GO-SPADE: Mining Sequential Patterns over Datasets with Consecutive Repetitions

Marion Leleu[1,2], Christophe Rigotti[1], Jean-François Boulicaut[1], and Guillaume Euvrard[2]

[1] Laboratoire d'Ingénierie des Systèmes d'Information, Bâtiment Blaise Pascal
INSA Lyon, 69621 Villeurbanne Cedex, France
{crigotti,jfboulic}@lisisun1.insa-lyon.fr
[2] Direction de la Stratégie, Informatique CDC, 113 rue Jean-Marin Naudin
F-92220 Bagneux, France
{marion.leleu,guillaume.euvrard}@caissedesdepots.fr

Abstract. Databases of sequences can contain consecutive repetitions of items. This is the case in particular when some items represent discretized quantitative values. We show that on such databases, a typical algorithm like the SPADE algorithm tends to loose its efficiency. SPADE is based on the used of lists containing the localization of the occurrences of a pattern in the sequences and these lists are not appropriated in the case of data with repetitions. We introduce the concept of *generalized occurrences* and the corresponding primitive operators to manipulate them. We present an algorithm called GO-SPADE that extends SPADE to incorporate generalized occurrences. Finally we present experiments showing that GO-SPADE can handle sequences containing consecutive repetitions at nearly no extra cost.

Keywords: frequent sequential pattern mining, generalized occurrences, SPADE

1 Introduction

Mining sequential patterns is an active data mining domain dedicated to sequential data. For example, customer purchases, Web log access, DNA sequences, geophysical data, and so on. The objective is to find all patterns satisfying some given criterion that can be hidden within a set of event sequences. Among the selection criterion proposed in the past (e.g., syntactic properties, similarity with a consensus pattern) the minimal frequency is still one of the most commonly used. Basically, the problem can be presented as follows: Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct items. Items are ordered by a total order on I. An *event* (also called *itemset*) of size l is a non empty set of l items from I : $(i_1 i_2 ... i_l)$, which is sorted in increasing order. A *sequence* α of *length* L is an ordered list of L events $\alpha_1, \ldots, \alpha_L$, denoted as $\alpha_1 \rightarrow \alpha_2 \rightarrow ... \rightarrow \alpha_L$. A database is composed of sequences, where each sequence has a unique sequence identifier (*sid*) and each event of each sequence has a temporal event identifier (*eid*) called timestamp. In a sequence, each *eid* is unique and if an event e_i precedes event e_j in a sequence,

P. Perner and A. Rosenfeld (Eds.): MLDM 2003, LNAI 2734, pp. 293–306, 2003.

then the *eid* of e_j must be strictly greater than the *eid* of e_i. Such a database can be represented by a table like, for example, the left table of Fig. 2. A *sequential pattern* (or *pattern*) is a sequence. We are interested in the so-called frequent sequential patterns defined as follows. A sequence $s_a = \alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n$ is called a *subsequence* of another sequence $s_b = \beta_1 \rightarrow \beta_2 \rightarrow \ldots \rightarrow \beta_m$ if and only if there exist integers $1 \leq i_1 < i_2 < \ldots < i_n \leq m$ such that $\alpha_1 \subseteq \beta_{i_1}$, $\alpha_2 \subseteq \beta_{i_2}$, ..., $\alpha_n \subseteq \beta_{i_n}$. Let N be a positive integer called *absolute support threshold*, a pattern p is frequent in a database D if p is a subsequence of at least N sequences of D. In this paper, we also use interchangeably *relative support threshold* expressed in the percentage of the number of sequences of D. A lot of work has been done since the introduction of the frequent sequential pattern mining problem in 1995 [2]. Each presents its own interests depending on the characteristics of the database to mine (e.g., [6,10,7,4,8,11,13,12]).
In this paper we consider the problem of mining frequent patterns in sequences where same items tend to be repeated in a consecutive way. This corresponds in particular to the important practical situation where databases are built in part from quantitative time series. In this case, these time series are discretized (using for example the method proposed in [3]) and the discrete values are encoded using items. This has an impact on the form of the resulting sequences that tend to contain more consecutive occurrences of the same items. Indeed, this research is motivated by sequential pattern mining from stock market data where we observed this situation [5]. For example, if items are used to encode a discretized stock price value having slow variations, we will often find in the sequences several consecutive occurrences of the same item. As far as we know, no specific work has been done to tailored the current algorithms towards this kind of data containing repetitions. Figure 1 shows the behavior of the SPADE algorithm [11,13] (a typical sequential pattern mining algorithm) on such datasets. The results of the experiments presented in Fig. 1 correspond to extractions on two datasets: $set1\_r0$ and $set1\_r5$. $set1\_r5$ contains the same sequences that $set1\_r0$ in which a few additional consecutive repetitions of some items have been added (see Sect. 5.1 for a description of these datasets). The curves of Fig. 1 represent the costs (in term of execution time) for the extraction of different amounts of frequent patterns on each dataset, i.e., for different support thresholds. These curves show that to extract a given number of frequent patterns, SPADE execution time is much more important on the dataset containing more consecutive repetitions ($set1\_r5$).

