elements of a discovered pattern are required to be *consecutive* in the data. Their algorithm converts the original sequences into a set of *maximal forward references* and then applies association rule mining techniques for finding frequent reference sequences in that set. Mannila et al. [8, 9] consider the problem of discovering frequent *episodes* (i.e., partial orders of events occurring within a small time window) in a large input sequence of events. Their solution is based on moving a time window across the input sequence, counting the episodes that occur in some user-specified percentage of windows. In a different paper, they describe the application of their techniques in the analysis of telecommunication network alarm databases [6]. The issue of providing users with a rich constraint specification language and exploiting user-specified constraints to reduce the computational cost of the mining process has not been addressed in any of these papers.

The problem of discovering approximate structural patterns in a database of genetic sequences, explored by Wang et al. [14], is closely related to our work. Besides the minimum support threshold, their solution allows users to specify (a) the desired form of the patterns as sequences of *consecutive* symbols separated by variable-length don't cares (VLDCs), (b) a lower bound on the length of the discovered patterns, and (c) an upper bound on the *edit distance* allowed between a mined pattern and a data sequence that contains it (assuming an optimal substitution for the VLDCs). Their algorithm uses a random sample of the input sequences to build a main-memory data structure, termed *generalized suffix tree*, that is used to obtain an initial set of candidate pattern segments and screen out candidates that are *unlikely* to be frequent based on their occurrence counts in the sample. The entire database is then scanned and filtered (using a pattern matching algorithm) to verify that the remaining candidates are indeed frequent answers to the user query.

The work of Wang et al. [14] differs from ours in several respects. First, they suggest a rather restricted form of user-specified constraints on patterns (i.e., segments of consecutive symbols separated by VLDCs), compared to our *fully general* RE constraints. Second, the algorithms in [14] are based on main memory data structures and probabilistic techniques (i.e., random sampling) for pruning unlikely candidates with high probability, so they typically return only a *subset* of the complete answer. Our techniques, on the other hand, are not restricted by main memory constructs and are guaranteed to find *all* qualifying patterns. Finally, we show how our algorithms can be extended to deal with the more general "sequences of itemsets" case, which is not considered in [14]. We should note, however, that Wang et al. also introduce the interesting idea of allowing *partial* matches between a pattern and a data sequence through the use of their edit distance threshold. Although we do not explore such notions of "matching with limited errors" in this paper, we plan to address the issue in our future research.

In the related problem of mining association rules from market-basket data, Tsur et al. [13] introduce the notion of "query flocks" as a generalization of associations and discuss optimization techniques that try to intelligently schedule the use of Apriori-style, support-based filtering for efficient query-flock evaluation. Their techniques are applicable only for filters/constraints that (like itemset support) satisfy certain monotonicity criteria [13]. Srikant et al. [11] and Ng et al. [10] investigate the problem of incorporating additional constraints on frequent itemsets for faster association-rule mining. The form of the constraints and the algorithmic techniques developed in these papers are tailored to simple *sets* of items and, consequently, are inapplicable to sequences which are the focal point of this paper. For example, none of the constraint mechanisms presented in [10, 11] allows users to specify *item ordering*, an essential ingredient of sequential patterns. More specifically, Srikant et al. [11] propose techniques for handling membership (i.e., subset) constraints in the context of a user-defined item taxonomy. They make the interesting observation that exploiting the itemset membership constraint to a larger degree for pruning may not always optimize performance. A small example is presented to show that, in some cases, more pruning due to the membership constraint may result in fewer candidates being pruned by the minimum support constraint. This is very similar to the phenomenon that we have observed for the SPIRIT family of pattern mining algorithms developed later in this paper. Ng et al. [10] consider more general categories of subset and aggregation constraints for itemsets. They identify and analyze two key properties of itemset constraints, *anti-monotonicity* and *succinctness*, that can be used to significantly increase the level of itemset pruning. Informally, a constraint $\mathcal{C}$ is anti-monotone if every subset of an itemset satisfying $\mathcal{C}$ is also guaranteed to satisfy $\mathcal{C}$. A constraint $\mathcal{C}$ is succinct if there exists a concise characterization of all itemsets satisfying $\mathcal{C}$ that uses only selection, union, minus, and powerset operations over the item domain. Unfortunately, RE constraints on sequential patterns are *not* anti-monotone and no obvious translation of the succinctness property exists for the sequential pattern domain. This clearly limits the usefulness of these earlier results for our problem setting.

3 Definitions and Problem Formulation

3.1 Sequences, Regular Expressions, and Finite Automata

The main input to our mining problem is a database of sequences, where each sequence is an ordered list of *elements*. These elements can be either (a) *simple items* from a fixed set of literals (e.g., the identifiers of WWW documents available at a server [5], the amino acid symbols used in protein analysis [14]), or (b) *itemsets*, that is, non-empty sets of items (e.g., books bought by a customer in the same transaction [12]). The list of elements of a data sequence s is denoted by $< s_1\ s_2 \cdots s_n >$, where s_i is the i^{th} element of s. If s is a sequence of itemsets, then we represent the set of simple items corresponding to element s_i as $\{s_i^1, \ldots, s_i^{k_i}\}$. We use $|s|$ to denote the *length* (i.e., number of elements) of sequence s and $|s_i|$ to denote the number of items in element s_i. A sequence of length k is referred to as a k-*sequence*. (We consider the terms "sequence" and "sequential pattern" to be equivalent for the remainder of our discussion.) Table 1 summarizes the notation used throughout the paper with a brief description of its semantics. We provide detailed definitions of some of these parameters in the text. Additional notation will be introduced when necessary.

Symbol	**Semantics**
$s, t, u, \ldots$	Generic sequences in the input database
$< s\ t >$	Sequence resulting from the concatenation of sequences s and t
$\|s\|$	Length, i.e., number of elements, of sequence s
s_i	i^{th} element of sequence s
$\|s_i\|$	Number of items in element s_i
s_i^*	Zero or more occurrences of element s_i (Kleene closure operator)
$s_i \mid s_j$	Select one element out of s_i and s_j (Disjunction operator)
$\mathcal{R}$	Regular expression (RE) constraint
$\mathcal{A_R}$	Deterministic finite automaton for RE $\mathcal{R}$
$b, c, d, \ldots$	Generic states in automaton $\mathcal{A_R}$
a	Start state of automaton $\mathcal{A_R}$
$b \xrightarrow{s_i} c$	Transition from state b to state c in $\mathcal{A_R}$ on element s_i
$b \stackrel{s}{\Rightarrow} c$	Transition path from state b to state c in $\mathcal{A_R}$ on the sequence of elements s
C_k	Set of candidate k-sequences
F_k	Set of frequent k-sequences

Table 1: Notation.

Consider two data sequences $s =< s_1\ s_2\ \cdots\ s_n >$ and $t =< t_1\ t_2\ \cdots\ t_m >$. We say that s is a *subsequence* of t if s is a "projection" of t, derived by deleting elements and/or items from t. More formally, s is a subsequence of t if there exist integers $j_1 < j_2 < \ldots < j_n$ such that $s_1 \subseteq t_{j_1}$, $s_2 \subseteq t_{j_2}$, ..., $s_n \subseteq t_{j_n}$. Note that for sequences of simple items the above condition translates to $s_1 = t_{j_1}$, $s_2 = t_{j_2}$, ..., $s_n = t_{j_n}$. For example, sequences $< 1\ 3 >$ and $< 1\ 2\ 4 >$ are subsequences of $< 1\ 2\ 3\ 4 >$, while $< 3\ 1 >$ is not. Srikant and Agrawal [12] observe that, when mining market-basket sequential patterns, users often want to place a bound on the *maximum distance* between the occurrence of adjacent pattern elements in a data sequence. For example, if a customer buys bread today and milk after a couple of weeks then the two purchases should probably not be seen as being correlated. Following [12], we define sequence s to be a *subsequence with a maximum distance constraint of* δ, or alternately δ-*distance subsequence*, of t if there exist integers $j_1 < j_2 < \ldots < j_n$ such that $s_1 \subseteq t_{j_1}$, $s_2 \subseteq t_{j_2}$, ..., $s_n \subseteq t_{j_n}$ *and* $j_k - j_{k-1} \leq \delta$ for each $k = 2, 3, \ldots, n$. That is, occurrences of adjacent elements of s within t are not separated by more than δ elements[2]. As a special case of the above definition, we say that s is a *contiguous* subsequence of t if s is a 1-distance subsequence of t, i.e., the elements of s can be mapped to a contiguous segment of t.

A sequence s is said to *contain* a sequence p, if p is a subsequence of s (or, δ-distance subsequence) of s. (The notion of "subsequence" used should always be clear from the context.) We define the *support* of a sequential pattern p as the fraction of the sequences in the input database that contain p. Given a set of sequences $\mathcal{S}$, we say that $s \in \mathcal{S}$ is *maximal* if there are no sequences in $\mathcal{S} - \{s\}$ that contain it.

A RE constraint $\mathcal{R}$ is specified as a RE over the alphabet of sequence elements using the established set of RE operators, such as disjunction ($|$) and Kleene closure (*) [7]. Thus, a RE constraint $\mathcal{R}$ specifies a language of strings over the element alphabet or, equivalently, a regular family of sequential patterns that is of interest to the user. A well-known result from complexity theory states that REs have exactly the same expressive power as *deterministic finite automata* [7]. Thus, given any RE $\mathcal{R}$, we can always build a deterministic finite automaton $\mathcal{A}_{\mathcal{R}}$ such that $\mathcal{A}_{\mathcal{R}}$ accepts exactly the language generated by $\mathcal{R}$. Informally, a deterministic finite automaton is a finite state machine with (a) a well-defined *start* state (denoted by a) and one or more *accept* states, and (b) deterministic transitions across states on symbols of the input alphabet (in our case, sequence elements). A transition from state b to state c on element s_i is denoted by $b \xrightarrow{s_i} c$. We also use the shorthand $b \stackrel{s}{\Rightarrow} c$ to denote the sequence of transitions on the elements of sequence s starting at state b and ending in state c. A sequence s is *accepted by* $\mathcal{A}_{\mathcal{R}}$ if following the sequence of transitions for the elements of s from the start state results in an accept state. Figure 1 depicts the state diagram of a deterministic finite automaton for the RE 1^* $(2\ 2 \mid 2\ 3\ 4 \mid 4\ 4)$ (i.e., all sequences of zero or more 1's followed by 2 2, 2 3 4, or 4 4). Following [7], we use double circles to indicate an accept state and $>$ to emphasize the start state (a) of the automaton. For brevity, we will simply use "automaton" as a synonym for "deterministic finite automaton" in the remainder of the paper.

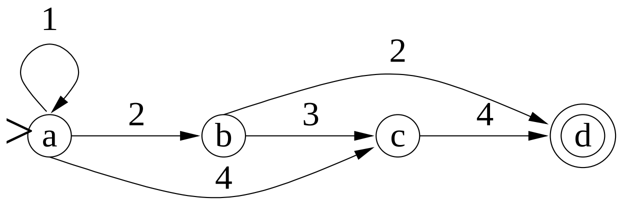


Figure 1: Deterministic finite automaton for the RE 1^* $(2\ 2 \mid 2\ 3\ 4 \mid 4\ 4)$.

3.2 Problem Statement

Given an input database of sequences, we define a sequential pattern to be *frequent* if its support in the database exceeds a user-specified minimum support threshold. Prior work has focused on efficient techniques for the discovery of frequent patterns, typically ignoring the possibility of allowing and exploiting flexible structural constraints during the mining process (Section 2). In this paper, we develop novel, efficient algorithms for mining frequent sequential patterns in the presence of user-specified RE constraints. To simplify the presentation, most of our discussion in the sections that follow focuses on the case of *sequences of simple items* with *no maximum distance constraints*. Section 5 shows how our schemes can be extended to handle itemset sequences and distance constraints for pattern occurrences. The following definitions establish some useful terminology for our discussion.

[2]Note that our definition is slightly different from that of Srikant and Agrawal [12] who consider the δ bound to be on the "transaction time" difference of adjacent element occurrences. Nevertheless, the two definitions are equivalent for all practical purposes.

Definition 3.1 A sequence s is said to be *legal with respect to state* b of automaton $\mathcal{A}_{\mathcal{R}}$ if every state transition in $\mathcal{A}_{\mathcal{R}}$ is defined when following the sequence of transitions for the elements of s from b.

Definition 3.2 A sequence s is said to be *valid with respect to state* b of automaton $\mathcal{A}_{\mathcal{R}}$ if s is legal with respect to b *and* the final state of the transition path from b on input s is an *accept* state of $\mathcal{A}_{\mathcal{R}}$. We say that s is *valid* if s is valid with respect to the start state a of $\mathcal{A}_{\mathcal{R}}$ (or, equivalently, if s is accepted by $\mathcal{A}_{\mathcal{R}}$).

Informally, a sequence is legal (resp. valid) with respect to some state if its list of elements defines a proper transition path (resp. transition path to an accept state) in the automaton, starting from that state. The following example helps in illustrating the above definitions.

Example 3.1: Consider the RE constraint $\mathcal{R} = 1^* \ (2\ 2 \mid 2\ 3\ 4 \mid 4\ 4)$ and the automaton $\mathcal{A}_{\mathcal{R}}$, shown in Figure 1. Sequence $< 1\ 2\ 3 >$ is legal with respect to state a and sequence $< 3\ 4 >$ is legal with respect to state b, while sequences $< 1\ 3\ 4 >$ and $< 2\ 4 >$ are not legal with respect to any state of $\mathcal{A}_{\mathcal{R}}$. Similarly, sequence $< 3\ 4 >$ is valid with respect to state b (since $b \stackrel{<3\ 4>}{\Longrightarrow} d$ and d is an accept state), however it is not valid, since it is not valid with respect to the start state a of $\mathcal{A}_{\mathcal{R}}$. Examples of valid sequences include $< 1\ 1\ 2\ 2 >$ and $< 2\ 3\ 4 >$. ∎

Having established the necessary notions and terminology, we can now provide an abstract definition of our constrained pattern mining problem as follows.

- **Given:** A database of sequences $\mathcal{D}$, a user-specified minimum support threshold, and a user-specified RE constraint $\mathcal{R}$ (or, equivalently, an automaton $\mathcal{A}_{\mathcal{R}}$).
- **Find:** All *frequent and valid* sequential patterns in $\mathcal{D}$.

Thus, our objective is to efficiently mine patterns that are not only frequent but also belong to the language of sequences generated by the RE $\mathcal{R}$[3]. To this end, the next section introduces the SPIRIT family of mining algorithms for pushing user-specified RE constraints to varying degrees inside the pattern mining process.

