

Negative Generator Border for Effective Pattern Maintenance

Mengling Feng,¹ Jinyan Li,¹ Limsoon Wong,²Yap-Peng Tan¹

¹Nanyang Technological University & ²National University of Singapore

¹{feng0010, jyli, eyptan}@ntu.edu.sg & ²wongls@comp.nus.edu.sg

Abstract. In this paper, we study the maintenance of frequent patterns in the context of the generator representation. The generator representation is a concise and lossless representation of frequent patterns. We effectively maintain the generator representation by systematically expanding its *Negative Generator Border*. According to our literature review, no prior work has studied the maintenance of the generator representation. To illustrate the proposed maintenance idea, a new algorithm is developed to maintain the generator representation for support threshold adjustment. Our experimental results show that the proposed algorithm is significantly faster than other state-of-the-art algorithms. This proposed maintenance idea can also be extended to other representations of frequent patterns as demonstrated in this paper.

1 Introduction

Frequent patterns, also called frequent itemsets, refer to patterns that appear frequently in a particular dataset [1]. The discovery of frequent patterns can be formally defined as follows. Let $\mathcal{I} = \{i_1, i_2, \dots, i_m\}$ be a set of distinct literals called “items”, and also let $\mathcal{D} = \{t_1, t_2, \dots, t_n\}$ be a transactional “dataset”, where t_i ($i \in [1, n]$) is a “transaction” that contains a non-empty set of items. Each subset of \mathcal{I} is called a “pattern” or an “itemset”. The “support” of a pattern P in a dataset \mathcal{D} is defined as $sup(P, \mathcal{D}) = |\{t | t \in \mathcal{D} \wedge P \subseteq t\}|$. A pattern P is said to be *frequent* in a dataset \mathcal{D} if $sup(P, \mathcal{D})$ is greater than or equal to a pre-specified support threshold ms . The support threshold, ms , can also be defined in terms of percentage, in which a pattern P is said to be *frequent* in a dataset \mathcal{D} if $sup(P, \mathcal{D}) \geq ms \times |\mathcal{D}|$. The collection of all frequent patterns in \mathcal{D} is called the “space of frequent patterns” and is denoted as $\mathcal{F}(\mathcal{D}, ms)$. The task of frequent pattern discovery is to find all the patterns in $\mathcal{F}(\mathcal{D}, ms)$. Figure 1 shows an example of transactional dataset and the corresponding frequent pattern space when $ms = 1$.

Datasets are dynamic in nature. From time to time, new transactions/items may be inserted; old and invalid transactions/items may be removed; and the support threshold may be adjusted to obtain the desirable sets of frequent patterns. Repeating the discovery process every time the dataset is updated is a naive and definitely inefficient solution. Thus, there is a strong demand for effective algorithms to maintain frequent patterns for data updates and support threshold adjustment.

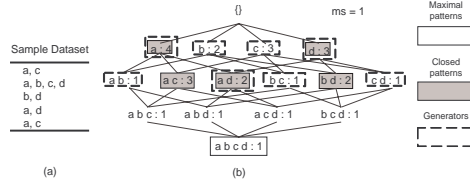


Fig. 1. (a) An example of transactional dataset. (b) The pattern space of the sample dataset in (a), and the concise representations of the pattern space.

Most of the current maintenance algorithms can be grouped into two major categories: *Apriori*-based and sliding window filtering (*SWF*). Both the *Apriori*-based and *SWF* approaches are developed based on the candidate-enumeration-and-elimination framework. *Apriori*-based algorithms [2, 6] enumerate new candidates iteratively based on the *a priori* property. *SWF* algorithms [5, 11] slice a dataset into several partitions and then employ a filtering threshold in each partition to generate candidate patterns. The *Apriori*-based algorithms and *SWF* algorithms aim to update and maintain the entire frequent pattern space. However, the undesirable large number of frequent patterns greatly limits their performance. To break the bottleneck, concise representations of frequent patterns, more importantly, efficient maintenance of the concise representations are highly desired. In this paper, we focus our investigation on the maintenance of the generator representation [14] — a concise and lossless¹ representation of frequent patterns.

In the literature, algorithms have been proposed to maintain two types of concise representations under some unfavorable restrictions. **Moment** [7] is one example. **Moment** dynamically maintains the frequent closed patterns [14]. However, **Moment** is proposed on the hypothesis that there are only *small changes* to the frequent closed patterns given a small amount of updates. Due to this strict constraint, the performance of **Moment** degrades dramatically when the amount of updates gets large. **ZIGZAG** [15] is another example, which effectively maintains the maximal patterns [3]. **ZIGZAG** updates the maximal patterns by a backtracking search, which is guided by the outcomes of the previous maintenance iteration. Although the maximal patterns can concisely represent frequent patterns, they do not provide support information for other frequent patterns. That is, the maximal patterns are a lossy representation. In this work, unlike **ZIGZAG** for the maximal patterns, our maintenance algorithm is for a lossless representation; unlike **Moment** which bears some unfavorable assumptions, our maintenance algorithm aims to handle wide range of changes efficiently.

We propose to maintain the generator representation by expanding the *Negative Generator Border*. The expansion of the negative generator border is guided by a systematic technique, which ensures the expansion is complete and yet

¹ We say a representation is lossless if it is sufficient to derive and determine the support of all frequent patterns without accessing the datasets.

involves no redundancy. To better illustrate the idea, we focus on the update scenario where the support threshold is adjusted. A novel algorithm — Support Threshold Update Maintainer (STUM) — is proposed. Although support thresholds can be defined in terms of either counts or percentages, (STUM) applies to both definitions. We further show that the proposed maintenance idea can be extended to other concise representations that share common characteristics with the generator representation.