The main contribution of this paper is to show that this extra extraction cost can be reduced drastically by using a more compact information representation. We propose such a representation and present an extension of SPADE, called GO-SPADE, that operates directly on it. We show that in practice it can be used to handle efficiently the consecutive repetitions of items in the data. This practical interest can be seen in particular the bottom right graph on Fig. 5 that presents the same experiments than Fig. 1 using both SPADE and GO-SPADE. This figure shows notably that the presence of consecutive repetitions has nearly no impact on GO-SPADE extraction time for a given amount of frequent patterns.

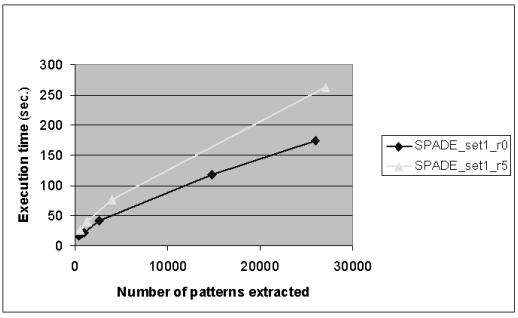


Fig. 1. Evolution of SPADE execution time on datasets with consecutive repetitions

This paper is organized as follows. Section 2 gives an overview of related work in the sequential pattern mining field. Section 3 presents in a synthetic way the SPADE algorithm before to introduce in Sect. 4 our contribution which is a novel SPADE-based algorithm. Section 5 presents experimental results that illustrate how GO-SPADE gains in efficiency compared to SPADE in the case of datasets presenting consecutive repetitions. We conclude in Sect. 6 by a summary and directions for future work.

2 Related Work

In the data mining community, the computation of the sequential patterns has been studied since 1995, e.g., [6,10,7,4,8,11,13,12]. It has lead to several algorithms that can process huge sets of sequences. These algorithms use three different types of approaches according to the way they evaluate the support of sequential pattern candidates. The first family contains algorithms that are based on the A-Priori scheme [1] and that perform a full scan of the database to evaluate the support of the current candidates, e.g., [2,10,7]. In these approaches, a particular effort is made to develop specific structures to represent the sequential patterns candidates to speed-up the support counting operations (e.g., the dedicated hash tree used in [10]). The second family (e.g., [4,8]) contains algorithms that try to reduce the size of the dataset to be scaned by performing projections of the initial database. The last family (e.g., [11,13,12]) concerns algorithms that keep in memory only the information needed for the support evaluation. These algorithms are based on the so called *occurrence lists* which contain the descriptions of the location where the pattern occur in the dataset. The projection database and occurrence list approaches seem to be more efficient than the first one in the case of low support threshold and long sequential patterns since the occurrence lists and the projected databases become more and more smaller. As far as we know, no comparative studies has been done enabling to affirm whether one approach is definitely better than the others. In the frequent itemset extraction field, these three families also exist (e.g., [9,14, 1]) and according to the experimental results of [14], it seems that techniques

based on occurrence lists are more efficient at very low support thresholds (while this is not always the case for higher thresholds).

Databases containing consecutive repetitions of items present a new specific problem and, to our knowledge, has not been studied yet. We propose an algorithm based on SPADE [11,13]. It uses *generalized occurrences* lists to represent consecutive occurrences of patterns.

3 The SPADE Algorithm

In this section, we recall the principle of the SPADE algorithm [11,13]. SPADE repeats two basic operations: a generation of candidate patterns and a support counting step. Let us introduce some needed concepts. A pattern with k items is called a *k-pattern*. For example, the pattern B $\rightarrow$ ACD $\rightarrow$ CDFG is a 8-pattern. A *prefix* of a k-pattern z is a subpattern of z constituted by the $k-1$ first items of z (items in the last event of z are ordered according to the lexicographical order) and its *suffix* corresponds to its last item. For example, the *prefix* of the pattern A $\rightarrow$ BC is the subpattern A $\rightarrow$ B and its *suffix* is item C. SPADE uses two frequent k-patterns z_1 and z_2 having the same $(k-1)$-pattern as prefix to generate a $(k+1)$-pattern z. We denote this operation as $merge(z_1, z_2)$. The support counting for the newly generated pattern is not made by scanning the whole database. Instead, SPADE has stored in specific lists, called $IdLists$, the positions where z_1 and z_2 occur in the database. It then uses these two lists denoted $IdList(z_1)$ and IdList(z_2) to determine where z occurs. Then $IdList(z)$ allows to compute directly the support of z. The computation of $IdList(z)$ is a kind of *join* and is denoted $join(z_1, z_2)$. There are several different *merge* and *join* operations used depending on the form of z_1 and z_2 for *merge* and on the form of z_1, z_2 and z for *join*. Before describing in more details these operations and the structure of $IdLists$ we give an abstract formulation of SPADE (algorithm 1).

To reduce the memory consumption and to enhance the efficiency, the SPADE algorithm uses various important optimizations (in particular a notion of equivalence class of patterns, dedicated breadth-first and depth-first search strategies and also a specific processing for 1-patterns and 2-patterns). These optimizations are not related to the problem tackled in this paper and we refer the reader to [11, 13] for their descriptions.