4 Mining Frequent and Valid Sequences: The SPIRIT Algorithms

4.1 Overview

Figure 2 depicts the basic algorithmic skeleton of the SPIRIT family, using an input parameter $\mathcal{C}$ to denote a generic user-specified constraint on the mined patterns. The output of a SPIRIT algorithm is the set of frequent sequences in the database $\mathcal{D}$ that satisfy constraint $\mathcal{C}$. At a high level, our algorithmic framework is similar in structure to the general Apriori strategy of Agrawal and Srikant [2]. Basically, SPIRIT algorithms work in passes, with each pass resulting in the discovery of longer patterns. In the k^{th} pass, a set of candidate (i.e., potentially frequent and valid) k-sequences C_k is generated and pruned using information from earlier passes. A scan over the data is then made, during which the support for each candidate sequence in C_k is counted and F_k is populated with the frequent k-sequences in C_k. There are, however, two crucial differences between the SPIRIT framework and conventional Apriori-type schemes (like GSP [12]) or the Constrained APriori (CAP) algorithm [10] for mining associations with anti-monotone and/or succinct constraints.

[3]Our algorithms can readily handle a *set* of RE constraints by collapsing them into a single RE [7].

1. *Relaxing $\mathcal{C}$ by inducing a weaker (i.e., less restrictive) constraint $\mathcal{C}'$ (Step 1).* Intuitively, constraint $\mathcal{C}'$ is *weaker* than $\mathcal{C}$ if every sequence that satisfies $\mathcal{C}$ also satisfies $\mathcal{C}'$. The "strength" of $\mathcal{C}'$ (i.e., how closely it emulates $\mathcal{C}$) essentially determines the degree to which the user-specified constraint $\mathcal{C}$ is pushed inside the pattern mining computation. The choice of $\mathcal{C}'$ differentiates among the members of the SPIRIT family and leads to interesting tradeoffs that are discussed in detail later in this section.

2. *Using the relaxed constraint $\mathcal{C}'$ in the candidate generation and candidate pruning phases of each pass.* SPIRIT algorithms maintain the set F of frequent sequences (up to a given length) that satisfy the relaxed constraint $\mathcal{C}'$. Both F and $\mathcal{C}'$ are used in:

    (a) the candidate generation phase of pass k (Step 6), to produce an initial set of candidate k-sequences C_k that satisfy $\mathcal{C}'$ by appropriately extending or combining sequences in F; and,

    (b) the candidate pruning phase of pass k (Steps 8-9), to delete from C_k all candidate k-sequences containing at least one subsequence that satisfies $\mathcal{C}'$ and does not appear in F.

Thus, a SPIRIT algorithm maintains the following *invariant*: at the end of pass k, F_k is exactly the set of all frequent k-sequences that satisfy the constraint $\mathcal{C}'$. Note that incorporating $\mathcal{C}'$ in candidate generation and pruning also impacts the terminating condition for the **repeat** loop in Step 15. Finally, since at the end of the loop, F contains frequent patterns satisfying the induced relaxed constraint $\mathcal{C}'$, an additional filtering step may be required (Step 17).

Procedure SPIRIT($\mathcal{D}$, $\mathcal{C}$)
begin
1. let $\mathcal{C}' :=$ a constraint *weaker* (i.e., less restrictive) than $\mathcal{C}$
2. $F := F_1 :=$ frequent items in $\mathcal{D}$ that satisfy $\mathcal{C}'$
3. $k := 2$
4. **repeat** {
5.   // *candidate generation*
6.   using $\mathcal{C}'$ and F generate $C_k :=$ { potentially frequent k-sequences that satisfy $\mathcal{C}'$ }
7.   // *candidate pruning*
8.   let $P := \{s \in C_k : s$ has a subsequence t that satisfies $\mathcal{C}'$ and $t \notin F\ \}$
9. $C_k := C_k - P$
10.   // *candidate counting*
11.   scan $\mathcal{D}$ counting the support for candidate k-sequences in C_k
12. $F_k :=$ frequent sequences in C_k
13. $F := F \cup F_k$
14. $k := k + 1$
15. } **until** TerminatingCondition(F , $\mathcal{C}'$) holds
16. // *enforce the original (stronger) constraint $\mathcal{C}$*
17. output sequences in F that satisfy $\mathcal{C}$

end

Figure 2: The SPIRIT constrained pattern mining framework.

Given a set of candidate k-sequences C_k, counting support for the members of C_k (Step 11) can be performed efficiently by employing specialized search structures, like the *hash tree* [12], for organizing the candidates. The implementation details can be found in [12]. The candidate counting step is typically the most expensive step of the pattern mining process and its overhead is directly proportional to the size of C_k [12]. Thus, at an abstract level, the goal of an efficient pattern mining strategy is to employ the minimum support requirement and any additional user-specified constraints to restrict as much as possible the set of candidate k-sequences counted during pass k. The SPIRIT framework strives to achieve this goal by using two different types of pruning within each pass k.

- *Constraint-based pruning* using a relaxation $\mathcal{C}'$ of the user-specified constraint $\mathcal{C}$; that is, ensuring that all candidate k-sequences in C_k satisfy $\mathcal{C}'$. This is accomplished by appropriately employing $\mathcal{C}'$ and F in the candidate generation phase (Step 6).

- *Support-based pruning*; that is, ensuring that all subsequences of a sequence s in C_k that satisfy $\mathcal{C}'$ are present in the current set of discovered frequent sequences F (Steps 8-9). Note that, even though *all* subsequences of s must in fact be frequent, we can only check the minimum support constraint for subsequences that satisfy $\mathcal{C}'$, since only these are retained in F.

Intuitively, constraint-based pruning tries to restrict C_k by (partially) enforcing the input constraint $\mathcal{C}$, whereas support-based pruning tries to restrict C_k by checking the minimum support constraint for qualifying subsequences. Note that, given a set of candidates C_k and a relaxation $\mathcal{C}'$ of $\mathcal{C}$, the amount of support-based pruning is maximized when $\mathcal{C}'$ is *anti-monotone* [10] (i.e., all subsequences of a sequence satisfying $\mathcal{C}'$ are guaranteed to also satisfy $\mathcal{C}'$). This is because support information for *all* of the subsequences of a candidate sequence s in C_k can be used to prune it. However, when $\mathcal{C}'$ is *not* anti-monotone, the amounts of constraint-based and support-based pruning achieved vary depending on the specific choice of $\mathcal{C}'$.

Pushing Non Anti-Monotone Constraints Inside the Mining Process. Consider the general problem of mining all frequent sequences that satisfy a user-specified constraint $\mathcal{C}$. If $\mathcal{C}$ is anti-monotone, then the most effective way of using $\mathcal{C}$ to prune candidates is to push $\mathcal{C}$ "all the way" inside the mining computation. In the context of the SPIRIT framework, this means using $\mathcal{C}$ *as is* (rather than some relaxation of $\mathcal{C}$) in the pattern discovery loop. The optimality of this solution for anti-monotone $\mathcal{C}$ stems from two observations. First, using $\mathcal{C}$ clearly maximizes the amount of constraint-based pruning since the strongest possible constraint (i.e., $\mathcal{C}$ itself) is employed. Second, since $\mathcal{C}$ is anti-monotone, all subsequences of a frequent candidate k-sequence that survives constraint-based pruning are guaranteed to be in F (since they also satisfy $\mathcal{C}$). Thus, using the full strength of an anti-monotone constraint $\mathcal{C}$ maximizes the effectiveness of constraint-based pruning as well as support-based pruning. Note that this is exactly the methodology used in the CAP algorithm [10] for anti-monotone itemset constraints. An additional benefit of using anti-monotone constraints is that they significantly simplify the candidate generation and candidate pruning tasks. More specifically, generating C_k is nothing but an appropriate "self-join" operation over F_{k-1} and determining the pruned set P (Step 8) is simplified by the fact that all subsequences of candidates are guaranteed to satisfy the constraint.

When $\mathcal{C}$ is *not* anti-monotone, however, things are not that clear-cut. A simple solution, suggested by Ng et al. [10] for itemset constraints, is to take an anti-monotone relaxation of $\mathcal{C}$ and use that relaxation for candidate pruning. Nevertheless, this simple approach may not always be feasible. For example, our RE constraints for sequences do not admit any non-trivial anti-monotone relaxations. In such cases, the degree to which the constraint $\mathcal{C}$ is pushed inside the mining process (i.e., the strength of the (non anti-monotone) relaxation $\mathcal{C}'$ used for pruning) impacts the effectiveness of both constraint-based pruning and support-based pruning in different ways. More specifically, while increasing the strength of $\mathcal{C}'$ obviously increases the effectiveness of constraint-based pruning, it can also have a negative effect on support-based pruning. The reason is that, for any given sequence in C_k that survives constraint-based pruning, *the number of its subsequences that satisfy the stronger, non anti-monotone constraint $\mathcal{C}'$ may decrease.* Again, note that only subsequences that satisfy $\mathcal{C}'$ can be used for support-based pruning, since this is the only "state" maintained from previous passes (in F).

Pushing a non anti-monotone constraint $\mathcal{C}'$ in the pattern discovery loop can also increase the computational complexity of the candidate generation and pruning tasks. For candidate generation, the fact that $\mathcal{C}'$ is not anti-monotone means that some (or, all) of a candidate's subsequences may be absent from F. In some cases, a "brute-force" approach (based on just $\mathcal{C}'$) may be required to generate an initial set of candidates C_k. For candidate pruning, computing the subsequences of a candidate that satisfy $\mathcal{C}'$ may no longer be trivial, implying additional computational overhead. We should note, however, that candidate generation and pruning are inexpensive CPU-bound operations that typically constitute only a small fraction of the overall computational cost. This fact is also clearly demonstrated in our experimental results (Section 6). Thus, the major tradeoff that needs to be considered when choosing a specific $\mathcal{C}'$ from among the spectrum of possible relaxations of $\mathcal{C}$ is the extent to which that choice impacts the effectiveness of constraint-based and support-based pruning. The objective, of course, is to strike a reasonable balance between the two different types of pruning so as to minimize the number of candidates for which support is actually counted in each pass.

The SPIRIT Algorithms. The four SPIRIT algorithms for constrained pattern mining are points spanning the entire spectrum of relaxations for the user-specified RE constraint $\mathcal{C} \equiv \mathcal{R}$. Essentially, the four algorithms represent a natural progression, with each algorithm pushing a stronger relaxation of $\mathcal{R}$ than its predecessor in the pattern mining loop [4]. The first SPIRIT algorithm, termed SPIRIT(N) ("N" for Naive), employs the weakest relaxation of $\mathcal{R}$ – it only prunes candidate sequences containing elements that do not appear in $\mathcal{R}$. The second algorithm, termed SPIRIT(L) ("L" for Legal), requires every candidate sequence to be *legal* with respect to some state of $\mathcal{A}_{\mathcal{R}}$. The third algorithm, termed SPIRIT(V) ("V" for Valid), goes one step further by filtering out candidate sequences that are not *valid with respect to any state of* $\mathcal{A}_{\mathcal{R}}$. Finally, the SPIRIT(R) algorithm ("R" for Regular) essentially pushes $\mathcal{R}$ "all the way" inside the mining process by counting support only for *valid* candidate sequences, i.e., sequences accepted by $\mathcal{A}_{\mathcal{R}}$. Table 2 summarizes the constraint choices for the four members of the SPIRIT family within the general framework depicted in Figure 2. Note that, of the four SPIRIT algorithms, SPIRIT(N) is the only one employing an anti-monotone (and, trivial) relaxation $\mathcal{C}'$. Also, note that the progressive increase in the strength of $\mathcal{C}'$ implies a subset relationship between the frequent sequences determined for each pass k; that is,

$$F_k^{SPIRIT(R)} \subseteq F_k^{SPIRIT(V)} \subseteq F_k^{SPIRIT(L)} \subseteq F_k^{SPIRIT(N)}.$$

Algorithm	**Input Constraint $\mathcal{C}$**	**Relaxed Constraint $\mathcal{C}'$**
SPIRIT(N)	RE constraint $\mathcal{R}$	all elements appear in $\mathcal{R}$
SPIRIT(L)	RE constraint $\mathcal{R}$	legal wrt some state of $\mathcal{A}_{\mathcal{R}}$
SPIRIT(V)	RE constraint $\mathcal{R}$	valid wrt some state of $\mathcal{A}_{\mathcal{R}}$
SPIRIT(R)	RE constraint $\mathcal{R}$	valid, i.e., $\mathcal{C}' \equiv \mathcal{R}$

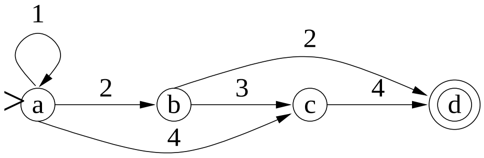
Table 2: The four SPIRIT algorithms.

Example 4.1: Consider the set of sequences $\mathcal{D}$ shown in Figure 3(a) and the automaton $\mathcal{A}_{\mathcal{R}}$ depicted in Figure 3(b) for $\mathcal{R} =$ 1* (2 2 | 2 3 4 | 4 4) (from Example 3.1). Let the minimum support threshold be 0.4; thus, a frequent sequence must be contained in at least two sequences in the data set. Figures 3(c),

[4] The development of the SPIRIT algorithms is based on the equivalent automaton form $\mathcal{A}_{\mathcal{R}}$ of the user-specified RE constraint $\mathcal{R}$. Algorithms for constructing $\mathcal{A}_{\mathcal{R}}$ from $\mathcal{R}$ can be found in the theory literature [7].

Data Set $\mathcal{D}$
< 1 2 3 2 >
< 1 1 2 2 >
< 2 4 3 4 >
< 2 3 4 3 >
< 1 1 2 3 >

(a) Data set.



(b) Automaton $\mathcal{A}_{\mathcal{R}}$.

F_2
< 1 1 >
< 1 2 >
< 1 3 >
< 2 2 >
< 2 3 >
< 2 4 >
< 3 4 >
< 4 3 >

C_3	Count
< 1 1 1 >	0
< 1 1 2 >	2
< 1 1 3 >	1
< 1 2 2 >	2
< 1 2 3 >	2
< 2 2 2 >	0
< 2 2 3 >	0
< 2 2 4 >	0
< 2 3 4 >	2
< 2 4 3 >	2

F_3
< 1 1 2 >
< 1 2 2 >
< 1 2 3 >
< 2 3 4 >
< 2 4 3 >

C_4	Count
< 1 1 2 2 >	1
< 1 1 2 3 >	1

(c) SPIRIT(N).

State	F_2
a	< 1 1 >
a	< 1 2 >
a	< 2 2 >
a	< 2 3 >
b	< 3 4 >

State	C_3	Count
a	< 1 1 1 >	0
a	< 1 1 2 >	2
a	< 1 2 2 >	2
a	< 1 2 3 >	2
a	< 2 3 4 >	2

State	F_3
a	< 1 1 2 >
a	< 1 2 2 >
a	< 1 2 3 >
a	< 2 3 4 >

State	C_4	Count
a	< 1 1 2 2 >	1
a	< 1 1 2 3 >	1

(d) SPIRIT(L).

