ERMiner: Sequential Rule Mining Using Equivalence Classes

Philippe Fournier-Viger[1], Ted Gueniche[1], Souleymane Zida[1], and Vincent S. Tseng[2]

[1] Dept. of Computer Science, University of Moncton, Canada
[2] Dept. of Computer Science and Information Engineering, National Cheng Kung University, Taiwan
{philippe.fournier-viger,esz2233}@umoncton.ca, ted.gueniche@gmail.com, tsengsm@mail.ncku.edu.tw

Abstract. Sequential rule mining is an important data mining task with wide applications. The current state-of-the-art algorithm (RuleGrowth) for this task relies on a pattern-growth approach to discover sequential rules. A drawback of this approach is that it repeatedly performs a costly database projection operation, which deteriorates performance for datasets containing dense or long sequences. In this paper, we address this issue by proposing an algorithm named ERMiner (Equivalence class based sequential Rule Miner) for mining sequential rules. It relies on the novel idea of searching using equivalence classes of rules having the same antecedent or consequent. Furthermore, it includes a data structure named SCM (Sparse Count Matrix) to prune the search space. An extensive experimental study with five real-life datasets shows that ERMiner is up to five times faster than RuleGrowth but consumes more memory.

Keywords: sequential rule mining, vertical database format, equivalence classes, sparse count matrix.

1 Introduction

Discovering interesting sequential patterns in sequences is a fundamental problem in data mining. Many studies have been proposed for mining interesting patterns in sequence databases [12]. Sequential pattern mining [1] is probably the most popular research topic among them. It consists of finding subsequences appearing frequently in a set of sequences. However, knowing that a sequence appears frequently is not sufficient for making predictions [4]. An alternative that addresses the problem of prediction is sequential rule mining [4]. A sequential rule indicates that if some item(s) occur in a sequence, some other item(s) are likely to occur afterward with a given confidence or probability.

Two main types of sequential rules have been proposed. The first type is rules where the antecedent and consequent are sequential patterns [11,15,13]. The second type is rules between two unordered sets of items [6,4]. In this paper we consider the second type because it is more general and it was shown to

H. Blockeel et al. (Eds.): IDA 2014, LNCS 8819, pp. 108–119, 2014.

provide considerably higher prediction accuracy for sequence prediction in some domains [5]. Moreover, another reason is that the second type has been used in many real applications such as e-learning [6], manufacturing simulation [9], quality control [2], web page prefetching [5], anti-pattern detection in service based systems [14], embedded systems [10], alarm sequence analysis [3] and restaurant recommendation [8].

Several algorithms have been proposed for mining this type of sequential rules. CMDeo [6] is an Apriori-based algorithm that explores the search space of rules using a breadth-first search. A major drawback of CMDeo is that it can generate a huge amount of candidates. As as alternative, the CMRules algorithm was proposed. It relies on the property that any sequential rules must also be an association rule to prune the search space of sequential rules [6]. It was shown to be much faster than CMDeo for sparse datasets. Recently, the RuleGrowth [4] algorithm was proposed. It relies on a pattern-growth approach to avoid candidate generation. It was shown to be more than an order of magnitude faster than CMDeo and CMRules. However, for datasets containing dense or long sequences, the performance of RuleGrowth rapidly deterioates because it has to repeatedly perform costly database projection operations. Because mining sequential rules remains a very computationally expensive data mining task, an important research question is: "Could we design faster algorithms?"

In this paper, we address this issue by proposing the ERMiner (Equivalence class based sequential Rule Miner) algorithm. It relies on a vertical representation of the database to avoid performing database projection and the novel idea of explorating the search space of rules using equivalence classes of rules having the same antecedent or consequent. Furthermore, it includes a data structure named SCM (Sparse Count Matrix) to prune the search space.

The rest of the paper is organized as follows. Section 2 defines the problem of sequential rule mining and introduces important definitions and properties. Section 3 describes the ERMiner algorithm. Section 4 presents the experimental study. Finally, Section 5 presents the conclusion.