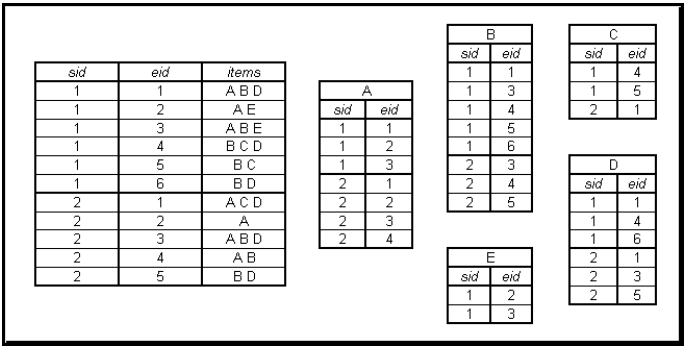
The $IdList$ of a pattern z contains only the information needed to compute the support of z and the $IdLists$ of the patterns that will be generated using z. $IdList(z)$ is a set of pairs $\langle sid, eid \rangle$, each pair describing an occurrence y of z in the database. sid is the identifier of the sequence containing y and eid is the timestamp of the last event of y. Examples of $IdLists$ for 1-patterns are given in Fig. 2 and for the same database, the two Right-Tables of Fig. 3 present examples of $IdLists$ for the 2-patterns C $\rightarrow$ D and CD.

The support of pattern z is obtained by counting the number of distinct $sids$ present in $IdList(z)$. For example, in Fig. 2, the support of A and E are respectively 2 and 1.

During the *merge* operation of the generation step, SPADE distinguishes two kinds of patterns: *sequence patterns* and *event patterns*, depending on the temporal relation between the *prefix* and the *suffix* of a pattern. A pattern having

Algorithm 1 (abstract SPADE)
Input: *a database of event sequences and a support threshold.*
Output: *the frequent sequential patterns contained in the database.*

Use the database to compute:
- F_1 the set of all frequent items
- $IdList(z)$ for all element z of F_1
let $i := 1$
while $F_i \neq \emptyset$ **do**
  let $F_{i+1} := \emptyset$
  for all $z_1 \in F_i$ **do**
    for all $z_2 \in F_i$ **do**
      if z_1 *and* z_2 *have the same prefix* **then**
        for all z *obtained by* $merge(z_1, z_2)$ **do**
          Compute $IdList(z)$ *by* $join(IdList(z_1), IdList(z_2))$.
          Use $IdList(z)$ *to determine if* z *is frequent.*
          if z *is frequent* **then**
            $F_{i+1} := F_{i+1} \cup \{z\}$
          fi
        od
      fi
    od
  od
  $i := i + 1$
od
output $\bigcup_{1 \leq j < i} F_j$



sid	eid	items
1	1	A B D
1	2	A E
1	3	A B E
1	4	B C D
1	5	B C
1	6	B D
2	1	A C D
2	2	A
2	3	A B D
2	4	A B
2	5	B D

A

sid	eid
1	1
1	2
1	3
2	1
2	2
2	3
2	4

B

sid	eid
1	1
1	3
1	4
1	5
1	6
2	3
2	4
2	5

C

sid	eid
1	4
1	5
2	1

D

sid	eid
1	1
1	4
1	6
2	1
2	3
2	5

E

sid	eid
1	2
1	3

Fig. 2. A database and IdList for items A, B, C, D and E

prefix p and suffix s is called an *event pattern*, denoted ps if s occurs at the same time than the last item of p. If s occurs strictly after the last item of p, the pattern is called a *sequence pattern* and is denoted $p \rightarrow s$. For example, pattern AB $\rightarrow$ C $\rightarrow$ BDF having pattern AB $\rightarrow$ C $\rightarrow$ BD as *prefix* and item F as *suffix* is an *event pattern*. Pattern AB $\rightarrow$ C whose prefix is AB and suffix is C is a *sequence pattern*.

Let z_1 and z_2 be patterns having the same prefix p with respective suffix s_1 and s_2. The *merge* operation used to generate a new pattern depends on the form of z_1 and z_2 (i.e., an event pattern or a sequence pattern). The form of z determines the kind of *join* performed to compute $IdList(z)$ from $IdList(z_1)$ and $IdList(z_2)$. If z is an event pattern (resp. a sequence pattern) the *join* is made using a procedure called $EqualityJoin$ (resp. $TemporalJoin$). We present these generation cases and then describe the *join* operations.

- when z_1 and z_2 are *event patterns* (generation case 1):
  z_1 and z_2 are of the forms $z_1 = ps_1$ and $z_2 = ps_2$. The pattern generated by *merge* is $z = ps_1s_2$ and its $IdList = EqualityJoin(IdList(z_1),IdList(z_2))$.
- when z_1 is an *event pattern* and z_2 a *sequence pattern* (generation case 2):
  z_1 and z_2 are of the forms $z_1 = ps_1$ and $z_2 = p \rightarrow s_2$. The pattern generated by *merge* is $z = ps_1 \rightarrow s_2$ and we have $IdList(z)$=$TemporalJoin(IdList(z_1)$, $IdList(z_2))$.
- when z_1 and z_2 are *sequence patterns*: z_1 and z_2 are of the forms $z_1 = p \rightarrow s_1$ and $z_2 = p \rightarrow s_2$. If $s_1 \neq s_2$, three patterns are generated:
  - (generation case 3) the pattern generated by *merge* is $z = p \rightarrow s_1s_2$ and $IdList(z) = EqualityJoin(IdList(z_1),IdList(z_2))$.
  - (generation case 4) the pattern generated by *merge* is $z = p \rightarrow s_1 \rightarrow s_2$ and $IdList(z) = TemporalJoin(IdList(z_1),IdList(z_2))$.