State	F_2
a	< 2 2 >
b	< 3 4 >

State	C_3	Count
a	< 1 2 2 >	2
a	< 2 3 4 >	2

State	F_3
a	< 1 2 2 >
a	< 2 3 4 >

State	C_4	Count
a	< 1 1 2 2 >	1
a	< 1 2 3 4 >	0

(e) SPIRIT(V).

F_2
< 2 2 >

C_3	Count
< 1 2 2 >	2
< 2 3 4 >	2

F_3
< 1 2 2 >
< 2 3 4 >

C_4	Count
< 1 1 2 2 >	1
< 1 2 3 4 >	0

(f) SPIRIT(R).

Figure 3: Candidates generated by SPIRIT algorithms for $\mathcal{R} = 1^* \ (2\ 2 \mid 2\ 3\ 4 \mid 4\ 4)$.

3(d), 3(e), and 3(f) illustrate the sets of candidate sequences C_k for which support is computed by each of the four SPIRIT algorithms. (The details of the candidate generation and pruning phases are presented in the remainder of this section.) We also show the support counts for each candidate in C_k and the frequent sequences in F_k. (For sequences generated by SPIRIT(L) and SPIRIT(V), the corresponding state of $\mathcal{A}_{\mathcal{R}}$ is also specified.)

Note that, even though the frequent sets F_k obviously satisfy the subset relationship mentioned above, the same does not necessarily hold for the candidate sets C_k. For instance, in Figure 3, both SPIRIT(V) and SPIRIT(R) generate the sequence $< 1\ 2\ 3\ 4 >$ that is not generated by either SPIRIT(N) or SPIRIT(L). This is a direct consequence of the constraint-based vs. support-based pruning tradeoff for non anti-monotone constraints. ∎

The remainder of this section provides a detailed discussion of the candidate generation and candidate pruning phases for each of the SPIRIT algorithms. Appropriate terminating conditions (Step 15) are also presented. The quantitative study of the constraint-based vs. support-based pruning tradeoff for the SPIRIT algorithms is deferred until the presentation of our experimental results (Section 6).

4.2 The SPIRIT(N) Algorithm

SPIRIT(N) is a simple modification of the GSP algorithm [12] for mining sequential patterns. SPIRIT(N) simply requires that all elements of a candidate sequence s in C_k appear in the RE $\mathcal{R}$. This constraint is clearly anti-monotone, so candidate generation and pruning are performed exactly as in GSP [12].

Candidate Generation. For every pair of $(k-1)$-sequences s and t in F_{k-1}, if $s_{j+1} = t_j$ for all $1 \le j \le k-2$, then $< s\ t_{k-1} >$ is added to C_k. This is basically a self-join of F_{k-1}, the join attributes being the last $k-2$ elements of the first sequence and the first $k-2$ elements of the second.

Candidate Pruning. A candidate sequence s is pruned from C_k if at least one of its $(k-1)$-subsequences does not belong to F_{k-1}.

Terminating Condition. The set of frequent k-sequences, F_k, is empty.

4.3 The SPIRIT(L) Algorithm

SPIRIT(L) uses the automaton $\mathcal{A}_{\mathcal{R}}$ to prune from C_k candidate k-sequences that are not *legal* with respect to any state of $\mathcal{A}_{\mathcal{R}}$. In our description of SPIRIT(L), we use $F_k(b)$ to denote the set of frequent k-sequences that are legal with respect to state b of $\mathcal{A}_{\mathcal{R}}$.

Candidate Generation. For each state b in $\mathcal{A}_{\mathcal{R}}$, we add to C_k candidate k-sequences that are legal with respect to b and have the potential to be frequent.

Lemma 4.2: Consider a k-sequence s that is legal with respect to state b in $\mathcal{A}_{\mathcal{R}}$, where $b \xrightarrow{s_1} c$ is a transition in $\mathcal{A}_{\mathcal{R}}$. For s to be frequent, $< s_1 \cdots s_{k-1} >$ must be in $F_{k-1}(b)$ and $< s_2 \cdots s_k >$ must be in $F_{k-1}(c)$. ∎

Thus, the candidate sequences for state b can be computed as follows. For every sequence s in $F_{k-1}(b)$, if $b \xrightarrow{s_1} c$ is a transition in $\mathcal{A}_{\mathcal{R}}$, then for every sequence t in $F_{k-1}(c)$ such that $s_{j+1} = t_j$ for all $1 \le j \le k-2$, the candidate sequence $< s\ t_{k-1} >$ is added to C_k. This is basically a join of $F_{k-1}(b)$ and $F_{k-1}(c)$, on the condition that the $(k-2)$-length suffix of $s \in F_{k-1}(b)$ matches the $(k-2)$-length prefix of $t \in F_{k-1}(c)$ and $b \xrightarrow{s_1} c$ is a transition in $\mathcal{A}_{\mathcal{R}}$.

Candidate Pruning. Given a sequence s in C_k, the candidate generation step ensures that both its prefix and suffix of length $k-1$ are frequent. We also know that in order for s to be frequent, every subsequence of s must also be frequent. However, since we only count support for sequences that are legal with respect to some state of $\mathcal{A}_{\mathcal{R}}$, we can prune s from C_k only if we find a *legal* subsequence of s that is not frequent (i.e., not in F). The candidate pruning procedure computes the set of maximal subsequences of s with length less than k that are legal with respect to some state of automaton $\mathcal{A}_{\mathcal{R}}$. If any of these maximal subsequences is not contained in F, then s is deleted from C_k.

Example 4.3: Consider the generation of candidate set C_4 in Figure 3(d). For state a, $F_3(a) = \{< 1\ 1\ 2 >, < 1\ 2\ 2 >, < 1\ 2\ 3 >, < 2\ 3\ 4 >\}$. For sequence $< 1\ 1\ 2 >$, the transition $a \xrightarrow{1} a$ is in $\mathcal{A}_{\mathcal{R}}$. Thus, since the first two elements of $< 1\ 2\ 2 >$ and $< 1\ 2\ 3 >$ match the last two elements of $< 1\ 1\ 2 >$, the sequences $< 1\ 1\ 2\ 2 >$ and $< 1\ 1\ 2\ 3 >$ are added to C_4. Similarly, sequence $< 1\ 2\ 3\ 4 >$ is also added to C_4 in the candidate generation step, but it is pruned in the candidate pruning step. This is because $< 1\ 2\ 3\ 4 >$ has a maximal legal subsequence (i.e., $< 1\ 4 >$) that is not frequent. ∎

Procedure FINDMAXSUBSEQ(*Start*, *End*, s)
begin
  for each state b in automaton $\mathcal{A}_{\mathcal{R}}$ **do**
    maxSeq[b] := $\emptyset$
  for $l := |s|$ **down to** 1 **do** {
    for each state b in automaton $\mathcal{A}_{\mathcal{R}}$ **do** {
      tmpSeq[b] = $\emptyset$
      if (there exists a transition $b \xrightarrow{s_l} c$ in $\mathcal{A}_{\mathcal{R}}$) {
        if ($c \in$ *End*) tmpSeq[b] := $\{s_l\}$
        tmpSeq[b] := tmpSeq[b] $\cup \{< s_l\ t >: t \in$ maxSeq[c]$\}$
      }
    }
    for each state b in automaton $\mathcal{A}_{\mathcal{R}}$ **do** {
      maxSeq[b] := maxSeq[b] $\cup$ tmpSeq[b]
      for each sequence t in maxSeq[b] **do**
        if (there exists a sequence u in maxSeq[b] $- \{< s_l \cdots s_{|s|} >\}$ such that t is a subsequence of u)
          delete t from maxSeq[b]
    }
  }
  return $\bigcup_{b \in Start}$ maxSeq[b] - $\{s\}$ (after deleting non-maximal sequences)
end

Figure 4: Algorithm for finding maximal subsequences.

We now describe an algorithm for computing the maximal legal subsequences of a candidate sequence s. Let maxSeq(b, s) denote the set of maximal subsequences of s that are legal with respect to state b of $\mathcal{A}_{\mathcal{R}}$. Then, if we let $t = < s_2 \cdots s_{|s|} >$, maxSeq($b$, s) can be computed from maxSeq(b, t) using the following relationship:

$$\text{maxSeq}(b,s) \subseteq \begin{cases} \text{maxSeq}(b,t) \cup \{< s_1\ u >: u \in \text{maxSeq}(c,t)\} \cup \{s_1\} & \text{if } b \xrightarrow{s_1} c \text{ is a transition in } \mathcal{A}_{\mathcal{R}} \\ \text{maxSeq}(b,t) & \text{otherwise} \end{cases}$$

The intuition is that for a subsequence $v \in \text{maxSeq}(b,s)$, either (a) v does not involve s_1, in which case v is a maximal subsequence of t that is legal with respect to b, or (b) $v_1 = s_1$ and $< v_2 \cdots v_{|v|} >$ is a maximal subsequence of t with respect to state c. Based on the above recurrence, we propose a *dynamic programming* algorithm, termed FINDMAXSUBSEQ, for computing maxSeq(b, s) for all states b of $\mathcal{A}_{\mathcal{R}}$

l	maxSeq[a]	maxSeq[b]	maxSeq[c]
4	$\{<4>\}$		$\{<4>\}$
3	$\{<4>\}$	$\{<3>, <3\ 4>\}$	$\{<4>\}$
2	$\{<4>, <2\ 3>, <2\ 3\ 4>\}$	$\{<2>, <3\ 4>\}$	$\{<4>\}$
1	$\{<1\ 4>, <1\ 2\ 3>, <2\ 3\ 4>, <1\ 2\ 3\ 4>\}$	$\{<2>, <3\ 4>\}$	$\{<4>\}$

Figure 5: Execution of FINDMAXSUBSEQ for $s =< 1\ 2\ 3\ 4 >$.

(Figure 4). Intuitively, FINDMAXSUBSEQ works by computing the set maxSeq for successively longer suffixes of the input sequence s, beginning with the suffix consisting of only the last element of s.

More specifically, given an input sequence s and two sets of states in $\mathcal{A}_\mathcal{R}$ (*Start* and *End*), algorithm FINDMAXSUBSEQ returns the set of all maximal subsequences t of s such that (a) the length of t is less than $|s|$, and (b) t is legal with respect to a state b in *Start* and if $b \overset{t}{\Rightarrow} c$, then $c \in End$. In each iteration of the for loop spanning Steps 3–17, for each state b in $\mathcal{A}_\mathcal{R}$, maximal legal subsequences for the suffix $< s_l \cdots s_{|s|} >$ are computed and stored in maxSeq[b]. At the start of the l^{th} iteration, maxSeq[b] contains the maximal subsequences of $< s_{l+1} \cdots s_{|s|} >$ that are both legal with respect to state b and result in a state in *End*. Thus, if a transition from b to c on element s_l is in $\mathcal{A}_\mathcal{R}$, then the maximal legal subsequences for b comprise those previously computed for $< s_{l+1} \cdots s_{|s|} >$ and certain new sequences involving element s_l. These new sequences containing s_l are computed in the body of the for loop spanning Steps 5–9 and stored in tmpSeq[b]. A point to note is that, since we are only interested in maximal legal subsequences that result in a state in *End*, we add s_l to tmpSeq[b] only if $c \in End$ (Step 7).

After the new maximal subsequences involving s_l are stored in tmpSeq[b] for every state b of $\mathcal{A}_\mathcal{R}$, they are added to maxSeq[b], following which, non-maximal subsequences in maxSeq[b] are deleted (Steps 11–16)[5]. Finally, after maximal legal subsequences for the entire sequence s have been computed for all the states of $\mathcal{A}_\mathcal{R}$, only those for states in *Start* are returned (Step 18).

To recap, the candidate pruning procedure of SPIRIT(L) invokes FINDMAXSUBSEQ to determine all the maximal legal subsequences of each candidate s in C_k, and deletes s from C_k if any of these subsequences is not frequent. For SPIRIT(L), algorithm FINDMAXSUBSEQ is invoked with *Start* and *End* both equal to the set of all states in $\mathcal{A}_\mathcal{R}$.

Example 4.4: Figure 5 illustrates the maxSeq set for the various states of automaton $\mathcal{A}_\mathcal{R}$ (from Figure 3(b)) and for decreasing values of l, when FINDMAXSUBSEQ is invoked by SPIRIT(L) with $s =< 1\ 2\ 3\ 4 >$. Consider the final iteration, i.e., $l = 1$. At the start of the iteration maxSeq[a] contains the sequences $< 4 >, < 2\ 3 >$, and $< 2\ 3\ 4 >$. Since $a \overset{1}{\rightarrow} a$, sequences $< 1 >, < 1\ 4 >, < 1\ 2\ 3 >$, and $< 1\ 2\ 3\ 4 >$ are added to maxSeq[a] (Steps 7–8). Of these, sequences $< 1 >, < 4 >$, and $< 2\ 3 >$ are deleted from maxSeq[a] since they are subsequences of $< 1\ 4 >$ and $< 1\ 2\ 3 >$ (Steps 14–15). The remaining subsequences stay in maxSeq[a], since $< 1\ 2\ 3\ 4 >$ cannot be used to prune non-maximal subsequences. Consequently, the final set of maximal legal subsequences returned by FINDMAXSUBSEQ is $\{< 1\ 4 >, < 1\ 2\ 3 >, < 2\ 3\ 4 >\}$. ∎

Note that, given the candidate $s =< s_1 \cdots s_k > \in C_k$, algorithm FINDMAXSUBSEQ actually needs to check only the legal subsequences of s that start with s_1 *and* end with s_k. This is because all other

[5] In Steps 13–15, we have to be careful not to consider $< s_l \cdots s_{|s|} >$ to delete other sequences in maxSeq[b] since we are interested in maximal sequences whose length is less than $|s|$. If we were to use $< s_l \cdots s_{|s|} >$ to prune other subsequences, then it is possible that maxSeq[b] for a state b may only contain the sequence s which has length $|s|$ and other maximal subsequences of length less than $|s|$ may have been pruned by it.

legal subsequences of s are also legal subsequences of either $< s_1 \cdots s_{k-1} >$ or $< s_2 \cdots s_k >$ which are themselves frequent and legal (by the Candidate Generation process). Thus, a possible optimization for SPIRIT(L) is to invoke algorithm FINDMAXSUBSEQ with $Start = \{b :$ the transition $b \xrightarrow{s_1} d$ is in $\mathcal{A}_{\mathcal{R}}\}$ and $End = \{c :$ the transition $d \xrightarrow{s_k} c$ is in $\mathcal{A}_{\mathcal{R}}\}$. (In the above example, this optimization implies that only subsequence $< 1\ 4 >$ would be returned.)