2 Problem Definition

Definition 1 (sequence database). Let $I = \{i_1, i_2, ..., i_l\}$ be a set of items (symbols). An *itemset* $I_x = \{i_1, i_2, ..., i_m\} \subseteq I$ is an unordered set of distinct items. The *lexicographical order* $\succ_{lex}$ is defined as any total order on I. Without loss of generality, it is assumed in the following that all itemsets are ordered according to $\succ_{lex}$. A *sequence* is an ordered list of itemsets $s = \langle I_1, I_2, ..., I_n \rangle$ such that $I_k \subseteq I$ $(1 \leq k \leq n)$. A *sequence database* SDB is a list of sequences $SDB = \langle s_1, s_2, ..., s_p \rangle$ having sequence identifiers (SIDs) $1, 2...p$.

Example 1. A sequence database is shown in Fig. 1 (left). It contains four sequences having the SIDs 1, 2, 3 and 4. Each single letter represents an item. Items between curly brackets represent an itemset. The first sequence $\langle \{a, b\}, \{c\}, \{f\}, \{g\}, \{e\} \rangle$ contains five itemsets. It indicates that items a and b occurred at the same time, were followed by c, then f and lastly e.

ID	Sequences
seq1	⟨{*a, b*},{*c*},{*f*},{*g*},{*e*}⟩
seq2	⟨{*a, d*},{*c*},{*b*},{*a, b, e, f*}⟩
seq3	⟨{*a*},{*b*},{*f*},{*e*}⟩
seq4	⟨{*b*},{*f, g, h*}⟩

ID	Rule	Support	Confidence
r1	{*a, b, c*}→{*e*}	0.5	1.0
r2	{*a*}→{*c, e, f*}	0.5	0.66
r3	{*a, b*}→{*e, f*}	0.75	1.0
r4	{*b*}→{*e, f*}	0.75	0.75
r5	{*a*}→{*e, f*}	0.75	1.0
r6	{*c*}→{*f*}	0.5	1.0
r7	{*a*}→{*b*}	0.5	0.66

Fig. 1. A sequence database (left) and some sequential rules found (right)

Definition 2 (sequential rule). A sequential rule $X \rightarrow Y$ is a relationship between two unordered itemsets $X, Y \subseteq I$ such that $X \cap Y = \emptyset$ and $X, Y \neq \emptyset$. The interpretation of a rule $X \rightarrow Y$ is that if items of X occur in a sequence, items of Y will occur afterward in the same sequence.

Definition 3 (itemset/rule occurrence). Let $s : \langle I_1, I_2 ... I_n \rangle$ be a sequence. An itemset I occurs or is contained in s (written as $I \sqsubseteq s$) iff $I \subseteq \bigcup_{i=1}^{n} I_i$. A rule $r : X \rightarrow Y$ occurs or is contained in s (written as $r \sqsubseteq s$) iff there exists an integer k such that $1 \leq k < n$, $X \subseteq \bigcup_{i=1}^{k} I_i$ and $Y \subseteq \bigcup_{i=k+1}^{n} I_i$.

Example 2. The itemset $\{a, b, f\}$ is contained in sequence $\langle \{a\}, \{b\}, \{f\}, \{e\} \rangle$. The rule $\{a, b, c\} \rightarrow \{e, f, g\}$ occurs in $\langle \{a, b\}, \{c\}, \{f\}, \{g\}, \{e\} \rangle$, whereas the rule $\{a, b, f\} \rightarrow \{c\}$ does not, because item c does not occur after f.

Definition 4 (sequential rule size). A rule $X \rightarrow Y$ is said to be of size $k * m$ if $|X| = k$ and $|Y| = m$. Furthermore, a rule of size $f * g$ is said to be larger than another rule of size $h * i$ if $f > h$ and $g \geq i$, or alternatively if $f \geq h$ and $g > i$.

Example 3. The rules $r : \{a, b, c\} \rightarrow \{e, f, g\}$ and $s : \{a\} \rightarrow \{e, f\}$ are respectively of size $3 * 3$ and $1 * 2$. Thus, r is larger than s.