  If $s_1 = s_2$ and $z_1 = z_2 = p \rightarrow s_1$ (generation case 5), there is only one generated pattern $z = p \rightarrow s_1 \rightarrow s_1$ and $IdList(z) = TemporalJoin(IdList(z_1)$, $IdList(z_2))$.

The two *join* operations are defined as follows:
Computation of $IdList(z)$ using $TemporalJoin(IdList(z_1),IdList(z_2))$: For each pair $\langle s_1, e_1\rangle$ in $IdList(z_1)$ and each pair $\langle s_2, e_2\rangle$ in $IdList(z_2)$ check if $\langle s_1, e_1\rangle$ represents an occurrence y_1 preceeding the occurrence y_2 represented by $\langle s_2, e_2\rangle$ in a sequence (i.e., $s_1 = s_2$ and $e_1 < e_2$). If this is the case, it means that the events in y_1 and y_2 form an occurrence of z, then add $\langle s_1, e_2\rangle$ to $IdList(z)$.

Computation of $IdList(z)$ using $EqualityJoin(IdList(z_1),IdList(z_2))$: For each pair $\langle s_1, e_1\rangle$ in $IdList(z_1)$ and each pair $\langle s_2, e_2\rangle$ in $IdList(z_2)$ check if $\langle s_1, e_1\rangle$ represents an occurrence y_1 ending at the same time than the last event of occurrence y_2 represented by $\langle s_2, e_2\rangle$ (i.e., $s_1 = s_2$ and $e_1 = e_2$). If this is the case, y_1 and y_2 form an occurrence of z and then add $\langle s_1, e_1\rangle$ to $IdList(z)$.

We now describe on an example how these joins are performed. Let consider the *IdList* of items C and D represented in Fig. 3 (from the example database of Fig. 2). The *IdList* of pattern C $\rightarrow$ D is obtained performing a $TemporalJoin$ between $IdList$(C) and $IdList$(D) as follows: for a given pair (s, e_1) in $IdList$(C), SPADE checks whether there exists a pair (s, e_2) in $IdList$(D)

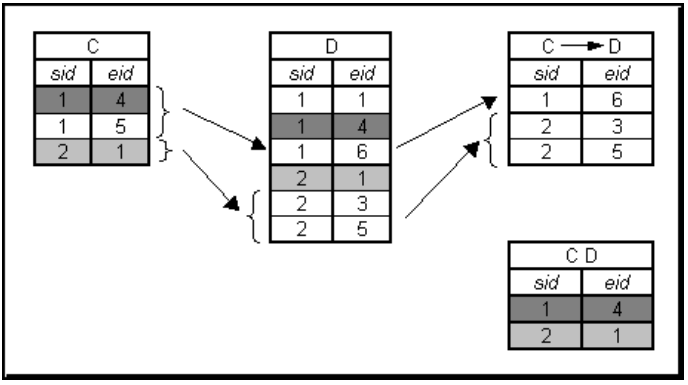


Fig. 3. Temporal and equality joins on IdList(C) and IdList(D)

with $e_2 > e_1$, which means that item D follows the item C in the sequence s. If this is true, then the pair (s, e_2) is added to the *IdList* of pattern C $\rightarrow$ D. The resulting list is represented in Fig. 3. The *IdList* of pattern CD is computed by *EqualityJoin*(*IdList*(C),*IdList*(D)) and is depicted on Fig. 3. This *EqualityJoin* is performed as follows: for a given pair (s, e_1) in *IdList*(C), SPADE checks whether there exists a pair (s, e_2) in *IdList*(D) with $e_2 = e_1$, which means that item D occurs at the same time than item C in the sequence s. If this is true, then the pair (s, e_2) is added to the *IdList* of pattern CD.

4 The GO-SPADE Algorithm

4.1 Motivations

Let us revisit the example of Fig. 2 and consider the *IdList* for item A. This item occurs in a consecutive way in the sequences: at *eid* 1, 2 and 3 in the first sequence and at *eid* 1, 2, 3 and 4 in the second one. Such a situation can appear in several kind of databases in particular when the events come from some quantitative data such as time series with smooth variations. SPADE *IdList* stores one line per occurrence, that is 3 lines for the occurrences of item A in sequence 1 and 4 lines for sequence 2. We introduce the concept of *generalized occurrence* to compact all these consecutive occurrences. For example, the 3 consecutive occurrences of item A in sequence 1 can be represented by only one generalized occurrence of the form $\langle 1, [1, 3]\rangle$ containing the sequence identifier (i.e., 1) and an interval [1,3] containing all the *eids* of the consecutive occurrences. When the pattern contains several events, the interval contains all *eids* of the consecutive locations of the last event. For example, for pattern A $\rightarrow$ B, its four occurrences in sequence 1 in Fig. 4 are represented by the single generalized occurrence $\langle 1, [3, 6]\rangle$.