Terminating Condidition. The set of frequent k-sequences that are legal with respect to the start state a of $\mathcal{A}_{\mathcal{R}}$ is empty; that is, $F_k(a)$ is empty.

Time Complexity. Consider the candidate pruning overhead for a candidate k-sequence s in C_k. Compared to the candidate pruning step of SPIRIT(N), which has a time complexity of $O(k)$ (to determine the k subsequences of s), the computational overhead of candidate pruning in SPIRIT(L) can be significantly higher. More specifically, the worst-case time complexity of computing the maximal legal subsequences of s using algorithm FINDMAXSUBSEQ can be shown to be $O(k^2 * |\mathcal{A}_{\mathcal{R}}| * |\text{maxSeq}(s)|)$, where $|\mathcal{A}_{\mathcal{R}}|$ is the number of states in $\mathcal{A}_{\mathcal{R}}$ and $|\text{maxSeq}(s)|$ is the number of maximal legal subsequences for s. To see this, note that the outermost **for** loop in Step 3 of FINDMAXSUBSEQ is executed k times. The time complexity of the first **for** loop in Step 4 is $O(|\mathcal{A}_{\mathcal{R}}| * |\text{maxSeq}(s)|)$, while that of the second **for** loop in Step 11 is $O(k * |\mathcal{A}_{\mathcal{R}}| * |\text{maxSeq}(s)|)$, since maxSeq[$b$] can be implemented as a trie, for which insertions, deletions, and subsequence checking for k-sequences can all be carried out in $O(k)$ time.

We must point out that the higher time complexity of candidate pruning in SPIRIT(L) is not a major efficiency concern since (a) the overhead of candidate generation and pruning is typically a tiny fraction of the cost of counting supports for candidates in C_k, and (b) in practice, $|\text{maxSeq}(s)|$ can be expected to be small for most sequences. In the worst case, however, for a k-sequence, $|\text{maxSeq}(s)|$ can be $O(2^k)$. This worst case scenario can be avoided by imposing an a-priori limit on the size of maxSeq[b] in FINDMAXSUBSEQ and using appropriate heuristics for selecting *victims* (to be ejected from maxSeq[b]) when its size exceeds that limit.

Space Overhead. SPIRIT(N) only utilizes F_{k-1} for the candidate generation and pruning phases during the k^{th} pass. In contrast, the candidate pruning step of SPIRIT(L) requires F to be stored in main memory since the maximal legal subsequences of a candidate k-sequence may be of any length less than k. However, this should not pose a serious problem since each F_k computed by SPIRIT(L) contains only frequent and legal k-sequences, which are typically few compared to all frequent k-sequences. In addition, powerful servers with several gigabytes of memory are now fairly commonplace. Thus, in most cases, it should be possible to accommodate all the sequences in F in main memory. In the occasional event that F does not fit in memory, one option would be to only store $F_{k-l}, \ldots, F_{k-1}$ for some $l \geq 1$. Of course, this means that maximal subsequences whose length is less than $k - l$ cannot be used to prune candidates from C_k during the candidate pruning step.

4.4 The SPIRIT(V) Algorithm

SPIRIT(V) uses a stronger relaxed constraint $\mathcal{C}'$ than SPIRIT(L) during candidate generation and pruning. More specifically, SPIRIT(V) requires every candidate sequence to be *valid* with respect to some state of $\mathcal{A}_{\mathcal{R}}$[6]. In our description of SPIRIT(V), we use $F_k(b)$ to denote the set of frequent k-sequences that are valid with respect to state b of $\mathcal{A}_{\mathcal{R}}$.

Candidate Generation. Since every candidate sequence s in C_k is required to be valid with respect to some state b, it must be the case that the $(k-1)$-length suffix of s is both frequent and valid with respect to state c, where $b \xrightarrow{s_1} c$ is a transition in $\mathcal{A}_{\mathcal{R}}$. Thus, given a state b of $\mathcal{A}_{\mathcal{R}}$, the set of potentially

[6] Note that an alternative approach would be to require candidates to be legal with respect to the start state of $\mathcal{A}_{\mathcal{R}}$. This approach is essentially symmetric to SPIRIT(V) and is not explored further in this paper.

frequent and valid k-sequences with respect to b can be generated using the following rule: for every transition $b \xrightarrow{s_i} c$, for every sequence t in $F_{k-1}(c)$, add $< s_i \ t >$ to the set of candidates for state b. The set C_k is simply the union of these candidate sets over all states b of $\mathcal{A}_{\mathcal{R}}$.

Candidate Pruning. The pruning phase of SPIRIT(V) is very similar to that of SPIRIT(L), except that only valid (rather that legal) subsequences of a candidate can be used for pruning. More specifically, given a candidate sequence s in C_k, we compute all maximal subsequences of s that are valid with respect to some state of $\mathcal{A}_{\mathcal{R}}$ and have length less than k. This is done by invoking algorithm FINDMAXSUBSEQ with *Start* equal to the set of all states of $\mathcal{A}_{\mathcal{R}}$ and *End* equal to the set of all *accept* states of $\mathcal{A}_{\mathcal{R}}$. (Again, a possible optimization for SPIRIT(V) is to use $Start = \{b :$ the transition $b \xrightarrow{s_1} d$ is in $\mathcal{A}_{\mathcal{R}}\}$.) If any of these subsequences is not contained in F, then s is deleted from C_k.

Example 4.5: Consider the generation of candidate set C_4 in Figure 3(e). For state a, $F_3(a)$ contains the sequences $< 1\ 2\ 3 >$ and $< 2\ 3\ 4 >$. Since $a \xrightarrow{1} a$ is a transition in $\mathcal{A}_{\mathcal{R}}$, sequences $< 1\ 1\ 2\ 3 >$ and $< 1\ 2\ 3\ 4 >$ are added to C_4 in the candidate generation step. Note that the sequence $< 1\ 2\ 3\ 4 >$ in C_4 in Figure 3(e) is not pruned since it has only one maximal valid subsequence, $< 2\ 3\ 4 >$, which is frequent. The same candidate sequence was deleted in the pruning step of SPIRIT(L) because one of its legal subsequences, $< 1\ 4 >$, was not frequent. ▮

Terminating Condition. The set of frequent k-sequences F_k is empty. Unlike SPIRIT(L), we cannot terminate SPIRIT(V) based on just $F_k(a)$ becoming empty (where a is the start state of $\mathcal{A}_{\mathcal{R}}$). The reason is that, even though there may be no frequent and valid sequences of length k for a, there could still be longer sequences that are frequent and valid with respect to a.

4.5 The SPIRIT(R) Algorithm

SPIRIT(R) essentially pushes the RE constraint $\mathcal{R}$ "all the way" inside the pattern mining computation, by requiring every candidate sequence for which support is counted to be valid (i.e., $\mathcal{C}' \equiv \mathcal{R}$).

Candidate Generation. Since F contains only valid and frequent sequences, there is no efficient mechanism for generating candidate k-sequences other than a "brute force" enumeration using the automaton $\mathcal{A}_{\mathcal{R}}$. The idea is to traverse the states and transitions of $\mathcal{A}_{\mathcal{R}}$ enumerating all paths of length k that begin with the start state and end at an accept state. Obviously, each such path corresponds to a valid k-sequence containing the elements that label the transitions in the path. (The terms "path" and "sequence" are used interchangeably in the following description.)

We employ two optimizations to improve the efficiency of the above exhaustive path enumeration scheme. Our first optimization uses the observation that, if a path of length less than k corresponds to a sequence that is valid but not frequent, then further extending the path is unnecessary since it cannot yield frequent k-sequences. The second optimization involves exploiting *cycles* in $\mathcal{A}_{\mathcal{R}}$ to reduce computation.

Lemma 4.6: Suppose for a path $< t\ u >$ (of length less than k), both t and $< t\ u >$ result in the same state from the start state a. (That is, u corresponds to a cycle in $\mathcal{A}_{\mathcal{R}}$.) Then, if the path $< t\ u\ v >$ obtained as a result of extending $< t\ u >$ with v is to yield a candidate k-sequence, it must be the case that $< t\ v >$ is both frequent and valid. ▮

Consider the generation of candidate k-sequences C_k. Given a path $< t\ u >$ satisfying the assumptions of Lemma 4.6, we only need to extend $< t\ u >$ with sequences v for which $< t\ v >$ belongs to $F_{|<t\ v>|}$ (since the length of $< t\ v >$ is less than k). Algorithm GENCANDIDATES, depicted in Figure 6, uses these observations in the computation of C_k. The algorithm enumerates paths by recursively invoking itself every time it traverses a transition of $\mathcal{A}_\mathcal{R}$. The input parameters to GENCANDIDATES are (1) s, the sequence corresponding to the transitions traversed so far, (2) the current state b, which is also the state that results when the path s is traversed from the start state a of $\mathcal{A}_\mathcal{R}$, and (3) B, the set of states visited when s is traversed starting from a. In order to compute the set of candidates C_k for SPIRIT(R), algorithm GENCANDIDATES is invoked with input parameters $s = \epsilon$ (the empty sequence), $b = a$, and $B = \{a\}$.

Procedure GENCANDIDATES(s, b, B)
begin
1. **for each** transition $b \xrightarrow{w_i} c$ in $\mathcal{A}_\mathcal{R}$ **do** {
2.   **if** ($|s| = k-1$ **and** c is an accept state) $C_k = C_k \cup \{< s\ w_i >\}$
3.   **if** ($|s| \neq k-1$ **and** (c is not an accept state **or** $< s\ w_i > \in F$)) {
4.   **if** ($c \in B$) {
5.   let $s = < t\ u >$, where t is the prefix of s for which $a \stackrel{t}{\Rightarrow} c$
6. $C_k := C_k \cup \{< t\ u\ w_i\ v >: < t\ v > \in F_{k-|u|-1}\}$
7.   }
8.   **else** GENCANDIDATES($< s\ w_i >, c, B \cup \{b\}$)
9.   }
10. }
end

Figure 6: Algorithm for generating candidates for SPIRIT(R).

The first of our two optimizations is performed in Step 3. If $< s\ w_i >$ is a valid sequence that is not frequent, then the edge labeled w_i is not traversed further since no extension of $< s\ w_i >$ can be frequent either. Our second optimization is applied in Steps 4–7. If $< s\ w_i >$ contains a cycle, then edge w_i is not traversed any further. Instead, assuming $s = < t\ u >$, with $< u\ w_i >$ causing the cycle, the candidates that result from extending $< s\ w_i >$ are computed (Step 6).

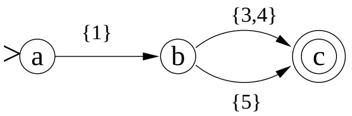
Example 4.7: Consider the generation of candidate set C_4 in Figure 3(f). At the start of the fourth pass, F contains the sequences $< 2\ 2 >$, $< 1\ 2\ 2 >$ and $< 2\ 3\ 4 >$. Since GENCANDIDATES is invoked with parameters ϵ, a and $\{a\}$, for transition $a \xrightarrow{1} a$, the optimization for cycles is used to generate candidates in Steps 5–6. Here, $s = t = \epsilon$ and $w_i = 1$; thus, sequences $v \in F_3$ are appended to w_i to generate candidates. Consequently, $< 1\ 1\ 2\ 2 >$ and $< 1\ 2\ 3\ 4 >$ are added to C_4. ∎

Candidate Pruning. A candidate sequence s in C_k can be pruned if a valid subsequence of s is not frequent. The maximal valid subsequences of s can be computed by invoking algorithm FINDMAXSUBSEQ with *Start* equal to $\{a\}$ and *End* equal to the set of all accept states of $\mathcal{A}_\mathcal{R}$.

Terminating Condition. For some iteration j, $F_j, \ldots, F_{j+|\mathcal{A}_\mathcal{R}|-1}$ are all empty, where $|\mathcal{A}_\mathcal{R}|$ is the number of states in automaton $\mathcal{A}_\mathcal{R}$. To see this, consider any frequent and valid sequence s whose length is greater than $j + |\mathcal{A}_\mathcal{R}| - 1$. Obviously, s contains at least one cycle of length at most $|\mathcal{A}_\mathcal{R}|$ and, therefore, s must contain at least one frequent and valid subsequence of length at least j. However, no valid sequence with length greater than or equal to j is frequent (since $F_j, \ldots, F_{j+|\mathcal{A}_\mathcal{R}|-1}$ are all empty). Thus, s cannot be a frequent and valid sequence.

$\mathcal{D}$
$< \{1,2\}\ \{3\}\ \{4,5\} >$
$< \{1,2\}\ \{3,4\}\ \{4,5\} >$
$< \{2\}\ \{3,4\} >$

(a) Data set.



(b) Automaton for $\{1\}(\{3,4\} \mid \{5\})$.

Frequent and Valid Sequences	Count
$< \{1\}\ \{5\} >$	2
$< \{1\}\ \{4,5\} >$	2
$< \{1,2\}\ \{5\} >$	2
$< \{1,2\}\ \{4,5\} >$	2

(c) Frequent and valid sequences.

Figure 7: Frequent and valid sequences for $\mathcal{R} = \{1\}(\{3,4\} \mid \{5\})$.

5 Extensions: Itemset Sequences and Distance Constraints

5.1 Generalization to Itemset Sequences

Recall from Section 3.1 that an *itemset* sequence is a sequence whose elements are *itemsets* containing one or more items. Further, for a pair of itemset sequences s and t, $s =< s_1 \cdots s_n >$ is a subsequence of t (or, t *contains* s) if $|t| \geq n$ and there exist integers $j_1 < j_2 < \cdots < j_n$ such that $s_1 \subseteq t_{j_1}$, $s_2 \subseteq t_{j_2}$, …, $s_n \subseteq t_{j_n}$. Finally, an itemset sequence is frequent if the fraction of data sequences containing it exceeds the (user-specified) minimum support threshold.

The syntax of RE constraints and the semantics of valid sequences can be naturally extended to the case of sequences with itemset elements. The RE constraint $\mathcal{R}$ for itemset sequences has itemsets (containing at least one item) serving as its basic building blocks. Consequently, transitions between states of automaton $\mathcal{A}_{\mathcal{R}}$ for $\mathcal{R}$ are on itemsets. We define an itemset sequence $s =< s_1 \cdots s_n >$ to be a *restriction* of another sequence $t =< t_1 \cdots t_n >$ containing the same number of elements, if $s_1 \subseteq t_1, \ldots, s_n \subseteq t_n$. Given a RE constraint $\mathcal{R}$, an itemset sequence s is *valid* if some restriction of s satisfies $\mathcal{R}$; that is, there exists a restriction of s that defines an accepting path in the constraint automaton $\mathcal{A}_{\mathcal{R}}$. Our pattern mining problem for itemset sequences is essentially the same as before: find all itemset sequences that are both frequent and valid.