Definition 5 (support). The *support* of a rule r in a sequence database SDB is defined as $sup_{SDB}(r) = |\{s | s \in SDB \wedge r \sqsubseteq s\}|/|SDB|$.

Definition 6 (confidence). The *confidence* of a rule $r : X \rightarrow Y$ in a sequence database SDB is defined as $conf_{SDB}(r) = |\{s | s \in SDB \wedge r \sqsubseteq s\}|/|\{s | s \in SDB \wedge X \sqsubseteq s\}|$.

Definition 7 (sequential rule mining). Let $minsup, minconf \in [0, 1]$ be thresholds set by the user and SDB be a sequence database. A rule r is a *frequent sequential rule* iff $sup_{SDB}(r) \geq minsup$. A rule r is a *valid sequential rule* iff it is frequent and $conf_{SDB}(r) \geq minconf$. The *problem of mining sequential rules* from a sequence database is to discover all valid sequential rules [6].

Example 4. Fig 1 (right) shows 7 valid rules found in the database illustrated in Table 1 for $minsup = 0.5$ and $minconf = 0.5$. For instance, the rule $\{a, b, c\} \rightarrow \{e\}$ has a support of $2/4 = 0.5$ and a confidence of $2/2 = 1$. Because those values are respectively no less than $minsup$ and $minconf$, the rule is deemed valid.

3 The ERMiner Algorithm

In this section, we present the ERMiner algorithm. It relies on the novel concept of equivalence classes of sequential rules, defined as follows.

Definition 8 (rule equivalence classes). For a sequence database, let $\mathcal{R}$ be the set of all frequent sequential rules and I be the set of all items. A left equivalence class $LE_{W,i}$ is the set of frequent rules $LE_{W,i} = \{W \rightarrow Y | Y \subseteq I \wedge |Y| = i\}$ such that $W \subseteq I$ and i is an integer. Similarly, a *right equivalence class* $RE_{W,i}$ is the set of frequent rules $RE_{W,i} = \{X \rightarrow W | X \subseteq I \wedge |X| = i\}$, where $W \subseteq I$, and i is an integer.

Example 5. For $minsup = 2$ and our running example, $LE_{\{c\},1} = \{\{c\} \rightarrow \{f\}, \{c\} \rightarrow \{e\}\}$, $RE_{\{e,f\},1} = \{\{a\} \rightarrow \{e,f\}, \{b\} \rightarrow \{e,f\}, \{c\} \rightarrow \{e,f\}\}$ and $RE_{\{e,f\},2} = \{\{a,b\} \rightarrow \{e,f\}, \{a,c\} \rightarrow \{e,f\}, \{b,c\} \rightarrow \{e,f\}\}$.

Two operations called left and right merges are used by ERMiner to explore the search space of frequent sequential rules. They allows to directly generate an equivalence class using a smaller equivalence class.

Definition 9 (left/right merges). Let be a left equivalence class $LE_{W,i}$ and two rules $r : W \rightarrow X$ and $s : W \rightarrow Y$ such that $r, s \in LE_{W,i}$ and $|X \cap Y| = |X - 1|$, i.e. X and Y are identical except for a single item. A *left merge* of r, s is the process of merging r, s to obtain $W \rightarrow X \cup Y$. Similarly, let be a right equivalence class $RE_{W,i}$ and two rules $r : X \rightarrow W$ and $r : Y \rightarrow W$ such that $r, s \in RE_{W,i}$ and $|X \cap Y| = |X - 1|$. A *right merge* of r, s is the process of merging r, s to obtain the rule $X \cup Y \rightarrow W$.

