Mining Sequential Patterns: Generalizations and Performance Improvements

Ramakrishnan Srikant[*] and Rakesh Agrawal

{srikant, ragrawal}@almaden.ibm.com
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract. The problem of mining sequential patterns was recently introduced in [3]. We are given a database of sequences, where each sequence is a list of transactions ordered by transaction-time, and each transaction is a set of items. The problem is to discover all sequential patterns with a user-specified minimum support, where the support of a pattern is the number of data-sequences that contain the pattern. An example of a sequential pattern is "5% of customers bought 'Foundation' and 'Ringworld' in one transaction, followed by 'Second Foundation' in a later transaction". We generalize the problem as follows. First, we add time constraints that specify a minimum and/or maximum time period between adjacent elements in a pattern. Second, we relax the restriction that the items in an element of a sequential pattern must come from the same transaction, instead allowing the items to be present in a set of transactions whose transaction-times are within a user-specified time window. Third, given a user-defined taxonomy (*is-a* hierarchy) on items, we allow sequential patterns to include items across all levels of the taxonomy.
We present GSP, a new algorithm that discovers these generalized sequential patterns. Empirical evaluation using synthetic and real-life data indicates that GSP is much faster than the AprioriAll algorithm presented in [3]. GSP scales linearly with the number of data-sequences, and has very good scale-up properties with respect to the average data-sequence size.

1 Introduction

Data mining, also known as knowledge discovery in databases, has been recognized as a promising new area for database research. This area can be defined as efficiently discovering interesting rules from large databases.

A new data mining problem, *discovering sequential patterns*, was introduced in [3]. The input data is a set of sequences, called *data-sequences*. Each data-sequence is a list of *transactions*, where each transaction is a sets of literals, called *items*. Typically there is a transaction-time associated with each transaction. A *sequential pattern* also consists of a list of sets of items. The problem is to find all

[*] Also, Department of Computer Science, University of Wisconsin, Madison.

sequential patterns with a user-specified minimum *support*, where the support of a sequential pattern is the percentage of data-sequences that contain the pattern.

For example, in the database of a book-club, each data-sequence may correspond to all book selections of a customer, and each transaction to the books selected by the customer in one order. A sequential pattern might be "5% of customers bought 'Foundation', then 'Foundation and Empire', and then 'Second Foundation'". The data-sequence corresponding to a customer who bought some other books in between these books still contains this sequential pattern; the data-sequence may also have other books in the same transaction as one of the books in the pattern. Elements of a sequential pattern can be sets of items, for example, "'Foundation' and 'Ringworld', followed by 'Foundation and Empire' and 'Ringworld Engineers', followed by 'Second Foundation'". However, all the items in an element of a sequential pattern must be present in a single transaction for the data-sequence to support the pattern.

This problem was motivated by applications in the retailing industry, including attached mailing, add-on sales, and customer satisfaction. But the results apply to many scientific and business domains. For instance, in the medical domain, a data-sequence may correspond to the symptoms or diseases of a patient, with a transaction corresponding to the symptoms exhibited or diseases diagnosed during a visit to the doctor. The patterns discovered using this data could be used in disease research to help identify symptoms/diseases that precede certain diseases.

However, the above problem definition as introduced in [3] has the following limitations:

1. **Absence of time constraints.** Users often want to specify maximum and/or minimum time gaps between adjacent elements of the sequential pattern. For example, a book club probably does not care if someone bought "Foundation", followed by "Foundation and Empire" three years later; they may want to specify that a customer should support a sequential pattern only if adjacent elements occur within a specified time interval, say three months. (So for a customer to support this pattern, the customer should have bought "Foundation and Empire" within three months of buying "Foundation".)

2. **Rigid definition of a transaction.** For many applications, it does not matter if items in an element of a sequential pattern were present in two different transactions, as long as the transaction-times of those transactions are within some small time window. That is, each element of the pattern can be contained in the union of the items bought in a set of transactions, as long as the difference between the maximum and minimum transaction-times is less than the size of a *sliding time window*. For example, if the book-club specifies a time window of a week, a customer who ordered the "Foundation" on Monday, "Ringworld" on Saturday, and then "Foundation and Empire" and "Ringworld Engineers" in a single order a few weeks later would still support the pattern "'Foundation' and 'Ringworld', followed by 'Foundation and Empire' and 'Ringworld Engineers'".