Note the distinction between a restriction and a subsequence of a sequence t – while a subsequence of t can have fewer elements than t, a restriction of t has exactly the same number of elements as t. The notion of a subsequence is used to define the sequence containment and frequency, but the stricter notion of restriction is required to define validity with respect to $\mathcal{R}$ since the length of the valid sequence has to match the length of the accepting path in $\mathcal{A}_{\mathcal{R}}$. We also require every element of a subsequence/restriction of t to be non-empty (i.e., contain at least one item).

Example 5.1: Consider the data set in Figure 7(a), and the RE constraint $\mathcal{R} = \{1\}(\{3,4\} \mid \{5\})$ (corresponding to the automaton $\mathcal{A}_{\mathcal{R}}$ shown in Figure 7(b)). In $\mathcal{A}_{\mathcal{R}}$, $a \xrightarrow{\{1\}} b$ and $a \stackrel{<\{1\}\ \{3,4\}>}{\Longrightarrow} c$. For a minimum support threshold of 0.6 and the above RE constraint, the frequent and valid sequences are as shown in Figure 7(c). The itemset sequence $< \{1,2\}\ \{4,5\} >$ is valid because one of its restrictions $< \{1\}\ \{5\} >$ satisfies $\mathcal{R}$. Furthermore, it is also frequent since it is contained in the following two data sequences: $< \{1,2\}\ \{3\}\ \{4,5\} >$ and $< \{1,2\}\ \{3,4\}\ \{4,5\} >$. Sequences $< \{1,2\}\ \{3\} >$ and $< \{2\}\ \{3,4\} >$ are examples of frequent sequences that are not valid. ∎

Note that our definition of a valid sequence in terms of its restriction is both powerful and general. For instance, we can compute *all* the frequent itemset sequences for the data set in Figure 7(a) by choosing $\mathcal{R}$ to be $(\{1\} \mid \{2\} \mid \{3\} \mid \{4\} \mid \{5\})^*$ since, for this $\mathcal{R}$, *every* itemset sequence is valid (by our definition). In the remainder of this section, we describe in detail how the SPIRIT(L), SPIRIT(V), and SPIRIT(R) algorithms can be extended to handle itemset sequences. (The details for SPIRIT(N) are omitted, since they follow directly from the GSP algorithm of Srikant and Agrawal [12].)

State	F_1
a	$<\{1\}>$
b	$<\{3\}>$
b	$<\{4\}>$
b	$<\{5\}>$

State	F_2
a	$<\{1,2\}>$
a	$<\{1\}\ \{3\}>$
a	$<\{1\}\ \{4\}>$
a	$<\{1\}\ \{5\}>$
b	$<\{3,4\}>$
b	$<\{4,5\}>$

State	F_3
a	$<\{1\}\ \{3,4\}>$
a	$<\{1\}\ \{4,5\}>$
a	$<\{1,2\}\ \{3\}>$
a	$<\{1,2\}\ \{4\}>$
a	$<\{1,2\}\ \{5\}>$

State	F_4
a	$<\{1,2\}\ \{4,5\}>$

Figure 8: Frequent and semi-legal sequences for $\mathcal{R} = \{1\}(\{3,4\} \mid \{5\})$.

5.1.1 SPIRIT(L) for Itemset Sequences

In the k^{th} pass, the set C_k consists of all candidate *k-item sequences* for which support is counted, where a k-item sequence is an itemset sequence containing exactly k items. Given a state b in $\mathcal{A}_{\mathcal{R}}$, we define a sequence s to be *semi-legal* with respect to b if there exists a restriction r of $< s_1 \cdots s_{|s|-1} >$ and a state c in $\mathcal{A}_{\mathcal{R}}$ with the following properties: (1) $b \overset{r}{\Rightarrow} c$, and (2) there is a transition out of c for an itemset w_i such that either $w_i \subseteq s_{|s|}$ or $s_{|s|} \subseteq w_i$. Intuitively, this means that the sequence s either defines or can be extended (by adding items to $s_{|s|}$) to define a path from b in the constraint automaton $\mathcal{A}_{\mathcal{R}}$. During pass k, SPIRIT(L) stores in F_k all the frequent k-item sequences that are *semi-legal* with respect to some state of $\mathcal{A}_{\mathcal{R}}$. (Note that this set clearly subsumes all frequent and *valid* k-item sequences.) Figure 8 depicts the frequent and semi-legal k-item sequences computed by SPIRIT(L) for the data set and RE constraint in Figure 7. Each table in the figure contains, for each sequence, the state of $\mathcal{A}_{\mathcal{R}}$ with respect to which the sequence is semi-legal. In the figure, sequence $< \{1,2\}\ \{3\} >$ is semi-legal with respect to a since $a \overset{<\{1\}>}{\Rightarrow} b$ and $\{3\} \subseteq \{3,4\}$ (for transition $b \overset{\{3,4\}}{\rightarrow} c$). Similarly, sequence $< \{1,2\}\ \{4,5\} >$ is semi-legal with respect to a since $a \overset{<\{1\}>}{\Rightarrow} b$ and $\{5\} \subseteq \{4,5\}$ (for transition $b \overset{\{5\}}{\rightarrow} c$).

The details of the candidate generation and pruning steps for SPIRIT(L) are presented below. As in the case of item sequences, the set of sequences in F_k that are semi-legal with respect to a specific state b of $\mathcal{A}_{\mathcal{R}}$ is denoted by $F_k(b)$.

Candidate Generation. We define a sequence s to be a *restricted prefix* of sequence t if $s_1 = t_1, \ldots, s_{|t|-1} = t_{|t|-1}$ and either (1) $|s| = |t| - 1$, or (2) $|s| = |t|$ and $s_{|t|} \subseteq t_{|t|}$. Then, from our definition of semi-legality, it follows that every k-item sequence s that is semi-legal with respect to state b of automaton $\mathcal{A}_{\mathcal{R}}$ (with a first transition of $b \overset{w_i}{\rightarrow} c$) must satisfy the following two properties:

1. If $k \leq |w_i|$, then s contains exactly one element/itemset s_1 and $s_1 \subseteq w_i$.

2. If $k > |w_i|$, then $w_i \subseteq s_1$ and the suffix $< s_2 \cdots s_{|s|} >$ is semi-legal with respect to state c. Further, if $w_i \subset s_1$ then for any item i not contained in w_i, $< (s_1 - \{i\})\ s_2 \cdots s_{|s|} >$ is also semi-legal with respect to b. Finally, it is easy to see that there always exists a $(k-1)$-item sequence t that is both a restricted prefix of s and semi-legal with respect to b.

In order to illustrate the latter property, consider the 4-item sequence $s = < \{1,2\}\ \{4,5\} >$ from Figure 8 that is semi-legal with repect to state a (due to transition $a \overset{\{1\}}{\rightarrow} b$). Note that the sequence $< \{4,5\} >$ is semi-legal with respect to b. Further, $< \{1\}\ \{4,5\} >$ is a 3-item sequence (resulting from the deletion of item 2 from s_1) that is also semi-legal with respect to state a. Finally, of the two 3-item restricted prefixes of s, $< \{1,2\}\ \{4\} >$ and $< \{1,2\}\ \{5\} >$, the latter is obviously semi-legal with respect to state a.

An implication of the above properties is that every candidate k-item sequence s can be generated from $(k-1)$-item frequent and semi-legal sequences in F_{k-1} that are either (1) restricted prefixes of s, or (2) derivable from s by deleting a single item from s_1. Thus, for every state b in automaton $\mathcal{A}_{\mathcal{R}}$ and for every transition $b \stackrel{w_i}{\rightarrow} c$, we generate candidate k-item sequences that are semi-legal with respect to b based on the following set of rules. (When possible, we provide examples of candidate generation for the data set and RE constraint shown in Figure 7.)

1. $\underline{k \leq |w_i|}$: In this case, all k-subsets of w_i (that is, subsets containing k items) whose $(k-1)$-subsets are in $F_{k-1}(b)$ are added to C_k. As an example, for $k=1$, state b, and transition $b \stackrel{\{3,4\}}{\rightarrow} c$, candidate 1-sequences $<\{3\}>$ and $<\{4\}>$ are added to C_1. Note that, if any of these subsets prove to be infrequent then the transition $b \stackrel{w_i}{\rightarrow} c$ can be safely pruned from the constraint automaton $\mathcal{A}_{\mathcal{R}}$. (As a possible optimization, the support for all itemset labels in $\mathcal{A}_{\mathcal{R}}$ can be counted in a pre-processing step and transitions with infrequent labels can be pruned; of course, this means that all subsets of surviving labels w_i can be directly added to F_k.)

2. $\underline{k > |w_i|}$: In this case, our candidate-generation process produces two types of candidate k-item sequences, based on the aforementioned properties of semi-legality. For the first type, termed *path-extending candidates*, we employ the information stored in frequent sets $F_l(c)$, where $l < k$, to try to "extend" the path corresponding to semi-legal $(k-1)$-item sequences in $F_{k-1}(b)$ by appending new items or itemsets. For the second type, termed *first-element-augmenting candidates*, we try to "augment" the *first itemset* of $(k-1)$-item sequences in $F_{k-1}(b)$ by adding new items. More formally:

   (a) *Path-Extending Candidates:* For every $(k-1)$-item sequence $t = <t_1\ v>$ in $F_{k-1}(b)$ (where v can be any itemset sequence, possibly empty), if there exists a sequence u in $F_{k-|t_1|}(c)$ such that v is a restricted prefix of u, then the k-item sequence $s = <t_1\ u>$ is added to C_k. For instance, for $k=3$, state a, and transition $a \stackrel{\{1\}}{\rightarrow} b$, itemset sequences $<\{1\}\ \{3,4\}>$ and $<\{1\}\ \{4,5\}>$ are added to C_3. The reason for adding $<\{1\}\ \{3,4\}>$ is that $<\{1\}\ \{3\}>$ is in $F_2(a)$, $<\{3,4\}>$ belongs to $F_2(b)$, and $<\{3\}>$ is a restricted prefix of $<\{3,4\}>$. (Similar justification applies for the candidate $<\{1\}\ \{4,5\}>$.) As an optimization to the above rule, if $w_i \subset t_1$ then we add the sequence $s = <t_1\ u>$ to C_k only if $<(t_1 - \{i\})\ u>$ belongs to $F_{k-1}(b)$ for every item $i \in t_1 - w_i$.

   (b) *First-Element-Augmenting Candidates:* We have two distinct candidate-generation rules for augmenting the first itemset of a sequence in $F_{k-1}(b)$, that give rise to candidates with $|w_i|+1$ or more items in the first itemset. First, if the itemset sequence $<w_i>$ is in $F_{k-1}(b)$ then, for every frequent item i, $<w_i \cup \{i\}>$ is added to C_k. (Note that this first rule is applicable only for $k = |w_i|+1$.) As an example, for $k=2$, state a, and transition $a \stackrel{\{1\}}{\rightarrow} b$, our first rule causes the following itemset sequences to be added to C_2: $<\{1,2\}>$, $<\{1,3\}>$, $<\{1,4\}>$, and $<\{1,5\}>$. Second, for each pair of sequences $<(t_1 \cup \{i\})\ u>$, $<(t_1 \cup \{j\})\ u>$ belonging to $F_{k-1}(b)$ for two distinct items i, j, and such that $w_i \subseteq t_1$, the k-item sequence $<(t_1 \cup \{i,j\})\ u>$ is added to C_k. This is because, as mentioned earlier, for $<(t_1 \cup \{i,j\})\ u>$ to be semi-legal and frequent, each of $<(t_1 \cup \{i\})\ u>$ and $<(t_1 \cup \{j\})\ u>$ must also be semi-legal and frequent. (Note that this second rule applies only for $k > |w_i|+1$.)

Candidate Pruning. The key idea in this step is to generate for every candidate k-item sequence $s \in C_k$, all its subsequences (with fewer than k items) that are semi-legal with respect to some state of $\mathcal{A}_{\mathcal{R}}$. If any of these subsequences is not frequent, then s cannot possibly be frequent and can thus be safely deleted from C_k.

Algorithm FindMaxSubseqItemsets in Figure 9 (similar to FindMaxSubseq presented earlier for item sequences) computes the maximal semi-legal subsequences of a candidate k-item sequence s that contain less than k items. Each of the sequences is semi-legal with respect to a state in *Start* and, when applied over the automaton $\mathcal{A}_{\mathcal{R}}$, results in a state in *End*. In order to compute all the semi-legal subsequences of s, FindMaxSubseqItemsets is invoked with *Start* and *End* both equal to the set of all states in $\mathcal{A}_{\mathcal{R}}$.

Procedure FindMaxSubseqItemsets(*Start*, *End*, s)
begin
1. **for each** state b in automaton $\mathcal{A}_{\mathcal{R}}$ **do**
2.   maxSeq[b] := $\emptyset$
3. **for** $l := |s|$ **down to** 1 **do** {
4.   **for each** state b in automaton $\mathcal{A}_{\mathcal{R}}$ **do** {
5.   tmpSeq[b] = $\emptyset$
6.   **for each** transition $b \xrightarrow{w_i} c$ in $\mathcal{A}_{\mathcal{R}}$ such that $w_i \cap s_l \neq \emptyset$ **do** {
7.   **if** ($c \in End$)
8.   **if** ($w_i \subset s_l$) add s_l and all ($|s_l| - 1$)-subsets of s_l that contain w_i to tmpSeq[b]
9.   **else** add $w_i \cap s_l$ and all ($|w_i \cap s_l| - 1$)-subsets of $w_i \cap s_l$ to tmpSeq[b]
10.   **if** ($w_i \subset s_l$) add to tmpSeq[b] all itemset sequences $< v_l \; t >$, where t is in maxSeq[c] and v_l
11.   is either s_l or a ($|s_l| - 1$)-subset of s_l that contains w_i
12.   **else if** ($w_i = s_l$) add to tmpSeq[b] itemset sequences $< w_i \; t >$, where t is in maxSeq[c]
13.   }
14.   }
15.   **for each** state b in automaton $\mathcal{A}_{\mathcal{R}}$ **do** {
16.   maxSeq[b] := maxSeq[b] $\cup$ tmpSeq[b]
17.   **for each** sequence t in maxSeq[b] **do**
18.   **if** (there exists a sequence $u \neq < s_l \cdots s_{|s|} >$ in maxSeq[b] such that t is a subsequence of u)
19.   delete t from maxSeq[b]
20.   }
21. }
22. **return** $\bigcup_{b \in Start}$ maxSeq[b] - $\{s\}$ (after deleting non-maximal sequences)
end

Figure 9: Algorithm for finding maximal itemset subsequences.