Property 1 (generating a left equivalence class). Let be a left equivalence class $LE_{W,i}$. $LE_{W,i+1}$ can be obtained by performing all left merges on pairs of rules from $LE_{W,i}$. **Proof.** Let be any rule $r : W \rightarrow \{a_1, a_2, ...a_{i+1}\}$ in $LE_{W,i+1}$. By Definition 8, rules $W \rightarrow \{a_1, a_2, ...a_{i-1}, a_i\}$ and $W \rightarrow \{a_1, a_2, ...a_{i-1}, a_{i+1}\}$ are members of $LE_{W,i}$, and a left merge of those rules will generate r.□

Property 2 (generating a right equivalence class). Let be a right equivalence class $RE_{W,i}$. $RE_{W,i+1}$ can be obtained by performing all right merges on pairs of rules from $RE_{W,i}$. **Proof.** The proof is similar to Property 1 and is therefore omitted.

To explore the search space of frequent sequential rules using the above merge operations, ERMiner first scans the database to build all equivalence classes for frequent rules of size $1 * 1$. Then, it recursively performs left/right merges starting from those equivalence classes to generate the other equivalence classes. To ensure that no rule is generated twice, the following ideas have been used.

First, an important observation is that a rule can be obtained by different combinations of left and right merges. For example, consider the rule $\{a, b\} \rightarrow \{c, d\}$. It can be obtained by performing left merges for $LE_{\{a\},1}$ and $LE_{\{b\},1}$

followed by right merges on $RE_{\{c,d\},1}$. But it can also be obtained by performing right merges on $RE_{\{c\},1}$ and $RE_{\{d\},1}$ followed by left merges using $LE_{\{a,b\},1}$. A simple solution to avoid this problem is to not allow performing a left merge after a right merge but to allow performing a right merge after a left merge. This solution is illustrated in Fig. 2.

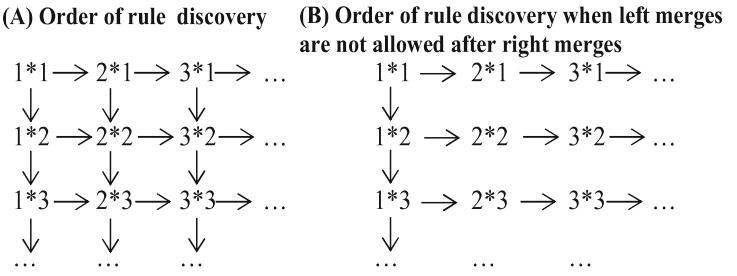

Fig. 2. The order of rule discovery by left/right merge operations

Second, another key observation is that a same rule may be obtained by merging different pairs of rules from the same equivalence class. For example, a rule $\{a, b, c\} \rightarrow \{e\}$ may be obtained by performing a left merge of $\{a, b\} \rightarrow \{e\}$ with $\{a, c\} \rightarrow \{e\}$ or with $\{b, c\} \rightarrow \{e\}$. To avoid generating the same rule twice, a simple solution is to impose a total order on items in rule antecedents (consequents) and to only perform a left merge (right merge) if the rule consequent (rule antecedent) shares all but the last item according to the total order. In the previous example, this means that $\{a, c\} \rightarrow \{e\}$ would not be merged with $\{b, c\} \rightarrow \{e\}$.

Using the above solutions, it can be easily seen that all rules are generated only once. However, to be efficient, a sequential rule mining algorithm should be able to prune the search space. This is done using the following properties for merge operations.

Property 3 (antimonotonicity with left/right merges). Let be a sequence database SDB and two frequent rules r, s. Let t be a rule obtained by a left or right merge of r, s. The support of t is lower or equal to the support of r and that of s. **Proof.** Since t contains exactly one more item than r and s, it can only appear in the same number sequences or less. □

Property 4 (pruning). If the support of a rule is less than *minsup*, then it should not be merged with any other rules because all such rules are infrequent. **Proof.** This directly follows from Property 3.

Because there does not exist any similar pruning properties for confidence, it is necessary to explore the search space of frequent rules to get the valid ones.