Using such a representation enables to reduce significantly the size of the *IdLists*, as soon as some consecutive occurrences appear in the database. This compact form of *IdList* containing generalized occurrences is termed *GoIdList*.

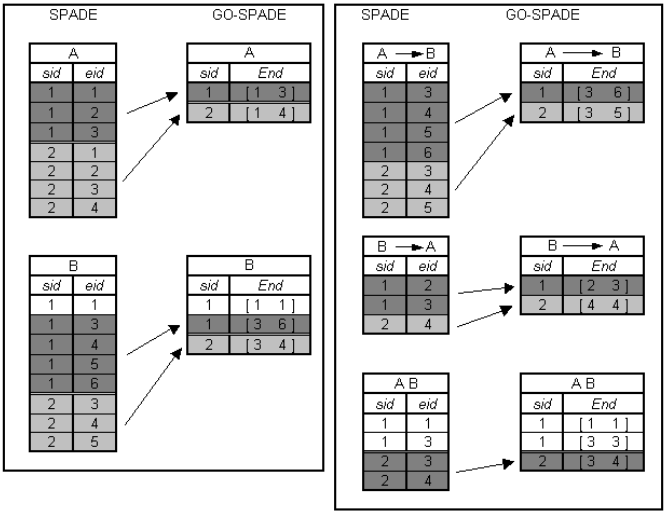


Fig. 4. *GoIdList* vs. *IdList*

Figure 4 illustrates these reductions and also shows how these reductions are propagated during the join operations. For example, *IdList*(A) contains 7 occurrences while the *GoIdList*(A) contains only 2 generalized occurrences. This figure also presents the reductions obtained for *IdList*(B) and for the *IdLists* of A $\rightarrow$ B, B $\rightarrow$ A and AB resulting from *TemporalJoin* and *EqualityJoin* operations on *IdList*(A) and *IdList*(B).
In the following, we present our new algorithm, GO-SPADE based on new join operations using *GoIdLists*.

This approach not only reduces the memory space used during an extraction process, it also reduces significantly the join cost , and thus the overall execution time. These effects (memory and time gains) will be described and analyzed in Sect. 5. For example,

4.2 GO-IdList: An IdList of Generalized Occurrences

A *generalized occurrence* represents in a compact way several occurrences of a pattern z, and contains the following informations:

- An identifier *sid* that corresponds to identifier of a sequence where pattern z occurs.
- An interval [*min*,*max*] corresponding to consecutive occurrences of the last event of pattern z.

Such a *generalized occurrence* is denoted as a tuple $\langle sid, [min, max] \rangle$.

A *GoIdList* is a list containing all the generalized occurrences of a sequential pattern. The generalized occurrence list of the sequential pattern z is denoted by *GoIdList*(z).

4.3 GO-SPADE Algorithm

The overall principle of GO-SPADE is the same that the one of SPADE presented in Algorithm 1. The generation process remains the same as in SPADE (i.e., a new pattern z is generated from two generator patterns z_1 and z_2 sharing a same prefix p).

The difference between the two algorithms is that in GO-SPADE the occurrences of the patterns are stored in generalized occurrence lists and that the $TemporalJoin$ and $EqualityJoin$ computations are replaced by dedicated procedures operating on this generalized form of occurrence.

We now present the new $TemporalJoin$ in Algorithm 2 and, in Algorithm 4, the new $EqualityJoin$.

Algorithms 2 and 4 generate a new $GoIdList$ from the $GoIdLists$ of two generator patterns z_1 and z_2. They proceed in a similar way. The nested loops of lines 1 and 2 iterate on the elements of $GoIdList(z_1)$ and $GoIdList(z_2)$. For each pair $(\langle sid_1, [min_1, max_1]\rangle, \langle sid_2, [min_2, max_2]\rangle)$, the algorithms call a function to join these two generalized occurrences using respectively $LocalTemporalJoin$ (algorithm 3) and $LocalEqualityJoin$ (algorithm 5). Algorithm 2 just checks before that $min_1 < max_2$ in order to verify that at least one occurrence of $\langle sid_1, [min_1, max_1]\rangle$ terminates before the end of at least one occurrence of $\langle sid_2, [min_2, max_2]\rangle$. Test in line 5 (resp. line 4) verifies that the generalized occurrence returned by $LocalTemporalJoin$ (resp. $LocalEqualityJoin$) is valid. If it is the case, then it can be added to the current generated $GoIdList$ (line 6, resp. line 5). These algorithms terminate after having proceeded with all couples of generalized occurrences $(\langle sid_1, [min_1, max_1]\rangle, \langle sid_2, [min_2, max_2]\rangle)$ returning the computed $GoIdList$.