3. **Absence of taxonomies.** Many datasets have a user-defined taxonomy

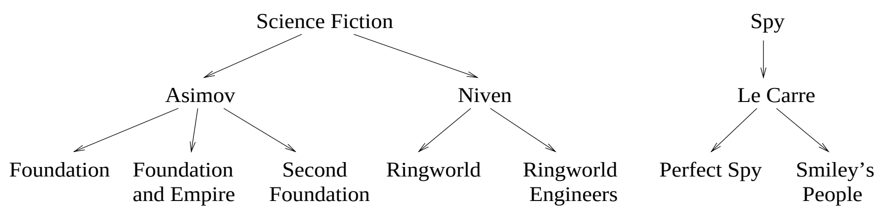

Fig. 1. Example of a Taxonomy

(*is-a* hierarchy) over the items in the data, and users want to find patterns that include items across different levels of the taxonomy. An example of a taxonomy is given in Figure 1. With this taxonomy, a customer who bought "Foundation" followed by "Perfect Spy" would support the patterns "'Foundation' followed by 'Perfect Spy'", "'Asimov' followed by 'Perfect Spy'", "'Science Fiction' followed by 'Le Carre'", etc.

In this paper, we generalize the problem definition given in [3] to incorporate time constraints, sliding time windows, and taxonomies in sequential patterns. We present GSP (Generalized Sequential Patterns), a new algorithm that discovers all such sequential patterns. Empirical evaluation shows that GSP scales linearly with the number of data-sequences, and has very good scale-up properties with respect to the number of transactions per data-sequence and number of items per transaction.

1.1 Related Work

In addition to introducing the problem of sequential patterns, [3] presented three algorithms for solving this problem, but these algorithms do not handle time constraints, sliding windows, or taxonomies. Two of these algorithms were designed to find only maximal sequential patterns; however, many applications require all patterns and their supports. The third algorithm, AprioriAll, finds all patterns; its performance was better than or comparable to the other two algorithms. Briefly, AprioriAll is a three-phase algorithm. It first finds all itemsets with minimum support (frequent itemsets), transforms the database so that each transaction is replaced by the set of all frequent itemsets contained in the transaction, and then finds sequential patterns. There are two problems with this approach. First, it is computationally expensive to do the data transformation on-the-fly during each pass while finding sequential patterns. The alternative, to transform the database once and store the transformed database, will be infeasible or unrealistic for many applications since it nearly doubles the disk space requirement which could be prohibitive for large databases. Second, while it is possible to extend this algorithm to handle time constraints and taxonomies, it does not appear feasible to incorporate sliding windows. For the cases that the extended AprioriAll can handle, our empirical evaluation shows that GSP is upto 20 times faster.

Somewhat related to our work is the problem of mining association rules [1]. Association rules are rules about what items are bought together within a transaction, and are thus intra-transaction patterns, unlike inter-transaction sequential patterns. The problem of finding association rules when there is a user-defined taxonomy on items has been addressed in [6] [4].

The problem of discovering similarities in a database of genetic sequences, presented in [8], is relevant. However, the patterns they wish to discover are subsequences made up of consecutive characters separated by a variable number of noise characters. A sequence in our problem consists of list of sets of characters (items), rather than being simply a list of characters. In addition, we are interested in finding *all* sequences with minimum support rather than some frequent patterns.

A problem of discovering frequent episodes in a sequence of events was presented in [5]. Their patterns are arbitrary DAG (directed acyclic graphs), where each vertex corresponds to a single event (or item) and an edge from event A to event B denotes that A occurred before B. They move a time window across the input sequence, and find all patterns that occur in some user-specified percentage of windows. Their algorithm is designed for counting the number of occurrences of a pattern when moving a window across a single sequence, while we are interested in finding patterns that occur in many different data-sequences.

1.2 Organization of the Paper

We give a formal description of the problem of mining generalized sequential patterns in Section 2. In Section 3, we describe GSP, an algorithm for finding such patterns. We empirically compared the performance of GSP with the AprioriAll algorithm [3], studied the scale-up properties of GSP, and examined the performance impact of time constraints and sliding windows. Due to space limitations, we could not include the details of these experiments which are reported in [7]. However, we include the gist of the main results in Section 4. We conclude with a summary in Section 5.

2 Problem Statement

Definitions Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called *items*. Let $\mathcal{T}$ be a directed acyclic graph on the literals. An edge in $\mathcal{T}$ represents an *is-a* relationship, and $\mathcal{T}$ represents a set of taxonomies. If there is an edge in $\mathcal{T}$ from p to c, we call p a *parent* of c and c a *child* of p. (p represents a generalization of c.) We model the taxonomy as a DAG rather than a tree to allow for multiple taxonomies. We call $\widehat{x}$ an *ancestor* of x (and x a *descendant* of $\widehat{x}$) if there is an edge from $\widehat{x}$ to x in transitive-closure($\mathcal{T}$).