Fig. 1 shows the main pseudocode of ERMiner, which integrates all the previous idea. ERMiner takes as input a sequence database SDB, and the *minsup*

and $minconf$ thresholds. It first scans the database once to build all equivalence classes of rules of size $1 * 1$, i.e. containing a single item in the antecedent and a single item in the consequent. Then, to discover larger rules, left merges are performed with all left equivalence classes by calling the *leftSearch* procedure. Similarly, right merges are performed for all right equivalence classes by calling the *rightSearch* procedure. Note that the *rightSearch* procedure may generate some new left-equivalence classes because left merges are allowed after right merges. These equivalence classes are stored in a structure named $leftStore$. To process these equivalence classes, an additional loop is performed. Finally, the algorithm returns the set of rules found $rules$.

Algorithm 1. The ERMiner algorithm

input : SDB: a sequence database, $minsup$ and $minconf$: the two user-specified thresholds
output: the set of valid sequential rules

```
1  leftStore ← ∅ ;
2  rules ← ∅ ;
3  Scan SDB once to calculate EQ, the set of all equivalence classes of rules of
   size 1*1;
4  foreach left equivalence class H ∈ EQ do
5  |  leftSearch (H, rules);
6  end
7  foreach right equivalence class J ∈ EQ do
8  |  rightSearch (J, rules, leftStore);
9  end
10 foreach left equivalence class K ∈ leftStore do
11 |  rightSearch (K);
12 end
13 return rules;
```

Fig. 2 shows the pseudocode of the *leftSearch* procedure. It takes as parameter an equivalence class LE. Then, for each rule r of that equivalence class, a left merge is performed with every other rules to generate a new equivalence class. Only frequent rules are kept. Furthermore, if a rule is valid, it is output. Then, $leftSearch$ is recursively called to explore each new equivalence class generated that way. The *rightSearch* (see Fig. 3) is similar. The main difference is that new left equivalences are stored in the left store structure because their exploration is delayed, as previously explained in the main procedure of ERMiner.

Now, it is important to explain how the support and confidence of each rule is calculated by ERMiner (we had previously deliberately ommitted this explanation). Due to space limitation and because this calculation is done similarly as in the RuleGrowth [4] algorithm, we here only give the main idea. Initially, a database scan is performed to record the first and last occurrences of each item in each sequence where it appears. Thereafter, the support of each rule

Algorithm 2. The leftSearch procedure

input : LE: a left equivalence class, $rules$: the set of valid rules found until now, $minsup$ and $minconf$: the two user-specified thresholds

```
1  foreach rule r ∈ LE do
2      LE' ← ∅ ;
3      foreach rule s ∈ LE such that r ≠ s and the pair r, s have not been
       processed do
4          Let c, d be the items respectively in r, s that do not appear in s, r ;
5          if countPruning(c, d) = false then
6              t ← leftMerge(r, s) ;
7              calculateSupport(t, r, s);
8              if sup(t) ≥ minsup then
9                  calculateConfidence(t, r, s);
10                 if conf(t) ≥ minconf then
11                     rules ← rules ∪ {t};
12                 end
13                 LE' ← LE' ∪ {t};
14             end
15         end
16     end
17     leftSearch (LE', rules);
18 end
```

of size 1*1 is directly generated by comparing first and last occurrences, without scanning the database. Similarly, the first and last occurrences of each rule antecedent and consequent are updated for larger rules without scanning the database. This allows to calculate confidence and support efficiently (see [4] for more details about how this calculation can be done).

Besides, an optimization is to use a structure that we name the *Sparse Count Matrix* (SCM) (aka CMAP [7]). This structure is built during the first database scan and record in how many sequences each item appears with each other items. For example, Fig. 3 shows the structure built for the database of Fig. 1 (left), represented as a triangular matrix. Consider the second row. It indicates that item b appear with items b, c, d, e, f, g and h respectively in 2, 1, 3, 4, 2 and 1 sequences. The SCM structure is used for pruning the search space as follows (implemented as the *countPruning* function in Fig. 3 and 2). Let be a pair of rules r, s that is considered for a left or right merge and c, d be the items of r, s, that respectively do not appear in s, r. If the count of r, s is less than $minsup$ in the SCM, then the merge does not need to be performed and the support of the rule is not calculated.