Algorithm 2 ($TemporalJoin$)
Input: $GoIdList(z_1)$, $GoIdList(z_2)$
Used subprograms: $LocalTemporalJoin$
Output: *a new GoIdList*

Initialize GoIdList to the empty list.
1. **for all** $occ_1 \in GoIdList(z_1)$ **do**
2. **for all** $occ_2 \in GoIdList(z_2)$ **do**
3. **if** $(min_1 < max_2)$ **then**
4. **let** $\langle v, add\rangle := LocalTemporalJoin$
$(\langle sid_1, [min_1, max_1]\rangle,$
$\langle sid_2, [min_2, max_2]\rangle)$
5. **if** *add* **then**
6. *Insert v in GoIdList*
7. **fi**
8. **fi**
9. **od**
10. **od**
11. **output** *GoIdList*

Algorithm 3 ($LocalTemporalJoin$)
Input: *Two generalized occurrences* $\langle sid_1, [min_1, max_1]\rangle$ *and* $\langle sid_2, [min_2, max_2]\rangle$
Output: $\langle v, add\rangle$ *where:* $v = \langle sid, [min, max]\rangle$ *and* add*, a boolean value that is false if v cannot be created.*

1. **let** *add := false*
2. **let** *v := null*
3. **if** $(sid_1 = sid_2)$ **then**
4. **find** *min the minimum element x of* $[min_2, max_2]$ *such that* $x > min_1$
5. **let** $sid := sid_1$
6. **let** $max := max_2$
7. **let** $v := \langle sid, [min, max]\rangle$
8. **let** *add := true*
9. **fi**
10. **output** $\langle v, add\rangle$

Algorithm 4 (*EqualityJoin*)
Input: $GoIdList(z_1)$, $GoIdList(z_2)$
Used subprograms: *LocalEqualityJoin (Algorithm 5)*
Output: *a new GoIdList*

```
1. for all occ1 ∈ GoIdList(z1) do
2.   for all occ2 ∈ GoIdList(z2) do
3.       let ⟨v, add⟩ :=
           LocalEqualityJoin(occ1,occ2)
4.       if add then
5.         Insert v in GoIdList
6.       fi
7.   od
8. od
9. output GoIdList
```

Algorithm 5 (*LocalEqualityJoin*)
Input: *Two generalized occurrences* $\langle sid_1, [min_1, max_1]\rangle$ *and* $\langle sid_2, [min_2, max_2]\rangle$
Output: $\langle v, add\rangle$ *where:* $v = \langle sid, [min, max]\rangle$ *and* add, *a boolean value that is false if v cannot be created.*

```
1. let add := false
2. let v := null
3. if (sid1 = sid2) then
4.   if (min1 ≤ max2 and max1 ≥ min2 ) then
5.     let sid := sid1
6.     let min := max(min1,min2)
7.     let max := min(max1,max2)
8.     let v := ⟨sid, [min, max]⟩
9.     let add := true
10.  fi
11. fi
12. output ⟨v, add⟩
```

Algorithm 3, *LocalTemporalJoin*, generates a new generalized occurrence from the two input ones. It first verifies that the two generalized occurrences are from a same sequence, that is $sid_1 = sid_2$ (line 3). Lines 4 to 8 generate a new generalized occurrence. Line 4 sets the min value of the generalized occurrence to be created with the minimum element of $[min_2, max_2]$ which is greater than min_1. This means that min is the first occurrence of the generalized occurrence $\langle sid_2, [min_2, max_2]\rangle$ that strictly follows the first occurrence of $\langle sid_1, [min_1, max_1]\rangle$. Secondly, Line 5 sets the sid value. Then, line 6 sets the max value of the new generalized occurrence to max_2 (the location of the last occurrence corresponding to z_2).

Algorithm 5, *LocalEqualityJoin*, first verifies that the two generalized occurrences come from the same sequence and then checks in line 4 if the intersection of the two intervals $[min_1, max_1]$ and $[min_2, max_2]$ is empty or not. If the intersection is not empty, it means that there exists occurrences of the new pattern ending at each eid in this intersection. Then the algorithm sets $[min,max]$ to the intersection of $[min_1, max_1]$ and $[min_2, max_2]$, and sets the value of sid.

4.4 Soundness and Completness

Definition 1. ***(v represents y)*** Let y be an occurrence of pattern z in a sequence S from a database β. Let $GoIdList(z)$ be the generalized occurrence list of this pattern and let v be one generalized occurrence from $GoIdList(z)$ denoted by the tuple $\langle sid, [min, max]\rangle$. We say that v *represents* y if $sid(v) = Id(S)$ and $min \leq end(y) \leq max$ where $end(y)$ denotes the eid of the last event of y.

Definition 2. ***(soundness)*** Let S be a sequence of β and z be a pattern with its generalized occurrence list $GoIdList(z)$. $GoIdList(z)$ is *sound* if for all v in

$GoIdList(z)$, where v is of the form $\langle sid, [min, max]\rangle$ with $sid(v) = \mathrm{Id}(S)$, we have: for all integer t_f in [min,max], there exists an occurrence of z in S such that $end(y) = t_f$.

Theorem 1. *For all patterns z, the GoIdList(z) generated by GO-SPADE is sound.*

Definition 3. ***(completness)*** Let z be a pattern, $GoIdList(z)$ its generalized occurrence list. $GoIdList(z)$ is *complete* if for all S in β and for all y such that y is an occurrence of z in S, then there exists v in $GoIdList(z)$ such that v *represents* y

Theorem 2. *For all patterns z, the GoIdList(z) generated by GO-SPADE is complete.*

The following theorem follows directly from Theorem 1 and 2.