An *itemset* is a non-empty set of items. A *sequence* is an ordered list of itemsets. We denote a sequence s by $\langle s_1 s_2 ... s_n \rangle$, where s_j is an itemset. We also call s_j an *element* of the sequence. We denote an element of a sequence by $(x_1, x_2, ..., x_m)$, where x_j is an item. An item can occur only once in an element of a sequence, but can occur multiple times in different elements. An itemset is considered to be a sequence with a single element. We assume without loss of generality that items in an element of a sequence are in lexicographic order.

A sequence $\langle a_1 a_2 ... a_n \rangle$ is a *subsequence* of another sequence $\langle b_1 b_2 ... b_m \rangle$ if there exist integers $i_1 < i_2 < ... < i_n$ such that $a_1 \subseteq b_{i_1}$, $a_2 \subseteq b_{i_2}$, ..., $a_n \subseteq b_{i_n}$. For example, the sequence $\langle$ (3) (4 5) (8) $\rangle$ is a subsequence of $\langle$ (7) (3, 8) (9) (4, 5, 6) (8) $\rangle$, since (3) $\subseteq$ (3, 8), (4, 5) $\subseteq$ (4, 5, 6) and (8) $\subseteq$ (8). However, the sequence $\langle$ (3) (5) $\rangle$ is not a subsequence of $\langle$ (3, 5) $\rangle$ (and vice versa).

Input We are given a database $\mathcal{D}$ of sequences called *data-sequences*. Each data-sequence is a list of transactions, ordered by increasing transaction-time. A transaction has the following fields: sequence-id, transaction-id, transaction-time, and the items present in the transaction. While we expect the items in a transaction to be leaves in $\mathcal{T}$, we do not require this.

For simplicity, we assume that no data-sequence has more than one transaction with the same transaction-time, and use the transaction-time as the transaction-identifier. We do not consider quantities of items in a transaction.

Support The *support count* (or simply *support*) for a sequence is defined as the fraction of total data-sequences that "contain" this sequence. (Although the word "contains" is not strictly accurate once we incorporate taxonomies, it captures the spirt of when a data-sequence contributes to the support of a sequential pattern.) We now define when a data-sequence *contains* a sequence, starting with the definition as in [3], and then adding taxonomies, sliding windows, and time constraints :

- **as in [3]:** In the absence of taxonomies, sliding windows and time constraints, a data-sequence contains a sequence s if s is a subsequence of the data-sequence.
- **plus taxonomies:** We say that a transaction T *contains* an item $x \in \mathcal{I}$ if x is in T or x is an ancestor of some item in T. We say that a transaction T *contains* an itemset $y \subseteq \mathcal{I}$ if T contains every item in y. A data-sequence $d = \langle d_1 ... d_m \rangle$ contains a sequence $s = \langle s_1 ... s_n \rangle$ if there exist integers $i_1 < i_2 < ... < i_n$ such that s_1 is contained in d_{i_1}, s_2 is contained in d_{i_2}, ..., s_n is contained in d_{i_n}. If there is no taxonomy, this degenerates into a simple subsequence test.
- **plus sliding windows:** The sliding window generalization relaxes the definition of when a data-sequence contributes to the support of a sequence by allowing a set of transactions to contain an element of a sequence, as long as the difference in transaction-times between the transactions in the set is less than the user-specified window-size. Formally, a data-sequence $d = \langle d_1 ... d_m \rangle$ contains a sequence $s = \langle s_1 ... s_n \rangle$ if there exist integers $l_1 \leq u_1 < l_2 \leq u_2 < ... < l_n \leq u_n$ such that
  1. s_i is contained in $\cup_{k=l_i}^{u_i} d_k$, $1 \leq i \leq n$, and
  2. transaction-time(d_{u_i}) $-$ transaction-time$(d_{l_i}) \leq$ window-size, $1 \leq i \leq n$.

- **plus time constraints:** Time constraints restrict the time gap between sets of transactions that contain consecutive elements of the sequence. Given user-specified window-size, max-gap and min-gap, a data-sequence $d = \langle d_1 ... d_m \rangle$ contains a sequence $s = \langle s_1 ... s_n \rangle$ if there exist integers $l_1 \leq u_1 < l_2 \leq u_2 < ... < l_n \leq u_n$ such that

  1. s_i is contained in $\cup_{k=l_i}^{u_i} d_k$, $1 \leq i \leq n$,
  2. transaction-time(d_{u_i}) − transaction-time(d_{l_i}) $\leq$ window-size, $1 \leq i \leq n$,
  3. transaction-time(d_{l_i}) − transaction-time($d_{u_{i-1}}$) $>$ min-gap, $2 \leq i \leq n$, and
  4. transaction-time(d_{u_i}) − transaction-time($d_{l_{i-1}}$) $\leq$ max-gap, $2 \leq i \leq n$.