Similar to FindMaxSubseq, algorithm FindMaxSubseqItemsets computes the maximal semi-legal subsequences for increasing suffixes of s in each iteration of the for loop spanning Steps 3–21. The computed semi-legal sequences for a state b are stored in maxSeq[b]. The key differences between the two algorithms arise in Steps 7–12, and can be attributed to the fact that every element of s is an itemset (instead of simply an item). Specifically, for a sequence that contains a single element/itemset v_l and that is semi-legal with respect to state b (due to a first transition of $b \overset{w_i}{\to} c$), it must be the case that either $w_i \subseteq v_l$ or $v_l \subseteq w_i$. The single-element itemset sequences added to tmpSeq[b] in Steps 8–9 correspond to these two cases. In addition, for sequences that are semi-legal with respect to b (due to a first transition $b \overset{w_i}{\to} c$) and that contain two or more elements the first of which is v_l, it must be the case that $w_i \subseteq v_l$. These multi-element sequences are generated in Steps 10–12 of the algorithm. Note that in Steps 7–12, whenever we generate a semi-legal sequence begining with s_l, we also generate semi-legal sequences whose first elements are maximal subsets of s_l. This is because we are interested in maximal subsequences of s that contain *less than* k items. If we do not consider maximal subsets of s_l, then it is possible that maxSeq[b] for a state b may only contain the k-item sequence s.

l	maxSeq[a]	maxSeq[b]
2		$\{< \{5\} >, < \{4,5\} >\}$
1	$\{< \{1\} \ \{4,5\} >, < \{1,2\} \ \{5\} >, < \{1,2\} \ \{4,5\} >\}$	$\{< \{5\} >, < \{4,5\} >\}$

Figure 10: Execution of FINDMAXSUBSEQITEMSETS for $s =< \{1,2\} \ \{4,5\} >$.

State	F_1
b	$< \{3\} >$
b	$< \{4\} >$
b	$< \{5\} >$

State	F_2
a	$< \{1\} \ \{3\} >$
a	$< \{1\} \ \{4\} >$
a	$< \{1\} \ \{5\} >$
b	$< \{3,4\} >$
b	$< \{4,5\} >$

State	F_3
a	$< \{1\} \ \{3,4\} >$
a	$< \{1\} \ \{4,5\} >$
a	$< \{1,2\} \ \{3\} >$
a	$< \{1,2\} \ \{4\} >$
a	$< \{1,2\} \ \{5\} >$

State	F_4
a	$< \{1,2\} \ \{4,5\} >$

Figure 11: Frequent and semi-valid sequences for $\mathcal{R} = \{1\}(\{3,4\} \mid \{5\})$.

Example 5.2: Figure 10 illustrates the maxSeq set for the various states of the automaton from Figure 7 and for decreasing values of l, when FINDMAXSUBSEQITEMSETS is invoked by SPIRIT(L) with $s =< \{1,2\} \ \{4,5\} >$. Consider the final iteration, i.e., $l = 1$. At the start of the iteration, maxSeq[b] contains two sequences $< \{5\} >$ and $< \{4,5\} >$. Since $a \overset{\{1\}}{\rightarrow} b$ and $s_1 = \{1,2\}$, sequences $< \{1\} >$ and $< \{1,2\} >$ are added to maxSeq[a] in Steps 7–8, and sequences $< \{1\} \ \{5\} >$, $< \{1\} \ \{4,5\} >$, $< \{1,2\} \ \{5\} >$ and $< \{1,2\} \ \{4,5\} >$ are added to maxSeq[a] in Steps 10–11. Of these, $< \{1\} \ \{5\} >$ is deleted from maxSeq[a] since it is a subsequence of both $< \{1\} \ \{4,5\} >$ and $< \{1,2\} \ \{5\} >$ (Steps 17–19). Consequently, the final set of maximal semi-legal subsequences returned by FINDMAXSUBSEQITEMSETS is $\{< \{1\} \ \{4,5\} >, < \{1,2\} \ \{5\} >\}$. ∎

Terminating Condition. The set of frequent sequences that are semi-legal with respect to the start state, $F_k(a)$, is empty.

5.1.2 SPIRIT(V) for Itemset Sequences

For a state b in $\mathcal{A}_{\mathcal{R}}$, we define a sequence s to be *semi-valid* with respect to b if there exists a restriction r of $< s_1 \cdots s_{|s|-1} >$ and a state c in $\mathcal{A}_{\mathcal{R}}$ with the following properties: (1) $b \overset{r}{\Rightarrow} c$, (2) there is a transition $c \overset{w_i}{\rightarrow} d$ out of c such that either $w_i \subseteq s_{|s|}$ or $s_{|s|} \subseteq w_i$, and (3) d is an accept state of $\mathcal{A}_{\mathcal{R}}$. Intuitively, this means that the sequence s either defines or can be extended (by adding items to $s_{|s|}$) to define a path from b to an accept state in the constraint automaton $\mathcal{A}_{\mathcal{R}}$. SPIRIT(V) stores in $F_k(b)$ all frequent k-item sequences that are also *semi-valid* with respect to state b of $\mathcal{A}_{\mathcal{R}}$, and in F_k all frequent and semi-valid k-item sequences.

Figure 11 depicts the frequent and semi-valid k-item sequences computed by SPIRIT(V) for the data set and RE constraint in Figure 7. Note that the semi-legal sequence $< \{1,2\} >$ that was generated by SPIRIT(L) is not generated by SPIRIT(V) since it is not semi-valid (the final state b is not an accept state of $\mathcal{A}_{\mathcal{R}}$). A crucial difference between semi-valid and semi-legal sequences is that for a semi-valid sequence s, there may not always exist a restricted prefix that is also semi-valid (unless the last element of s contains two or more items). For example, the sequence $< \{1,2\} \ \{5\} >$ is semi-valid, but its only restricted prefix $< \{1,2\} >$ is not. On the other hand, the sequence $< \{1,2\} \ \{4,5\} >$ whose last element contains two items has a restricted prefix $< \{1,2\} \ \{5\} >$ that is semi-valid. Thus, the candidate generation step of SPIRIT(V), described below, differs slightly from that of SPIRIT(L).

Candidate Generation. Our definition of semi-validity implies that every k-item sequence s that is semi-valid with respect to state b of automaton $\mathcal{A}_{\mathcal{R}}$ (due to a first transition of $b \stackrel{w_i}{\rightarrow} c$) must satisfy the following two properties:

1. If $k \leq |w_i|$, then c is an *accept* state of $\mathcal{A}_{\mathcal{R}}$, and s contains exactly one element/itemset s_1 with $s_1 \subseteq w_i$.

2. If $k > |w_i|$, then $w_i \subseteq s_1$ and the suffix $< s_2 \cdots s_{|s|} >$ is semi-valid with respect to state c. Further, if $w_i \subset s_1$ then for any item i not contained in w_i, $< (s_1 - \{i\}) \ s_2 \cdots s_{|s|} >$ is also semi-valid with respect to b. Finally, if $|s_{|s|}| \geq 2$ then there exists a $(k-1)$-item sequence t that is both a restricted prefix of s and semi-valid with respect to b.

Our candidate-generation rules for SPIRIT(V) are based on these properties. More specifically, for every state b in the automaton $\mathcal{A}_{\mathcal{R}}$ and for every transition $b \stackrel{w_i}{\rightarrow} c$, we generate candidate k-item sequences that are semi-valid with respect to b as follows.

1. $\underline{k \leq |w_i|}$: In this case, if c is an *accept* state, all k-subsets of w_i (that is, subsets containing k items) whose $(k-1)$-subsets are in $F_{k-1}(b)$ are added to C_k. Note that, if any of these subsets prove to be infrequent then the (accepting) transition $b \stackrel{w_i}{\rightarrow} c$ can be safely pruned from the constraint automaton $\mathcal{A}_{\mathcal{R}}$. (A pre-processing optimization similar to that developed for the same case of SPIRIT(L) is also applicable here.)

2. $\underline{k > |w_i|}$: In this case, our candidate-generation process again produces two types of candidate k-item sequences, based on the aforementioned properties of semi-validity.

   (a) *Path-Extending Candidates:* For every sequence u in $F_{k-|w_i|}(c)$ the k-item sequence $s =< w_i \ u >$ is added to C_k. An optimization for candidate sequences $s =< w_i \ u >$ for which the last element of u contains more than one item is to add s to C_k only if there exists a sequence $< w_i \ t >$ in $F_{k-1}(b)$ such that t is a restricted prefix of u.

   (b) *First-Element-Augmenting Candidates:* We have two distinct candidate-generation rules for augmenting the first itemset of a sequence in $F_{k-1}(b)$, that give rise to candidates with $|w_i| + 1$ or more items in the first itemset. First, for every sequence $< w_i \ t >$ in $F_{k-1}(b)$, the sequence $< (w_i \cup \{i\}) \ t >$ is added to C_k for every frequent item i. An optimization for sequences $< w_i \ t >$ for which the last element of t contains more than one item is to add $< (w_i \cup \{i\}) \ t >$ to C_k only if there exists a sequence $< (w_i \cup \{i\}) \ u >$ in $F_{k-1}(b)$ such that u is a restricted prefix of t. Second, for each pair of sequences $< (t_1 \cup \{i\}) \ u >$, $< (t_1 \cup \{j\}) \ u >$ belonging to $F_{k-1}(b)$ for two distinct items i, j, and such that $w_i \subseteq t_1$, the k-item sequence $< (t_1 \cup \{i, j\}) \ u >$ is added to C_k.

Candidate Pruning. For a candidate sequence s in C_k, algorithm FINDMAXSUBSEQITEMSETS can be used to compute the maximal semi-valid subsequences of s (with respect to some state of the automaton $\mathcal{A}_{\mathcal{R}}$) containing fewer than k items. The parameter *Start* is set to be equal to all the states of $\mathcal{A}_{\mathcal{R}}$, while *End* is the set of all *accept* states of $\mathcal{A}_{\mathcal{R}}$. If any of these subsequences is not contained in F, then s is deleted from C_k.

Terminating Condition. For some j, $F_j, \ldots, F_{j+maxLabel-1}$ is empty, where $maxLabel$ is the maximum number of items contained in any transition label of automaton $\mathcal{A}_{\mathcal{R}}$. The reason for this is that if there is a frequent and semi-valid sequence s containing $(j + maxLabel)$ or more items, then $< s_2 \cdots s_{|s|} >$ must also be a semi-valid and frequent sequence containing at least j items, and thus one of $F_j, \ldots, F_{j+maxLabel-1}$ would not be empty.

5.1.3 SPIRIT(R) for Itemset Sequences

During the k^{th} pass, SPIRIT(R) stores in F_k frequent and valid k-item sequences. Figure 7 depicts the frequent and valid k-item sequences computed by SPIRIT(R) for the data set and RE constraint in the figure. We shall refer to sequences that can be derived by concatenating the sequence of transitions from the start state to an accept state as *base sequences.* Sequences that are derived from a base sequence by adding items to itemsets in the base sequence are referred to as *derived sequences.* Thus, every valid sequence is either a base sequence or a derived sequence. For example, in Figure 7, $<\{1\}\ \{5\}>$ is a base sequence and sequences $<\{1,2\}\ \{5\}>$ and $<\{1,2\}\ \{4,5\}>$ are sequences derived from it. Note that it is possible for a valid sequence to be both a base sequence as well as a derived sequence.

The base candidate itemset sequences containing k items can be computed in the same manner as in the simple items case described in Section 4.5. Algorithm GenCandidateItemsets, depicted in Figure 12, computes all candidate base sequences with k items by essentially enumerating paths in the automaton $\mathcal{A}_{\mathcal{R}}$. In the following, we show how the derived sequences can also be computed. The set B_k denotes the frequent *base* k-item sequences, while F_k denotes all the frequent sequences (base and derived) containing k items. The function *numItems*(s) returns the total number of items contained in sequence s.

Procedure GenCandidateItemsets(s, b, B)
begin
for each transition $b \xrightarrow{w_i} c$ in $\mathcal{A}_{\mathcal{R}}$ **do** {
  if ($numItems(<s\ w_i>) = k$ **and** c is an accept state) $C_k = C_k \cup \{<s\ w_i>\}$
  if ($numItems(<s\ w_i>) < k$ **and** (c is not an accept state **or** $<s\ w_i> \in B_{numItems(<s\ w_i>)}$)) {
    if ($c \in B$) {
      let $s = <t\ u>$, where t is the prefix of s for which $a \stackrel{t}{\Rightarrow} c$
      $C_k := C_k \cup \{<t\ u\ w_i\ v>: <t\ v> \in B_{k-numItems(<u\ w_i>)}\}$
    }
    else GenCandidateItemsets($<s\ w_i>, c, B \cup \{b\}$)
  }
}
end

Figure 12: Algorithm for generating candidate base sequences.

Candidate Generation. As mentioned earlier, candidate base k-item sequences can be generated by invoking procedure GenCandidateItemsets with input arguments $s = \epsilon$ (the empty sequence), $b = a$ ($\mathcal{A}_{\mathcal{R}}$'s start state), and $B = \{a\}$. Candidate derived sequences that result due to the addition of items to some base sequence of size less than k can be generated as follows.

1. For each base sequence s in B_{k-1}, for each element w_j of s, and for each frequent item i, a candidate sequence containing k items is derived by replacing w_j in the sequence s with $w_j \cup \{i\}$ (the derived sequence is added to C_k). Thus, in Figure 7, for $k = 3$, since itemset sequence $<\{1\}\ \{5\}>$ belongs to B_2, derived sequences $<\{1,2\}\ \{5\}>$ and $<\{1\}\ \{4,5\}>$ (among others) are added to C_3.

2. For each pair of derived sequences s and t (from the same base sequence) of length $k-1$ in F_{k-1}, if it is the case that for some l, m: $s_l - t_l = \{i\}$ and $t_m - s_m = \{j\}$ for two distinct items i and j, and for the remaining itemsets $p \neq l, m$, $s_p = t_p$, then the sequence $<s_1 \cdots (s_m \cup \{j\}) \cdots s_{|s|}>$ is added to C_k. To see why this works, note that if a sequence is derived from a base sequence by adding

two or more items, then every subsequence that results from deleting one of the added items is a derived sequence of the base sequence. Thus, in Figure 7, for $k = 4$, sequence $< \{1,2\}\ \{4,5\} >$ is added to C_4 since the pair of sequences $< \{1,2\}\ \{5\} >$ and $< \{1\}\ \{4,5\} >$ (both derived from the same base sequence $< \{1\}\ \{5\} >$) belong to F_3.

Candidate Pruning. For each candidate sequence s in C_k, if a valid subsequence of s with fewer than k items is not contained in F, then s is deleted from C_k. The maximal valid subsequences of s can be computed using algorithm FINDMAXSUBSEQITEMSETS (Figure 9) with a slight modification, since we are interested in *valid* (rather than semi-valid) subsequences. Thus, Step 9 should read "**else if** ($w_i = s_l$) add w_i to tmpSeq[b]" (since if $w_i \not\subseteq s_l$, then s_l cannot be part of a valid sequence involving transition w_i). Of course, algorithm FINDMAXSUBSEQITEMSETS must be invoked with *Start* and *End* equal to the start and accept states of A, respectively.