Theorem 3. ***(correctness)*** *For all patterns z, the support determined by GO-SPADE using GoIdList is the same as the support determined by SPADE using IdList.*

5 Experimental Results

We present experimental results showing that the behavior of SPADE algorithm is greatly enhanced by the use of generalized occurrences when datasets contain consecutive repetitions. Both GO-SPADE and SPADE algorithms have been implemented using Microsoft Visual C++ 6.0, with the same kind of low level optimization to allow a fair comparison. All experiments have been performed on a PC with 196 MB of memory and a 500 MHz Pentium III processor under Microsoft Windows 2000.

The experimentations have been run on synthetic datasets generated using the Dataquest generator of IBM [2]. Two datasets have been generated using the following parameters: C10-T2.5-S4-I1.25-D1K over an alphabet of 100 items (called $set1$) and C10-T2.5-S4-I1.25-D10K over an alphabet of 1000 items (called $set2$). The first one contains 1000 sequences, the second one 10000 sequences. In both cases, the average size of the sequences is 10 (see [2] for more details on the generator parameters). In these datasets, the time interval between two time stamps is 1, and there is one event per time stamp.

In order to have datasets presenting parameterized consecutive repetitions on certain items, we performed a post-processing on $set1$ and $set2$. Each item founded in an event of a sequence has a probability fixed to 10% to be repeated. When an item is repeated, we simply duplicate it in the next i consecutive events. If the end of the sequence is reached during the duplication process the sequence is not extended (no new event is created) and thus, the current item is not completely duplicated. For dataset $set1$ (resp. $set2$) we denote $set1\_r\{i\}$ (resp. $set2\_r\{i\}$) the dataset obtained with a repetition parameter of value i. For the sake of uniformity, $set1$ (resp. $set2$) is denoted $set1\_r0$ (resp.

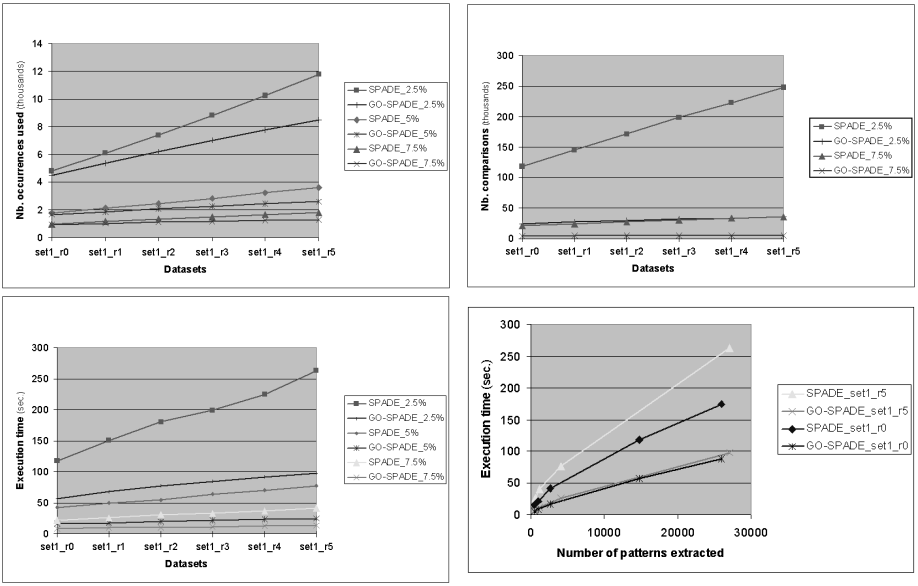


Fig. 5. Evolution of the total number of occurrences used (top left), of the total number of comparisons (top right) and of the total execution time (bottom left). Influence of consecutive repetitions on SPADE vs. Go-SPADE (bottom right)

$set2\_r0$). The post-processing on $set1\_r0$ leads to the creation of 5 new datasets $set1\_r1$,..., $set1\_r5$. They all have been created simultaneously, to repeat the same items in all the new datasets. For example, if item A occurring in sequence 10 at timestamp 5 is chosen to be repeated, then it will be added to event at timestamp 6 in sequence 10 in $set1\_r1$, and to events at timestamps 6 and 7 in sequence 10 in $set1\_r2$, and so on.

5.1 Generalized Occurrences Impact on the List Sizes

Generalized occurrences represent in a compact way all consecutive occurrences that can be found in a sequence database. The top left graph of Fig. 5 shows the sizes of *IdLists* and *GoIdLists* (in number of elements) for extractions performed on files $set1\_r0$, $set1\_r1$, ..., $set1\_r5$ using several support thresholds (7.5%, 5% and 2.5%). The number of occurrences used by SPADE is greater than the number of generalized occurrences used by GO-SPADE. As expected, this reduction is more important when the consecutive repetition parameter increases.