  The first two conditions are the same as in the earlier definition of when a data-sequence contains a pattern. The third condition specifies the minimum time-gap constraint, and the last the maximum time-gap constraint.
  We will refer to transaction-time(d_{l_i}) as *start-time*(s_i), and transaction-time(d_{u_i}) as *end-time*(s_i). In other-words, start-time(s_i) and end-time(s_i) correspond to the first and last transaction-times of the set of transactions that contain s_i.

Note that if there is no taxonomy, min-gap = 0, max-gap = ∞ and window-size = 0 we get the notion of sequential patterns as introduced in [3], where there are no time constraints and items in an element come from a single transaction.

2.1 Problem Definition

Given a database $\mathcal{D}$ of data-sequences, a taxonomy $\mathcal{T}$, user-specified min-gap and max-gap time constraints, and a user-specified sliding-window size, the problem of mining sequential patterns is to find all sequences whose support is greater than the user-specified minimum support. Each such sequence represents a *sequential pattern*, also called a *frequent* sequence.

Given a frequent sequence $s = \langle s_1 ... s_n \rangle$, it is often useful to know the "support relationship" between the elements of the sequence. That is, what fraction of the data-sequences that support $\langle s_1 ... s_i \rangle$ support the entire sequence s. Since $\langle s_1 ... s_i \rangle$ must also be a frequent sequence, this relationship can easily be computed.

2.2 Example

Consider the data-sequences shown in Figure 2. For simplicity, we have assumed that the transaction-times are integers; they could represent, for instance, the number of days after January 1, 1995. We have used an abbreviated version of the taxonomy given in Figure 1. Assume that the minimum support has been set to 2 data-sequences.

With the [3] problem definition, the only 2-element sequential patterns is:

⟨ (Ringworld) (Ringworld Engineers) ⟩

Database $\mathcal{D}$

Sequence-Id	Transaction Time	Items
C1	1	Ringworld
C1	2	Foundation
C1	15	Ringworld Engineers, Second Foundation
C2	1	Foundation, Ringworld
C2	20	Foundation and Empire
C2	50	Ringworld Engineers

Taxonomy $\mathcal{T}$

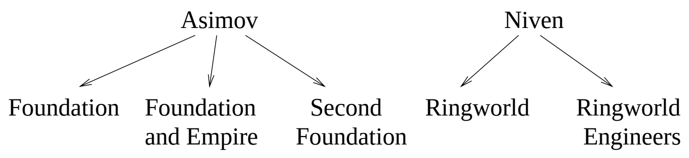

Fig. 2. Example

Setting a sliding-window of 7 days adds the pattern

⟨(Foundation, Ringworld) (Ringworld Engineers)⟩

since C1 now supports this pattern. ("Foundation" and "Ringworld" are present within a period of 7 days in data-sequence C1.)

Further setting a max-gap of 30 days results in both the patterns being dropped, since they are no longer supported by customer C2.

If we only add the taxonomy, but no sliding-window or time constraints, one of the patterns added is:

⟨(Foundation) (Asimov)⟩

Observe that this pattern is not simply a replacement of an item with its ancestor in an existing pattern.

3 Algorithm "GSP"

The basic structure of the GSP algorithm for finding sequential patterns is as follows. The algorithm makes multiple passes over the data. The first pass determines the support of each item, that is, the number of data-sequences that include the item. At the end of the first pass, the algorithm knows which items are frequent, that is, have minimum support. Each such item yields a 1-element frequent sequence consisting of that item. Each subsequent pass starts with a seed set: the frequent sequences found in the previous pass. The seed set is used to generate new potentially frequent sequences, called *candidate* sequences. Each candidate sequence has one more item than a seed sequence; so all the candidate sequences in a pass will have the same number of items. The support for these candidate sequences is found during the pass over the data. At the end of the pass, the algorithm determines which of the candidate sequences are actually frequent. These frequent candidates become the seed for the next pass. The algorithm terminates when there are no frequent sequences at the end of a pass, or when there are no candidate sequences generated.

We need to specify two key details:

1. *Candidate generation:* how candidates sequences are generated before the pass begins. We want to generate as few candidates as possible while maintaining completeness.
2. *Counting candidates:* how the support count for the candidate sequences is determined.

Candidate generation is discussed in Section 3.1, and candidate counting in Section 3.2. We incorporate time constraints and sliding windows in this discussion, but do not consider taxonomies. Extensions required to handle taxonomies are described in Section 3.3.

Our algorithm is not a main-memory algorithm. If the candidates do not fit in memory, the algorithm generates only as many candidates as will fit in memory and the data is scanned to count the support of these candidates. Frequent sequences resulting from these candidates are written to disk, while those candidates without minimum support are deleted. This procedure is repeated until all the candidates have been counted. Further details about memory management can be found in [7].