Terminating Condition. For some j, $F_j, \ldots, F_{j+maxPath-1}$ are all empty, where $maxPath$ is the maximum number of items along paths of $\mathcal{A}_{\mathcal{R}}$ that do not contain cycles. To see this, consider any frequent and valid sequence s containing greater than $j + maxPath - 1$ items. Obviously, s contains at least one cycle containing at most $maxPath$ items and, therefore, s must contain at least one frequent and valid subsequence consisting of at least j items. However, no valid sequence with greater than or equal to j items is frequent (since $F_j, \ldots, F_{j+maxPath-1}$ are all empty). Thus, s cannot be a frequent and valid sequence.

Optimizations. If a base sequence s contains a cycle in the automaton $\mathcal{A}_{\mathcal{R}}$, then when generating derived candidate sequences from s that contain $numItems(s)+1$ items by adding an item to an element of s, we can exploit our earlier cycle optimizations to make this step more efficient. For instance, let $s =< t\ u\ v >$ where the subsequence u defines a cycle in $\mathcal{A}_{\mathcal{R}}$. Now, $< t\ v >$ is a base sequence that is frequent with fewer items than s. Thus, when considering the items with which an element of t or v of the new longer base sequence s can be extended by, we only need to consider the items that, when used to extend the corresponding element of t or v in the shorter base sequence $< t\ v >$, resulted in a frequent itemset sequence. This can be used to prune the number of candidates derived from the longer base sequence s by the addition of a single item.

5.2 Handling the Maximum Distance Constraint

In the presence of a distance constraint δ, the problem is to compute valid sequences s such that the fraction of data sequences of which s is a δ-distance subsequence, exceeds the minimum support threshold (see Section 3.1). A key difference, when the distance constraint is specified, is that *every subsequence of a frequent sequence may not necessarily also be frequent* [12]. However, every *contiguous* (i.e., 1-distance) subsequence of a frequent sequence is still guaranteed to be frequent. We consider the impact of maximum distance constraints on the SPIRIT algorithms for both the item and itemset cases.

Item Sequences. The contiguity requirement for frequent subsequences obviously makes the candidate pruning steps of SPIRIT(N) and SPIRIT(L) presented earlier inapplicable. However, except for this, the candidate generation and termination steps remain the same and can be used to mine sequences in the presence of distance constraints.

For SPIRIT(V), too, except for the candidate pruning step, the other steps remain the same. The candidate pruning phase first computes the maximal prefix of the candidate k-sequence s whose length is less than k and that is valid with respect to some state of $\mathcal{A}_{\mathcal{R}}$ (the computation of this in time $O(k * |\mathcal{A}_{\mathcal{R}}|)$ is straightforward). If this maximal prefix is not contained in F, then the candidate is pruned.

Finally, both the candidate generation and the candidate pruning steps of SPIRIT(R) need to be modified to handle the distance constraint. In the candidate generation step, the second optimization to exploit cycles in the automaton $\mathcal{A}_{\mathcal{R}}$ cannot be used since eliminating cycles from a sequence does not result in a contiguous subsequence of the original sequence. Thus, Steps 4-9 that span the body of the **if** condition in Step 3 must simply be replaced with GENCANDIDATES($< s\ w_i >, b, B \cup \{b\}$). In the candidate pruning step, a candidate sequence s is pruned from C_k if some valid contiguous subsequence of s with length less than k is not in F (this can be computed in $O(k^2)$ steps).

Itemset Sequences. As in the case of simple items, the contiguity requirement imposed by the maximum distance constraint mainly affects the candidate pruning phase of the SPIRIT algorithms. More specifically, the new pruning rules can be summarized as follows.

- SPIRIT(L): Consider the longest (contiguous) *prefix* t of a candidate k-item sequence s, such that (a) t contains at most $k-1$ items, and (b) t is *semi-legal* with respect to some state of $\mathcal{A}_{\mathcal{R}}$. If t is not frequent then s can be pruned from C_k.
- SPIRIT(V): Consider the longest (contiguous) *prefix* t of a candidate k-item sequence s, such that (a) t contains at most $k-1$ items, and (b) t is *semi-valid* with respect to some state of $\mathcal{A}_{\mathcal{R}}$. If t is not frequent then s can be pruned from C_k.
- SPIRIT(R): Consider any contiguous subsequence t of a candidate k-item sequence s such that (a) t contains at most $k-1$ items, and (b) t is *valid.* If t is not frequent then s can be pruned from C_k.

6 Experimental Results

In this section, we present an empirical study of the four SPIRIT algorithms with synthetic and real-life data sets. The objective of this study is twofold: (1) to establish the effectiveness of allowing and exploiting RE constraints during sequential pattern mining; and, (2) to quantify the constraint-based vs. support-based pruning tradeoff for the SPIRIT family of algorithms (Section 4.1).

In general, RE constraints whose automata contain fewer transitions per state, fewer cycles, and longer paths tend to be more *selective*, since they impose more stringent restrictions on the ordering of items in the mined patterns. Our expectation is that for RE constraints that are more selective, constraint-based pruning will be very effective and the latter SPIRIT algorithms will perform better. On the other hand, less selective REs increase the importance of good support-based pruning, putting algorithms that use the RE constraint too aggressively (like SPIRIT(R)) at a disadvantage. Our experimental results corroborate our expectations. More specifically, our findings can be summarized as follows.

1. The SPIRIT(V) algorithm emerges as the overall winner, providing consistently good performance over the entire range of RE constraints. For certain REs, SPIRIT(V) is more than an order of magnitude faster than the "naive" SPIRIT(N) scheme.
2. For highly selective RE constraints, SPIRIT(R) outperforms the remaining algorithms. However, as the RE constraint becomes less selective, the number of candidates generated by SPIRIT(R) explodes and the algorithm fails to even complete execution for certain cases (it runs out of virtual memory).
3. The overheads of the candidate generation and pruning phases for the SPIRIT(L) and SPIRIT(V) algorithms are negligible. They typically constitute less than 1% of the total execution time, even for complex REs with automata containing large numbers of transitions, states, and cycles.

Thus, our experimental results validate the thesis of this paper that incorporating RE constraints into the mining process can lead to significant performance benefits. All experiments reported in this section were performed on a Sun Ultra-2/200 workstation with 512 MB of main memory, running Solaris 2.5. The data sets were stored on a local disk.

6.1 Synthetic Data Sets

We used a synthetic data set generator to create a database of sequences containing items. The input parameters to our generator include the number of sequences in the database, the average length of each sequence, the number of distinct items, and a Zipf parameter z that governs the probability of occurrence, $\frac{1}{i^z}/\Sigma_i \frac{1}{i^z}$, of each item i in the database. The length for each sequence is selected from a Poisson distribution with mean equal to the average sequence length. Note that an item can appear multiple times in a single data sequence.

In addition, since we are interested in a sensitivity analysis of our algorithms with respect to the RE constraint $\mathcal{R}$, we used a RE generator to produce constraints with a broad range of selectivities. Each RE constraint output by the generator consists of *blocks* and each block in turn contains *terms* with the following structure. A term T_i is a disjunction of items and has the form $(s_1 \mid s_2 \mid \cdots \mid s_l)$. Each block B_i is simply a concatenation of terms, $T_1 T_2 \cdots T_m$. Finally, the constraint $\mathcal{R}$ is constructed from blocks and has the form $(B_1 \mid B_2 \mid \cdots \mid B_n)^*$ – thus, every sequence that satisfies $\mathcal{R}$ is a concatenation of one or more sequences satisfying the block constraints. The generic structure of the automaton $\mathcal{A}_{\mathcal{R}}$ for $\mathcal{R}$ is shown in Figure 13. RE constraints with different selectivities can be generated by varying the number of items per term, the number of terms per block, and the number of blocks in $\mathcal{R}$. Note that, in terms of the automaton $\mathcal{A}_{\mathcal{R}}$, these parameters correspond to the number of transitions between a pair of states in $\mathcal{A}_{\mathcal{R}}$, the length of each cycle, and the number of cycles contained in $\mathcal{A}_{\mathcal{R}}$, respectively.

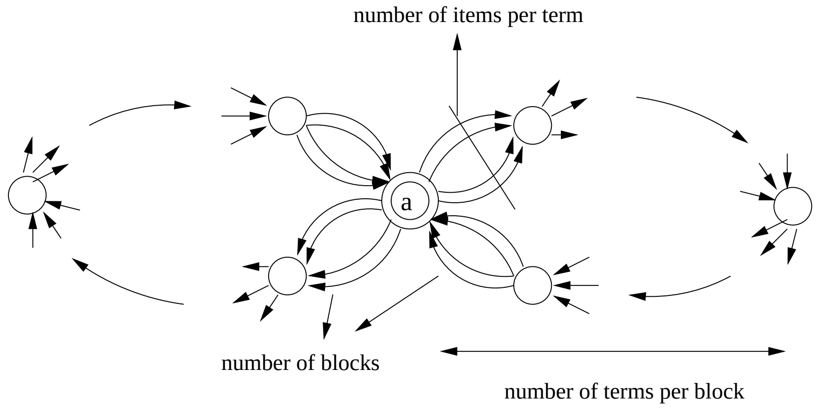
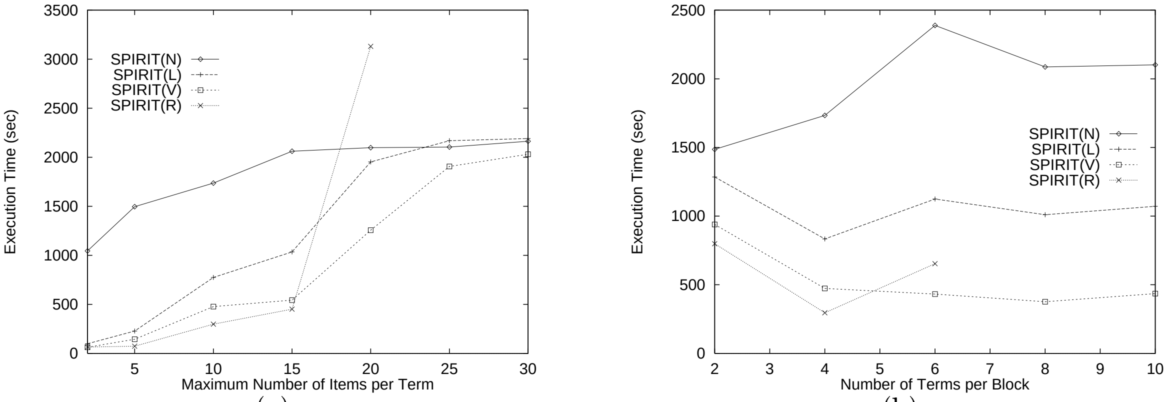


Figure 13: Structure of automaton for the RE constraint generator.

The RE generator accepts the maximum number of items per term, the number of terms per block, and the number of blocks as input parameters. In the RE constraint that it outputs, the number of items per term is uniformly distributed between 1 and the maximum specified value. The items in each term of $\mathcal{R}$ are chosen using the same Zipfian distribution that was used to generate the data set. The RE generator thus enables us to carry out an extensive study of the sensitivity of our algorithms to a wide range of RE constraints with different selectivities.

Table 3 shows the parameters for the data set and the RE constraint, along with their default values and the range of values for which experiments were conducted. The default value of $z = 1.0$ was chosen to model an (approximate) 70-30 rule and to ensure that the item skew was sufficient for some



(a) (b)

Figure 14: Performance results for (a) number of items and (b) number of terms.

interesting patterns to appear in the data sequences. In each experiment, one parameter was varied with all other parameters fixed at their default values.

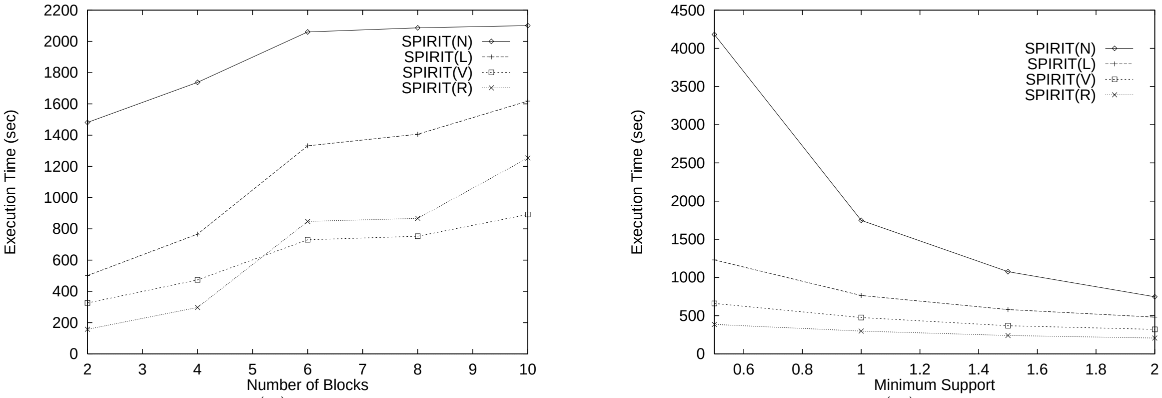
Parameter	**Default Value**	**Range of Values**
Number of Sequences	100000	50000 to 250000
Average Length of Sequence	10	
Number of Items	1000	
Zipf Value	1.0	
Maximum Number of Items Per Term	10	2 to 30
Number of Terms Per Block	4	2 to 10
Number of Blocks	4	2 to 10
Minimum Support	1.0	0.5 to 2.0
Maximum Distance	2	0 to 15

Table 3: Synthetic data and RE constraint parameters.

6.2 Performance Results with Synthetic Data Sets

Maximum Number of Items Per Term. Figure 14(a) illustrates the execution times of the SPIRIT algorithms as the maximum number of items per term in $\mathcal{R}$ is increased. As expected, as the number of items is increased, the number of transitions per state in $\mathcal{A}_{\mathcal{R}}$ also increases and so do the numbers of legal and valid sequences. Thus, constraint-based pruning becomes less effective and the performance of all SPIRIT algorithms deteriorates as more items are added to each term. As long as the number of items per term does not exceed 15, $\mathcal{R}$ is fairly selective; consequently, constraint-based pruning works well and the SPIRIT algorithms that use $\mathcal{R}$ to prune more candidates perform better. For instance, when the maximum number of items per term is 10, the SPIRIT(N), SPIRIT(L), SPIRIT(V), and SPIRIT(R) algorithms count support for 7105, 1418, 974, and 3822 candidate sequences, respectively. SPIRIT(R) makes only two passes over the data for valid candidate sequences of lengths 4 and 8. The remaining algorithms make 8 passes to count supports for candidates with lengths up to 8, a majority of which have lengths 4 and 5.