Lastly, another important optimization is how to implement the left store structure for efficiently storing left equivalence classes of rules that are generated by right merges. In our implementation, we use a hashmap of hashmaps, where the first hash function is applied to the size of a rule and the second hash function

Algorithm 3. The rightSearch procedure

input : RE: a right equivalence class, $rules$: the set of valid rules found until now, $minsup$ and $minconf$: the two user-specified thresholds, $leftStore$: the structure to store left-equivalence classes of rules generated by right-merges

```
1  foreach rule r ∈ RE do
2      RE' ← ∅ ;
3      foreach rule s ∈ RE such that r ≠ s and the pair r, s have not been
       processed do
4          Let c, d be the items respectively in r, s that do not appear in s, r ;
5          if countPruning(c, d) = false then
6              t ← rightMerge(r, s) ;
7              calculateSupport(t, r, s);
8              if sup(t) ≥ minsup then
9                  calculateConfidence(t, r, s);
10                 if conf(t) ≥ minconf then
11                     rules ← rules ∪ {t};
12                 end
13                 RE' ← RE' ∪ {t};
14                 addToLeftStore(t)
15             end
16         end
17     end
18     rightSearch (RE', rules);
19 end
```

Item	a	b	c	d	e	f
b	3					
c	2	2				
d	1	1	1			
e	3	3	2	1		
f	3	4	2	1	3	
g	1	2	1	0	1	2
h	0	1	0	0	0	1

Fig. 3. The Sparse Count Matrix

is applied to the left itemset of the rule. This allows to quickly find to which equivalence class belongs a rule generated by a right merge.

4 Experimental Evaluation

We performed experiments to assess the performance of the proposed algorithm. Experiments were performed on a computer with a third generation Core i5 processor running Windows 7 and 5 GB of free RAM. We compared the performance

of ERMiner with the state-of-the-art algorithms for sequential rule mining RuleGrowth [4]. All algorithms were implemented in Java.

All memory measurements were done using the Java API. Experiments were carried on five real-life datasets having varied characteristics and representing four different types of data (web click stream, sign language utterances and protein sequences). Those datasets are *Sign*, *Snake*, *FIFA*, *BMS* and *Kosarak10k*. Table 2 4 summarizes their characteristics. The source code of all algorithms and datasets used in our experiments can be downloaded from `http://goo.gl/aAegWH`.

Table 1. Dataset characteristics

dataset	sequence count	distinct item count	avg. seq. length (items)	type of data
Sign	730	267	51.99 (std = 12.3)	language utterances
Snake	163	20	60 (std = 0.59)	protein sequences
FIFA	20450	2990	34.74 (std = 24.08)	web click stream
BMS	59601	497	2.51 (std = 4.85)	web click stream
Kosarak10k	10000	10094	8.14 (std = 22)	web click stream

We ran all the algorithms on each dataset while decreasing the *minsup* threshold until algorithms became too long to execute, ran out of memory or a clear winner was observed. For these experiments, we fixed the *minconf* threshold to 0.75. However, note that results are similar for other values of the *minconf* parameter since the confidence is not used to prune the search space by the compared algorithms. For each dataset, we recorded the execution time, the percentage of candidate pruned by the SCM structure and the total size of SCMs.

Execution times. The comparison of execution times is shown in Fig. 4. It can be seen that ERMiner is faster than RuleGrowth on all datasets and that the performance gap increases for lower *minsup* values. ERMiner is up to about five times faster than RuleGrowth. This is because RuleGrowth has to perform costly database projection operations.

Memory overhead of using SCM. We have measured the overhead produced by using the SCM structure by ERMiner. The size of SCM is generally quite small (less than 35 MB). The reason is that we have implemented it as a sparse matrix (a hashmap of hashmaps) rather than a full matrix (a $n \times n$ array for n items). If a full matrix is used the size of SCM increased up to about 300 MB.