3.1 Candidate Generation

We refer to a sequence with k items as a k-sequence. (If an item occurs multiple times in different elements of a sequence, each occurrence contributes to the value of k.) Let L_k denote the set of all frequent k-sequences, and C_k the set of candidate k-sequences.

Given L_{k-1}, the set of all frequent $(k-1)$-sequences, we want to generate a superset of the set of all frequent k-sequences. We first define the notion of a contiguous subsequence.

Definition Given a sequence $s = \langle\, s_1 s_2 ... s_n \,\rangle$ and a subsequence c, c is a *contiguous* subsequence of s if any of the following conditions hold:

1. c is derived from s by dropping an item from either s_1 or s_n.
2. c is derived from s by dropping an item from an element s_i which has at least 2 items.
3. c is a contiguous subsequence of c', and c' is a contiguous subsequence of s.

For example, consider the sequence $s = \langle\,(1,\ 2)\ (3,\ 4)\ (5)\ (6)\,\rangle$. The sequences $\langle\,(2)\ (3,\ 4)\ (5)\,\rangle$, $\langle\,(1,\ 2)\ (3)\ (5)\ (6)\,\rangle$ and $\langle\,(3)\ (5)\,\rangle$ are some of the contiguous subsequences of s. However, $\langle\,(1,\ 2)\ (3,\ 4)\ (6)\,\rangle$ and $\langle\,(1)\ (5)\ (6)\,\rangle$ are not.

We show in [7] that any data-sequence that contains a sequence s will also contain any contiguous subsequence of s. If there is no max-gap constraint, the data-sequence will contain all subsequences of s (including non-contiguous

Frequent 3-Sequences	Candidate 4-Sequences	
	after join	after pruning
⟨ (1, 2) (3) ⟩	⟨ (1, 2) (3, 4) ⟩	⟨ (1, 2) (3, 4) ⟩
⟨ (1, 2) (4) ⟩	⟨ (1, 2) (3) (5) ⟩	
⟨ (1) (3, 4) ⟩		
⟨ (1, 3) (5) ⟩		
⟨ (2) (3, 4) ⟩		
⟨ (2) (3) (5) ⟩		

Fig. 3. Candidate Generation: Example

subsequences). This property provides the basis for the candidate generation procedure.

Candidates are generated in two steps:

1. **Join Phase.** We generate candidate sequences by joining L_{k-1} with L_{k-1}. A sequence s_1 joins with s_2 if the subsequence obtained by dropping the first item of s_1 is the same as the subsequence obtained by dropping the last item of s_2. The candidate sequence generated by joining s_1 with s_2 is the sequence s_1 extended with the last item in s_2. The added item becomes a separate element if it was a separate element in s_2, and part of the last element of s_1 otherwise. When joining L_1 with L_1, we need to add the item in s_2 both as part of an itemset and as a separate element, since both ⟨ (x) (y) ⟩ and ⟨ (x y) ⟩ give the same sequence ⟨ (y) ⟩ upon deleting the first item. (Observe that s_1 and s_2 are contiguous subsequences of the new candidate sequence.)
2. **Prune Phase.** We delete candidate sequences that have a contiguous (k–1)-subsequence whose support count is less than the minimum support. If there is no max-gap constraint, we also delete candidate sequences that have any subsequence without minimum support.

The above procedure is reminiscent of the candidate generation procedure for finding association rules [2]; however details are quite different. A proof of correctness of this procedure is given in [7].

Example Figure 3 shows L_3, and C_4 after the join and prune phases. In the join phase, the sequence ⟨ (1, 2) (3) ⟩ joins with ⟨ (2) (3, 4) ⟩ to generate ⟨ (1, 2) (3, 4) ⟩ and with ⟨ (2) (3) (5) ⟩ to generate ⟨ (1, 2) (3) (5) ⟩. The remaining sequences do not join with any sequence in L_3. For instance, ⟨ (1, 2) (4) ⟩ does not join with any sequence since there is no sequence of the form ⟨ (2) (4 x) ⟩ or ⟨ (2) (4) (x) ⟩. In the prune phase, ⟨ (1, 2) (3) (5) ⟩ is dropped since its contiguous subsequence ⟨ (1) (3) (5) ⟩ is not in L_3.

3.2 Counting Candidates

While making a pass, we read one data-sequence at a time and increment the support count of candidates contained in the data-sequence. Thus, given a set of candidate sequences C and a data-sequence d, we need to find all sequences in C that are contained in d. We use two techniques to solve this problem:

1. We use a *hash-tree* data structure to reduce the number of candidates in C that are checked for a data-sequence.
2. We transform the representation of the data-sequence d so that we can efficiently find whether a specific candidate is a subsequence of d.