However, beyond 15 items per term, the performance of the algorithms that rely more heavily on constraint $\mathcal{R}$ for pruning candidates degenerates rapidly. SPIRIT(R) sustains the hardest hit since



(a) (b)

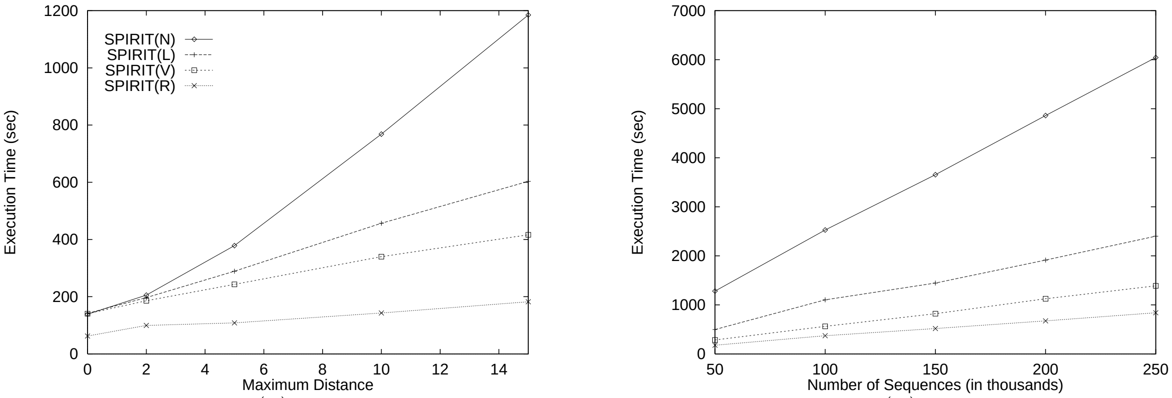
Figure 15: Performance results for (a) number of blocks and (b) minimum support.

it performs very little support-based pruning and its exhaustive enumeration approach for candidate generation results in an enormous number of candidates of length 4. In contrast, since SPIRIT(N) only uses $\mathcal{R}$ to prune sequences not involving items in $\mathcal{R}$, and few new items are added to terms in $\mathcal{R}$ once the number of items per term reaches 15, the execution times for the SPIRIT(N) algorithm hold steady. Beyond 25 items per term, the running times of SPIRIT(L) and SPIRIT(V) also stabilize, since decreases in the amount of constraint-based pruning as $\mathcal{R}$ becomes less selective are counterbalanced by increases in support-based pruning. At 30 items per term, SPIRIT(V) continues to provide a good balance of constraint-based and support-based pruning and, thus, performs the best.

Number of Terms Per Block. The graph in Figure 14(b) plots the running times for the SPIRIT algorithms as the number of terms per block is varied from 2 to 10. Increasing the number of terms per block actually causes each cycle (involving the start state a) to become longer. The initial dip in execution times for SPIRIT(L), SPIRIT(V), and SPIRIT(R) when the number of terms is increased from 2 to 4 is due to the reduction in the number of candidate sequences of lengths 4 and 5. This happens because with short cycles of length 2 in $\mathcal{A}_{\mathcal{R}}$, sequences of length 4 and 5 visit the start state multiple times and the start state has a large number of outgoing transitions. But when $\mathcal{A}_{\mathcal{R}}$ contains cycles of length 4 or more, the start state is visited at most once, thus causing the number of candidate sequences of lengths 4 and 5 to decrease. As cycle lengths grow beyond 4, the number of legal sequences (with respect to a state in $\mathcal{A}_{\mathcal{R}}$) starts to increase due to the increase in the number of states in each cycle. However, the number of valid sequences (with respect to a state in $\mathcal{A}_{\mathcal{R}}$) does not vary much since each of them is still required to terminate at the start state a.

Note that when the number of terms exceeds 6, the number of candidates generated by SPIRIT(R) simply explodes due to the longer cycles. On the other hand, SPIRIT(V) provides consistently good performance throughout the entire range of block sizes.

Number of Blocks. Figure 15(a) depicts the performance of the four algorithms as the number of blocks in $\mathcal{R}$ is increased from 2 to 10. The behavior of the four algorithms has similarities to the "number of items per term" case (Figure 14(a)). The only difference is that, as the number of blocks is increased, the decrease in $\mathcal{R}$'s selectivity and the increase in the number of legal and valid sequences in $\mathcal{A}_{\mathcal{R}}$ are not as dramatic. This is because the number of blocks only affects the number of transitions associated with the start state – the number of transitions for other states in $\mathcal{A}_{\mathcal{R}}$ stays the same. Once again, SPIRIT(V) performs well consistently, for the entire range of numbers of blocks. An interesting case is that of SPIRIT(R) whose execution time does degrade beyond SPIRIT(V)'s, as the number of blocks is increased, but it still manages to do better than SPIRIT(L), even when $\mathcal{R}$ contains 10 blocks.



(a) (b)

Figure 16: Performance results for (a) maximum distance and (b) data set size.

This can be attributed predominantly to the effectiveness of the optimization for cycles in $\mathcal{A}_{\mathcal{R}}$ that is applied during SPIRIT(R)'s candidate generation phase. In general, due to our cycle optimization, one can expect the SPIRIT(R) algorithm to perform reasonably well, even when $\mathcal{A}_{\mathcal{R}}$ contains a large number of cycles of moderate length.

Minimum Support. The execution times for the SPIRIT algorithms as the minimum support threshold is increased from 0.5 to 2.0 are depicted in Figure 15(b). As expected, the performance of all algorithms improves as the minimum support threshold is increased. This is because fewer candidates have the potential to be frequent for higher values of minimum support. Furthermore, note that the running times of algorithms that rely more heavily on support-based pruning improve much more rapidly.

Maximum Distance. Figure 16(a) illustrates the effect on execution times of varying the maximum distance constraint δ from 0 to 15. The performance of all the algorithms degrades as δ is increased, since more subsequences contained within each data sequence satisfy the RE constraint $\mathcal{R}$ and the minimum support constraint. As a consequence, the number of candidates generated by every algorithm increases. Also, as discussed in Section 5.2, in the presence of the distance constraint, very little support-based pruning is possible. Thus, for algorithms that rely more on constraint-based pruning, the number of candidates grows slowly, and their performance deteriorates more gradually as δ is increased. For small values of δ (e.g., 0, 2), since the number of candidate sequences generated by the algorithms is small (SPIRIT(N) and SPIRIT(V) generate 2249 and 293 candidates, respectively), the running times for all the algorithms are similar and do not depend much on the number of candidates.

Data Set Size. The graph in Figure 16(b) plots the execution times of the algorithms as the number of sequences in the data set is increased. As shown in the figure, the times for all the SPIRIT algorithms scale linearly with data set size. This is because the number of candidates generated by each algorithm is independent of the data set size. Furthermore, the overall computation time is dominated by the cost of determining which candidate sequences are contained within each data sequence (during the support counting step), which is proportional to the size of the data set.

Size	Frequent and Valid Sequences
2	< /main.html /academics/ms-program.html >
3	< /main.html /general/contacts.html /academics/ms-program.html >
	< /main.html /general/nav.html /academics/ms-program.html >
	< /main.html /academics/academics.html /academics/ms-program.html >
	< /main.html /academics/nav.html /academics/ms-program.html >
	< /main.html /admissions/nav.html /academics/ms-program.html >
	< /main.html /admissions/admissions.html /academics/ms-program.html >
4	</main.html /general/nav.html /general/contacts.html /academics/ms-program.html >
	</main.html /academics/nav.html /academics/academics.html /academics/ms-program.html >
	</main.html /admissions/nav.html /admissions/admissions.html /academics/ms-program.html >

Table 4: Frequent and valid patterns discovered in the WWW logs.

6.3 Real-life Data Set

For our real-life data experiments, we used the WWW server access logs from the web site of an academic CS department[7]. The logs contain the sequences of web pages accessed by each user[8] starting from the department's web site, for the duration of a week. The department's home page contains links to a number of topics, including `Academics`, `Admissions`, `Events`, `General information`, `Research`, `People`, and `Resources`. There are additional links to the university and college home pages to which the CS department belongs, but we chose not to use these links in our RE constraint. Users navigate through the web pages by clicking on links in each page, and the sequences of pages accessed by a user are captured in the server logs.

We used a RE constraint to focus on user access patterns that start with the department's home page (located at `/main.html`) and end at the web page containing information on the M.S. degree program (located at `/academics/ms-program.html`). In addition, we restricted ourselves to patterns for which the intermediate pages belong to one of the aforementioned 7 topics (e.g., `Academics`). Thus, the automaton $\mathcal{A}_{\mathcal{R}}$ contains three states. There is a transition from the first (start) state to the second on `/main.html` and a transition from the second state to the third (accept) state on `/academics/ms-program.html`. The second state has 15 transitions to itself, each labeled with the location of a web page belonging to one of the above 7 topics. We used a minimum support threshold of 0.3%. The number of access sequences logged in the one week data set was 12868.

The mined frequent and valid access patterns are listed in increasing order of size in Table 4. Note that there is a number of distinct ways to access the M.S. degree program web page by following different sequences of links (e.g., via admissions, academics). The execution times and the numbers of candidates generated by the four SPIRIT algorithms are presented in Table 5. As expected, since the RE constraint is fairly selective, both SPIRIT(V) and SPIRIT(R) have the smallest running times. SPIRIT(L) is about twice as slow compared to SPIRIT(V) and SPIRIT(R). The execution time for SPIRIT(N) is almost two orders of magnitude worse than SPIRIT(V) and SPIRIT(R), since it generates a significantly larger number of candidate sequences with lengths between 5 and 9 (almost 4000). We believe that our results clearly demonstrate the significant performance gains that can be attained by pushing RE constraints inside a *real-life* pattern mining task.

[7]At the department's request, we do not disclose its identity.

[8]We use IP addresses to distinguish between users.

Algorithm	Execution Time (secs)	Number of Candidates	Number of Passes
SPIRIT(N)	1562.8	5896	13
SPIRIT(L)	32.77	1393	10
SPIRIT(V)	16.0.	59	5
SPIRIT(R)	17.67	52	7

Table 5: Execution times and candidate numbers for the SPIRIT algorithms.

7 Conclusions and Future Work

In this paper, we have proposed the use of Regular Expressions (REs) as a flexible constraint specification tool that enables user-controlled focus to be incorporated into the pattern mining process. We have developed a family of novel algorithms (termed SPIRIT – Sequential Pattern mIning with Regular expressIon consTraints) for mining frequent sequential patterns that also satisfy user-specified RE constraints. The main distinguishing factor among the proposed schemes is the degree to which the RE constraints are enforced to prune the search space of patterns during computation. The SPIRIT algorithms are illustrative of the tradeoffs that arise when constraints that do not subscribe to nice properties (like anti-monotonicity) are integrated into the mining process. To explore these tradeoffs, we have conducted an extensive experimental study on synthetic and real-life data sets. The experimental results clearly validate the effectiveness of our approach, showing that speedups of more than an order of magnitude are possible when RE constraints are pushed deep inside the mining process. Our experimentation with real-life data also illustrates the versatility of REs as a user-level tool for focusing on interesting patterns.

We believe that integrating user-specified constraints into mining algorithms is an important and fertile area of research that has received relatively little attention from the data mining community. We are actively pursuing several open problems in this broad area, including (a) a detailed characterization of the possible performance tradeoffs for processing non anti-monotone constraints, and (b) developing a taxonomy of non anti-monotone constraints based on key properties that impact the candidate generation and pruning phases.

Acknowledgments: The work of Kyuseok Shim was partially supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center (AITrc). Without the support of Yesook Shim, it would have been impossible to complete this work.

References

[1] Rakesh Agrawal, Giuseppe Psaila, Edward L. Wimmers, and Mohamed Zaït. "Querying Shapes of Histories". In *Proceedings of the 21st International Conference on Very Large Data Bases*, Zurich, Switzerland, September 1995.

[2] Rakesh Agrawal and Ramakrishnan Srikant. "Fast Algorithms for Mining Association Rules". In *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, September 1994.

[3] Rakesh Agrawal and Ramakrishnan Srikant. "Mining Sequential Patterns". In *Proceedings of the Eleventh International Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[4] Soumen Chakrabarti, Byron Dom, Rakesh Agrawal, and Prabhakar Raghavan. "Using taxonomy, discriminants, and signatures for navigating in text databases". In *Proceedings of the 23rd International Conference on Very Large Data Bases*, Athens, Greece, August 1997.

[5] Ming-Syan Chen, Jong Soo Park, and Philip S. Yu. "Efficient Data Mining for Path Traversal Patterns". *IEEE Transactions on Knowledge and Data Engineering*, 10(2):209–221, March 1998.

[6] K. Hätönen, M. Klemettinen, P. Mannila, H. Ronkainen, and H. Toivonen. "Knowledge Discovery from Telecommunication Network Alarm Databases". In *Proceedings of the Twelfth International Conference on Data Engineering*, pages 115–122, New Orleans, Louisiana, February 1996.

[7] Harry R. Lewis and Christos Papadimitriou. *"Elements of the Theory of Computation"*. Prentice Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

[8] Heikki Mannila and Hannu Toivonen. "Discovering Generalized Episodes Using Minimal Occurrences". In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, Portland, Oregon, August 1996.

[9] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. "Discovering Frequent Episodes in Sequences". In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, pages 210–215, Montreal, Canada, August 1995.

[10] Raymond T. Ng, Laks V.S. Lakshmanan, Jiawei Han, and Alex Pang. "Exploratory Mining and Pruning Optimizations of Constrained Association Rules". In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, Seattle, Washington, June 1998.

[11] Ramakrishnan Srikant, Quoc Vu, and Rakesh Agrawal. "Mining Association Rules with Item Constraints". In *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, Newport Beach, California, August 1997.

[12] Ramakrishnan Srikant and Rakesh Agrawal. "Mining Sequential Patterns: Generalizations and Performance Improvements". In *Proceedings of the Fifth International Conference on Extending Database Technology (EDBT'96)*, Avignon, France, March 1996.

[13] Dick Tsur, Jeffrey R. Ullman, Serge Abiteboul, Chris Clifton, Rajeev Motwani, Svetlozar Nestorov and Arnon Rosenthal. "Query Flocks: A Generalization of Association-rule Mining". In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 1–12, Seattle, Washington, May 1998.

[14] Jason Tsong-Li Wang, Gung-Wei Chirn, Thomas G. Marr, Bruce Shapiro, Dennis Shasha, and Kaizhong Zhang. "Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results". In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 115–125, Minneapolis, Minnesota, May 1994.