Overall memory usage. The maximum memory usage of RuleGrowth / ERMiner for the Snake, FIFA, Sign, BMS and Kosarak datasets were respectively 300 MB / 1950 MB, 478 MB / 2030 MB, 347 MB / 1881 MB, 1328 MB / 2193 MB and 669 MB / 1441 MB. We therefore notice that thhere is a trade-off between having faster execution times with ERMiner versus having lower memory consumption with RuleGrowth. The higher memory consumption for ERMiner is in great part due to the usage of the left store structure which requires maintaining several equivalence classes into memory at the same time.

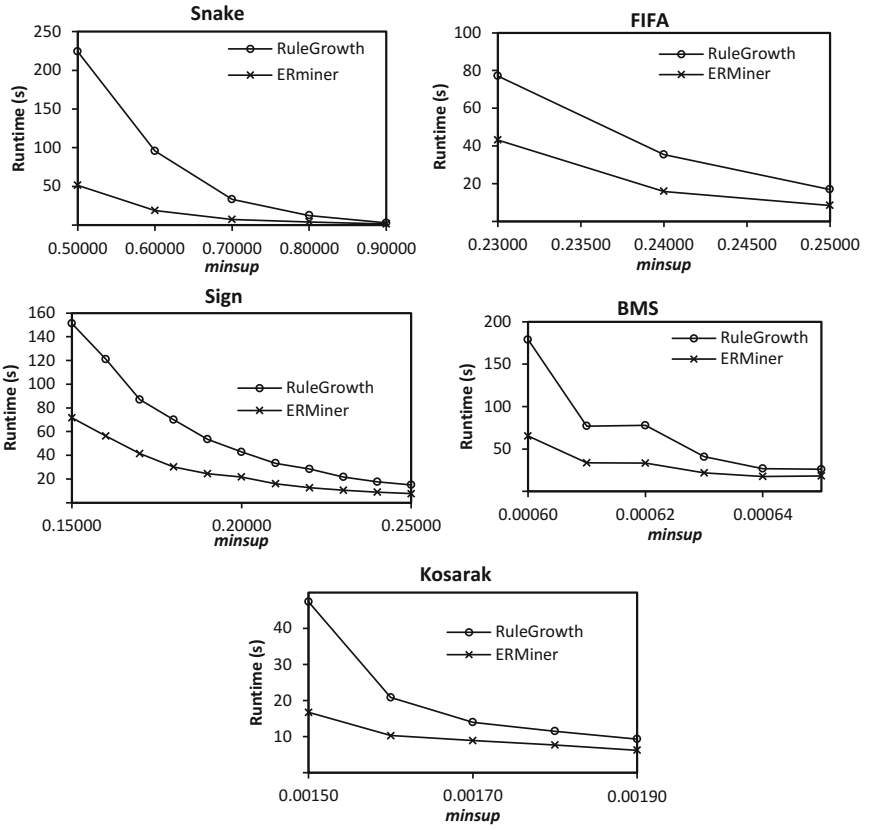

Fig. 4. Execution times

Effectiveness of candidate pruning. The percentage of candidate rules pruned by using the SCM data structure in ERMiner for the Snake, FIFA, Sign, BMS and Kosarak datasets were respectively 1 %, 0.2 %, 3.9 %, 3 % and 51 %. It can be concluded that pruning based on SCM is less effective for datasets containing dense or long sequences(e.g. Snake, FIFA, Sign) where each item co-occurs many times with every other items. It could therefore be desactivated on such datasets.

5 Conclusion

In this paper, we proposed a new sequential rule mining algorithm named ERMiner (Equivalence class based sequential Rule Miner). It relies on the novel idea of searching using equivalence classes of rules having the same antecedent or consequent. Furthermore, it an includes a data structure named SCM (Sparse Count Matrix) to prune the search space. An extensive experimental study with five real-life datasets shows that ERMiner is up to five times faster than the state-of-the-art algorithm but comsumes more memory. It can therefore be seen as

an interesting trade-off when speed is more important than memory. The source code of all algorithms and datasets used in our experiments can be downloaded from `http://goo.gl/aAegWH`.

Acknowledgement. This work is financed by a National Science and Engineering Research Council (NSERC) of Canada research grant.