3.2.1 Reducing the number of candidates that need to be checked

We adapt the hash-tree data structure of [2] for this purpose. A node of the hash-tree either contains a list of sequences (a *leaf* node) or a hash table (an *interior* node). In an interior node, each non-empty bucket of the hash table points to another node. The root of the hash-tree is defined to be at depth 1. An interior node at depth p points to nodes at depth $p+1$.

Adding candidate sequences to the hash-tree. When we add a sequence s, we start from the root and go down the tree until we reach a leaf. At an interior node at depth p, we decide which branch to follow by applying a hash function to the pth item of the sequence. Note that we apply the hash function to the pth item, not the pth element. All nodes are initially created as leaf nodes. When the number of sequences in a leaf node exceeds a threshold, the leaf node is converted to an interior node.

Finding the candidates contained in a data-sequence. Starting from the root node, we find all the candidates contained in a data-sequence d. We apply the following procedure, based on the type of node we are at:

- *Interior node, if it is the root*: We apply the hash function to each item in d, and recursively apply this procedure to the node in the corresponding bucket. For any sequence s contained in the data-sequence d, the first item of s must be in d. By hashing on every item in d, we ensure that we only ignore sequences that start with an item not in d.
- *Interior node, if it is not the root*: Assume we reached this node by hashing on an item x whose transaction-time is t. We apply the hash function to each item in d whose transaction-time is in $[t - \text{window-size}, t + \max(\text{window-size}, \text{max-gap})]$ and recursively apply this procedure to the node in the corresponding bucket.
  To see why this returns the desired set of candidates, consider a candidate sequence s with two consecutive items x and y. Let x be contained in a transaction in d whose transaction-time is t. For d to contain s, the transaction-time corresponding to y must be in $[t-\text{window-size}, t+\text{window-size}]$ if y is part of the same element as x, or in the interval $(t, t+\text{max-gap}]$ if y is part of the next element. Hence if we reached this node by hashing on an item x with transaction-time t, y must be contained in a transaction whose transaction-time is in the interval $[t–\text{window-size}, t+\max(\text{window-size}, \text{max-gap})]$ for the data-sequence to support the sequence. Thus we only need to apply the hash function to the items in d whose transaction-times are in the above interval, and check the corresponding nodes.

- *Leaf node*: For each sequence s in the leaf, we check whether d contains s, and add s to the answer set if necessary. (We will discuss below exactly how to find whether d contains a specific candidate sequence.) Since we check each sequence contained in this node, we don't miss any sequences.

3.2.2 Checking whether a data-sequence contains a specific sequence

Let d be a data-sequence, and let $s = \langle s_1 ... s_n \rangle$ be a candidate sequence. We first describe the algorithm for checking if d contains s, assuming existence of a procedure that finds the first occurrence of an element of s in d after a given time, and then describe this procedure.

Contains test: The algorithm for checking if the data-sequence d contains a candidate sequence s alternates between two phases. The algorithm starts in the forward phase from the first element.

- **Forward phase:** The algorithm finds successive elements of s in d as long as the difference between the end-time of the element just found and the start-time of the previous element is less than max-gap. (Recall that for an element s_i, start-time(s_i) and end-time(s_i) correspond to the first and last transaction-times of the set of transactions that contain s_i.) If the difference is more than max-gap, the algorithm switches to the backward phase. If an element is not found, the data-sequence does not contain s.
- **Backward phase:** The algorithm backtracks and "pulls up" previous elements. If s_i is the current element and end-time(s_i) $= t$, the algorithm finds the first set of transactions containing s_{i-1} whose transaction-times are after $t -$ max-gap. The start-time for s_{i-1} (after s_{i-1} is pulled up) could be after the end-time for s_i. Pulling up s_{i-1} may necessitate pulling up s_{i-2} because the max-gap constraint between s_{i-1} and s_{i-2} may no longer be satisfied. The algorithm moves backwards until either the max-gap constraint between the element just pulled up and the previous element is satisfied, or the first element has been pulled up. The algorithm then switches to the forward phase, finding elements of s in d starting from the element after the last element pulled up. If any element cannot be pulled up (that is, there is no subsequent set of transactions which contain the element), the data-sequence does not contain s.

This procedure is repeated, switching between the backward and forward phases, until all the elements are found. Though the algorithm moves back and forth among the elements of s, it terminates because for any element s_i, the algorithm always checks whether a later set of transactions contains s_i; thus the transaction-times for an element always increase.