5.2 Generalized Occurrences Impact on the Join Costs

As shown in the previous experiments, the size of *GoIdList* is smaller than the size of *IdList*. This reduction has a direct impact on the join costs. Indeed,

let n_s and m_s be the number of occurrences of two generator patterns in a sequence s. In the worst case, and assuming that there are $nbSeq$ sequences in the database, the number of comparisons needed to perform one join between these two generator patterns is $\sum_s n_s m_s$, $s \in [1, \ldots, nbSeq]$. Suppose now that all n_s and m_s are reduced by an average factor of $\gamma \leq 1$, then the number of comparisons becomes $\sum_s \gamma^2 n_s m_s$, $s \in [1, \ldots, nbSeq]$. In this case, the number of comparisons used by GO-SPADE is reduced by γ^2 compared to SPADE.
The top right graph of Fig. 5 shows this reduction in practice during extractions performed on $set1\_r0$ to $set1\_r5$ with support thresholds 2.5% and 7.5%. For example, the cost in term of number of comparisons needed during a GO-SPADE extraction at 2.5% is significantly lower than the cost for SPADE at the same support threshold and furthermore is close to the cost for SPADE extraction at 7.5%.

5.3 Generalized Occurrences Impact on the Execution Time

The reduction of the list sizes and the reduction of the comparison number enable to greatly reduce the overall execution time of extractions. This is illustrated on the bottom left graph of Fig. 5, that presents the execution time of SPADE and GO-SPADE on datasets $set1\_r0$ to $set1\_r5$ for support thresholds 2.5%, 5% and 7.5%.

In Fig. 1 (Sect. 1), we have presented how the time needed by SPADE (to extract a given number of patterns) increases in presence of sequences containing consecutive repetitions. The bottom right graph of Fig. 5 completes these results with the corresponding times for GO-SPADE. It shows that the execution time of GO-SPADE to find a given number of patterns remains quite the same in presence of repetitions.

6 Conclusion and Future Works

We considered databases of sequences presenting some consecutive repetition of items. We showed that the SPADE algorithm [11,13], a typical sequential pattern extraction algorithm, turns out to become significantly less efficient on this kind of databases. SPADE is based on lists containing information about the localization of the patterns in the sequences. The consecutive repetitions lead to a defavorable growth of the size of these occurrence lists and thus increase the total extraction time. We defined a notion of generalized occurrences to handle in a compact way the pattern localizations. We propose an algorithm, called GO-SPADE, that extends SPADE to handle these generalized occurrences. Finally, we showed by means of experiments that GO-SPADE remains efficient when used on sequences containing consecutive repetitions. In the data mining community, the frequent sequential pattern extraction process has been enhanced by the consideration of other constraints that the minimal frequency to specify beforehand the relevancy of extracted patterns. These constraint specifications can be used to reduce both the number of extracted patterns and the search space. The c-SPADE algorithm [12], a constrained version of SPADE, is an example of such a constrained-base sequential pattern mining algorithm. A promising direction for future work is to extend c-SPADE with an appropriated form of

generalized occurrences to process efficiently sequences with consecutive repetitions. Furthermore, we can now proceed with the real data about stock market analysis that has motivated this research.

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the VLDB Conference*, Santiago, Chile, September 1994.
2. R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the 11th International Conference on Data Engineering (ICDE'95)*, pages 3–14, Taipei, Taiwan, March 1995. IEEE Computer Society.
3. G. Das, L. K.I., H. Mannila, G. Renganathan, and P. Padhraic Smyth. Rule discovery from time series. In *Proc. of the 4th International Conference on Knowledge Discovery and Data Mining (KDD'98)*, pages 16–22, New York (USA), August 1998. AAAI Press.
4. J. Han, J. Pei, B. Han Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. In *Proc. 2000 Int. Conf. Knowledge Discovery and Data Mining (KDD'00)*, pages 355–359, August 2000.
5. M. Leleu and J. Boulicaut. Signing stock market situations by means of characteristic sequential patterns. In *Proc. of the 3rd International Conference on Data Mining (DM'02)*, Bologna, Italy, September 2002. WIT Press.
6. H. Mannila, H. Toivonen, and A. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–298, November 1997.
7. F. Masseglia, C. F., and P. P. The PSP approach for mining sequential patterns. In *Proc. of the 2nd European Symposium on Principles of Data Mining and Knowledge Discovery in Databases (PKDD'98)*, pages 176–184, Nantes, France, September 1998. Lecture Notes in Artificial Intelligence, Springer Verlag.
8. J. Pei, B. Han, B. Mortazavi-Asl, and H. Pinto. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. of the 17th International Conference on Data Engineering (ICDE'01)*, 2001.
9. J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, May 2000.
10. R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the 5th International Conference on Extending Database Technology (EDBT'96)*, pages 3–17, Avignon, France, September 1996.
11. M. Zaki. Efficient enumeration of frequent sequences. In *Proc. of the 7th International Conference on Information and Knowledge Management (CIKM'98)*, pages 68–75, November 1998.
12. M. Zaki. Sequence mining in categorical domains: incorporating constraints. In *Proc. of the 9th International Conference on Information and Knowledge Management (CIKM'00)*, pages 422–429, Washington, DC, USA, November 2000.
13. M. Zaki. Spade: an efficient algorithm for mining frequent sequences. *Machine Learning, Special issue on Unsupervised Learning*, 42(1/2):31–60, Jan/Feb 2001.
14. M. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proc. of the 2nd SIAM International Conference on Data Mining*, Arlington, Virginia , USA, April 2002.