References

1. Agrawal, R., Ramakrishnan, S.: Mining sequential patterns. In: Proc. 11th Intern. Conf. Data Engineering, pp. 3–14. IEEE (1995)
2. Bogon, T., Timm, I.J., Lattner, A.D., Paraskevopoulos, D., Jessen, U., Schmitz, M., Wenzel, S., Spieckermann, S.: Towards Assisted Input and Output Data Analysis in Manufacturing Simulation: The EDASIM Approach. In: Proc. 2012 Winter Simulation Conference, pp. 257–269 (2012)
3. Bogon, T., Timm, I.J., Lattner, A.D., Paraskevopoulos, D., Jessen, U., Schmitz, M., Wenzel, S., Spieckermann, S.: Towards Assisted Input and Output Data Analysis in Manufacturing Simulation: The EDASIM Approach. In: Proc. 2012 Winter Simulation Conference, pp. 257–269 (2012)
4. Fournier-Viger, P., Nkambou, R., Tseng, V.S.: RuleGrowth: Mining Sequential Rules Common to Several Sequences by Pattern-Growth. In: Proc. ACM 26th Symposium on Applied Computing, pp. 954–959 (2011)
5. Fournier-Viger, P., Gueniche, T., Tseng, V.S.: Using Partially-Ordered Sequential Rules to Generate More Accurate Sequence Prediction. In: Zhou, S., Zhang, S., Karypis, G. (eds.) ADMA 2012. LNCS, vol. 7713, pp. 431–442. Springer, Heidelberg (2012)
6. Fournier-Viger, P., Faghihi, U., Nkambou, R., Mephu Nguifo, E.: CMRules: Mining Sequential Rules Common to Several Sequences. Knowledge-based Systems 25(1), 63–76 (2012)
7. Fournier-Viger, P., Gomariz, A., Campos, M., Thomas, R.: Fast Vertical Mining of Sequential Patterns Using Co-occurrence Information. In: Tseng, V.S., Ho, T.B., Zhou, Z.-H., Chen, A.L.P., Kao, H.-Y. (eds.) PAKDD 2014, Part I. LNCS, vol. 8443, pp. 40–52. Springer, Heidelberg (2014)
8. Han, M., Wang, Z., Yuan, J.: Mining Constraint Based Sequential Patterns and Rules on Restaurant Recommendation System. Journal of Computational Information Systems 9(10), 3901–3908 (2013)
9. Kamsu-Foguem, B., Rigal, F., Mauget, F.: Mining association rules for the quality improvement of the production process. Expert Systems and Applications 40(4), 1034–1045 (2012)
10. Leneve, O., Berges, M., Noh, H.Y.: Exploring Sequential and Association Rule Mining for Pattern-based Energy Demand Characterization. In: Proc. 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings, pp. 1–2. ACM (2013)
11. Lo, D., Khoo, S.-C., Wong, L.: Non-redundant sequential rules - Theory and algorithm. Information Systems 34(4-5), 438–453 (2009)
12. Mabroukeh, N.R., Ezeife, C.I.: A taxonomy of sequential pattern mining algorithms. ACM Computing Surveys 43(1), 1–41 (2010)

13. Pham, T.T., Luo, J., Hong, T.P., Vo, B.: An efficient method for mining non-redundant sequential rules using attributed prefix-trees. Engineering Applications of Artificial Intelligence 32, 88–99 (2014)
14. Nayrolles, M., Moha, N., Valtchev, P.: Improving SOA antipatterns detection in Service Based Systems by mining execution traces. In: Proc. 20th IEEE Working Conference on Reverse Engineering, pp. 321–330 (2013)
15. Zhao, Y., Zhang, H., Cao, L., Zhang, C., Bohlscheid, H.: Mining both positive and negative impact-oriented sequential rules from transactional data. In: Theeramunkong, T., Kijsirikul, B., Cercone, N., Ho, T.-B. (eds.) PAKDD 2009. LNCS, vol. 5476, pp. 656–663. Springer, Heidelberg (2009)