Example Consider the data-sequence shown in Figure 4. Consider the case when max-gap is 30, min-gap is 5, and window-size is 0. For the candidate-sequence $\langle (1, 2)\ (3)\ (4) \rangle$, we would first find (1, 2) at transaction-time 10, and then find (3) at time 45. Since the gap between these two elements (35 days)

Transaction-Time	Items
10	1, 2
25	4, 6
45	3
50	1, 2
65	3
90	2, 4
95	6

Fig. 4. Example Data-Sequence

Item	Times
1	$\rightarrow$ 10 $\rightarrow$ 50 $\rightarrow$ NULL
2	$\rightarrow$ 10 $\rightarrow$ 50 $\rightarrow$ 90 $\rightarrow$ NULL
3	$\rightarrow$ 45 $\rightarrow$ 65 $\rightarrow$ NULL
4	$\rightarrow$ 25 $\rightarrow$ 90 $\rightarrow$ NULL
5	$\rightarrow$ NULL
6	$\rightarrow$ 25 $\rightarrow$ 95 $\rightarrow$ NULL
7	$\rightarrow$ NULL

Fig. 5. Alternate Representation

is more than max-gap, we "pull up" (1, 2). We search for the first occurrence of (1, 2) after time 15, because end-time((3)) = 45 and max-gap is 30, and so even if (1, 2) occurs at some time before 15, it still will not satisfy the max-gap constraint. We find (1, 2) at time 50. Since this is the first element, we do not have to check to see if the max-gap constraint between (1, 2) and the element before that is satisfied. We now move forward. Since (3) no longer occurs more than 5 days after (1, 2), we search for the next occurrence of (3) after time 55. We find (3) at time 65. Since the max-gap constraint between (3) and (1, 2) is satisfied, we continue to move forward and find (4) at time 90. The max-gap constraint between (4) and (3) is satisfied; so we are done.

Finding a single element: To describe the procedure for finding the first occurrence of an element in a data sequence, we first discuss how to efficiently find a single item. A straightforward approach would be to scan consecutive transactions of the data-sequence until we find the item. A faster alternative is to transform the representation of d as follows.

Create an array that has as many elements as the number of items in the database. For each item in the data-sequence d, store in this array a list of transaction-times of the transactions of d that contain the item. To find the first occurrence of an item after time t, the procedure simply traverses the list corresponding to the item till it finds a transaction-time greater than t. Assuming that the dataset has 7 items, Figure 5 shows the tranformed representation of the data-sequence in Figure 4. This transformation has a one-time overhead of O(total-number-of-items-in-dataset) over the whole execution (to allocate and initialize the array), plus an overhead of O(no-of-items-in-d) for each data-sequence.

Now, to find the first occurrence of an element after time t, the algorithm makes one pass through the items in the element and finds the first transaction-time greater than t for each item. If the difference between the start-time and end-time is less than or equal to the window-size, we are done. Otherwise, t is set to the end-time minus the window-size, and the procedure is repeated.[2]

[2] An alternate approach would be to "pull up" previous items as soon as we find that the transaction-time for an item is too high. Such a procedure would be similar to the algorithm that does the contains test for a sequence.

Example Consider the data-sequence shown in Figure 4. Assume window-size is set to 7 days, and we have to find the first occurrence of the element (2, 6) after time $t = 20$. We find 2 at time 50, and 6 at time 25. Since end-time((2,6)) − start-time((2,6)) > 7, we set t to 43 (= end-time((2,6)) − window-size) and try again. Item 2 remains at time 50, while item 6 is found at time 95. The time gap is still greater than the window-size, so we set t to 88, and repeat the procedure. We now find item 2 at time 90, while item 6 remains at time 95. Since the time gap between 90 and 95 is less than the window size, we are done.

3.3 Taxonomies

The ideas presented in [6] for discovering association rules with taxonomies carry over to the current problem. The basic approach is to replace each data-sequence d with an "extended-sequence" d', where each transaction d'_i of d' contains the items in the corresponding transaction d_i of d, as well as all the ancestors of each item in d_i. For example, with the taxonomy shown in Figure 1, a data-sequence ⟨ (Foundation, Ringworld) (Second Foundation) ⟩ would be replaced with the extended-sequence ⟨ (Foundation, Ringworld, Asimov, Niven, Science Fiction) (Second Foundation, Asimov, Science Fiction) ⟩. We now run GSP on these "extended-sequences".

There are two optimizations that improve performance considerably. The first is to pre-compute the ancestors of each item and drop ancestors which are not in any of the candidates being counted before making a pass over the data. For instance, if "Ringworld", "Second Foundation" and "Niven" are not in any of the candidates being counted in the current pass, we would replace the data-sequence ⟨ (Foundation, Ringworld) (Second Foundation) ⟩ with the extended-sequence ⟨ (Foundation, Asimov, Science Fiction) (Asimov, Science Fiction) ⟩ (instead of the extended-sequence ⟨ (Foundation, Ringworld, Asimov, Niven, Science Fiction) (Second Foundation, Asimov, Science Fiction) ⟩). The second optimization is to not count sequential patterns with an element that contains both an item x and its ancestor y, since the support for that will always be the same as the support for the sequential pattern without y. (Any transaction that contains x will also contain y.)

A related issue is that incorporating taxonomies can result in many redundant sequential patterns. For example, let the support of "Asimov" be 20%, the support of "Foundation" 10% and the support of the pattern ⟨ (Asimov) (Ringworld) ⟩ 15%. Given this information, we would "expect" the support of the pattern ⟨ (Foundation) (Ringworld) ⟩ to be 7.5%, since half the "Asimov"s are "Foundation"s. If the actual support of ⟨ (Foundation) (Ringworld) ⟩ is close to 7.5%, the pattern can be considered "redundant". The interest measure introduced in [6] also carries over and can be used to prune such redundant patterns. The essential idea is that given a user-specified interest-level I, we display patterns that have no ancestors, or patterns whose actual support is at least I times their expected support (based on the support of their ancestors).

4 Performance Evaluation

We compared the performance of GSP to the AprioriAll algorithm given in [3], using both synthetic and real-life datasets. Due to lack of space, we only summarize the main results in this section. Details of the experiments, including performance graphs and detailed explanations of the results, can be found in [7].

Comparison of GSP and AprioriAll. On the synthetic datasets, GSP was between 30% to 5 times faster than AprioriAll, with the performance gap often increasing at low levels of minimum support. The results were similar on the three customer datasets, with GSP running 2 to 20 times faster than AprioriAll. There are two main reasons why GSP does better than AprioriAll.

1. GSP counts fewer candidates than AprioriAll.
2. AprioriAll has to first find which frequent itemsets are present in each element of a data-sequence during the data transformation, and then find which candidate sequences are present in it. This is typically somewhat slower than directly finding the candidate sequences.

Scaleup. GSP scales linearly with the number of data-sequences. For a constant database size, the execution time of GSP increases with the number of items in the data-sequence, but only gradually.

Effects of Time Constraints and Sliding Windows. To see the effect of the sliding window and time constraints on performance, we ran GSP on the three customer datasets, with and without the min-gap, max-gap, sliding-window constraints. The sliding-window was set to 1 day, so that the effect on the number of sequential patterns would be small. Similarly, the max-gap was set to more than the total time-span of the transactions in the dataset, and the min-gap was set to 1 day. We found that the min-gap constraint comes for "free"; there was no performance degradation due to specifying a min-gap constraint. However, there was a performance penalty of 5% to 30% for using the max-gap constraint or sliding windows.

5 Summary

We are given a database of sequences, where each sequence is a list of transactions ordered by transaction-time, and each transaction is a set of items. The problem of mining sequential patterns introduced in [3] is to discover all sequential patterns with a user-specified minimum support, where the support of a pattern is the number of data-sequences that contain the pattern.

We addressed some critical limitations of the earlier work in order to make sequential patterns useful for real applications. In particular, we generalized the definition of sequential patterns to admit max-gap and min-gap time constraints between adjacent elements of a sequential pattern. We also relaxed the restriction that all the items in an element of a sequential pattern must come from the same transaction, and allowed a user-specified window-size within which the items can be present. Finally, if a user-defined taxonomy over the items in the database is available, the sequential patterns may include items across different levels of the taxonomy.

We presented GSP, a new algorithm that discovers these generalized sequential patterns. It is a complete algorithm in that it guarantees finding all rules that have a user-specified minimum support. Empirical evaluation using synthetic and real-life data indicates that GSP is much faster than the AprioriAll algorithm presented in [3]. GSP scales linearly with the number of data-sequences, and has very good scale-up properties with respect to the average data-sequence size.

The GSP algorithm has been implemented as part of the Quest data mining prototype at IBM Research, and is incorporated in the IBM data mining product. It runs on several platforms, including AIX and MVS flat files, DB2/CS and DB2/MVS. It has also been parallelized for the SP/2 shared-nothing multiprocessor.

References

1. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, Washington, D.C., May 1993.
2. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.
3. R. Agrawal and R. Srikant. Mining Sequential Patterns. In *Proc. of the 11th Int'l Conference on Data Engineering*, Taipei, Taiwan, March 1995.
4. J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.
5. H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. of the Int'l Conference on Knowledge Discovery in Databases and Data Mining (KDD-95)*, Montreal, Canada, August 1995.
6. R. Srikant and R. Agrawal. Mining Generalized Association Rules. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.
7. R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. Research Report RJ 9994, IBM Almaden Research Center, San Jose, California, December 1995.
8. J. T.-L. Wang, G.-W. Chirn, T. G. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: Some preliminary results. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Minneapolis, May 1994.