2 Generators and Negative Generator Border: A Concise Representation

The concept of **generator**, also known as the **key pattern**, is first introduced in [14]. The generators, together with the close patterns and maximal patterns, are commonly used concise representations of the frequent pattern space. Figure 1 (b) demonstrates how these representations are applied to the frequent pattern space of the sample dataset in Figure 1 (a). Other types of frequent pattern representations are also available such as the free-sets [4] and the disjunctive-free sets [10]. But the support inference by these representations is very complicated. Details of these representations can be found in the Appendix (<http://www.ntu.edu.sg/home5/feng0010/appendix.pdf>).

To effectively maintain the frequent patterns, we propose to represent the space of frequent patterns with both the frequent generators and the negative generator border. For ease of discussion, this representation is referred as *the generator representation* for the rest of the paper.

Definition 1 (Generator). *Given a dataset \mathcal{D} , a pattern P is a “generator” iff for every $P' \subset P$, it is the case that $\text{sup}(P', \mathcal{D}) > \text{sup}(P, \mathcal{D})$.*

For a dataset \mathcal{D} and support threshold ms , the set of frequent generators, $\mathcal{FG}(\mathcal{D}, ms)$, includes all generators that are frequent. On the other hand, the **negative generator border**, $NBd(\mathcal{FG}(\mathcal{D}, ms))$, refers to the set of the minimal infrequent generators, and it is equivalent to the set of 0-free-sets [4].

Definition 2 (Negative Generator Border). *Given a dataset \mathcal{D} and support threshold ms , $NBd(\mathcal{FG}(\mathcal{D}, ms)) = \{G | G \notin \mathcal{FG}(\mathcal{D}, ms) \wedge (\forall G' \subset G, G' \in \mathcal{FG}(\mathcal{D}, ms))\}$.*

Generators in the negative generator border are named **negative border generators**. For a dataset \mathcal{D} and support threshold ms , the generator representation includes: the set of frequent generators, $\mathcal{FG}(\mathcal{D}, ms)$, the negative generator border, $NBd(\mathcal{FG}(\mathcal{D}, ms))$, and their corresponding support values. Following Definition 1,2 and the *a priori* property of frequent patterns, we have the following corollary.

Corollary 1. *Given a dataset \mathcal{D} and support threshold ms , (1) a pattern P is infrequent iff $\exists G | P \supseteq G \wedge G \in NBd(\mathcal{FG}(\mathcal{D}, ms))$; (2) a pattern P is frequent iff $\nexists G | P \supseteq G \wedge G \in NBd(\mathcal{FG}(\mathcal{D}, ms))$; and (3) for any frequent pattern P , $\text{sup}(P, \mathcal{D}) = \min\{\text{sup}(G, \mathcal{D}) | G \subseteq P, G \in \mathcal{FG}(\mathcal{D}, ms)\}$.*

Corollary 1 implies that the generator representation is sufficient to determine all frequent patterns and their support values. Therefore, the generator representation is a lossless concise representation. We also observe that generators follow the *a priori* property as stated in FACT 3. When datasets are updated, the *a priori* characteristic of generators allows us to effectively enumerate newly emerged generators based on the negative generator border. The negative generator border acts conveniently as a start point for us to resume the pattern enumerations.

Fact 3 (Cf. [12]) *Let P be a pattern in \mathcal{D} . If P is a generator, then every subset of P is also a generator in \mathcal{D} . Furthermore, if P is a frequent generator, then every subset of P is also a frequent generator in \mathcal{D} .*

Another advantage of the generator representation is that it can derive maximal patterns and closed patterns easily. One can derive frequent closed patterns from the frequent generators with the “closure” operation [14]. For a particular generator G , the “closure” operation is to find the maximal pattern C such that C and G always appear together in the dataset. Li. et al [12] have proposed some efficient techniques to conduct the “closure” operation. For frequent maximal patterns, they are basically the longest frequent closed patterns. Therefore, following the similar procedure as the derivation of closed patterns, one can also derive frequent maximal patterns easily from the frequent generators.

By concisely representing the frequent patterns using the generator representation, we greatly reduce the number of involved patterns and thus the complexity of the frequent pattern maintenance problem. Instead of maintaining the large number of frequent patterns, we only need to maintain the generators and the negative generator border. Moreover, the *a priori* characteristic of generators allows us to generate new generators and update the negative generator border effectively by expanding the exiting border.

3 Negative Generator Border in Pattern Maintenance

We investigate in this section how the concept of negative generator border can be employed to facilitate the maintenance of the generator representation. In this paper, we focus on the update scenario where the support threshold is adjusted. We also introduce systematic enumeration techniques to ensure the maintenance process with the negative generator border is complete and efficient. It is also discovered that the proposed maintenance idea can be generalized to other more complicated representations of frequent patterns, e.g. the free-sets [4] and disjunctive-free sets [10].

3.1 Support threshold adjustment maintenance

Setting the right support threshold is crucial in frequent pattern mining. Inadequate support threshold may produce too few patterns to be meaningful or too

many to be processed. It is unlikely to set the appropriate threshold at the first time. Thus the support threshold is often adjusted to obtain desirable knowledge. Moreover, in the case where the support threshold ms is defined in terms of percentage, data updates, such as transaction insertion and deletion, also induce changes in the absolute support threshold. Transaction insertions cause increases in the data size $|\mathcal{D}|$ and thus increases in the absolute support threshold, which is calculated as $ms \times |\mathcal{D}|$. Likewise, transaction deletions lead to decreases in the absolute support threshold.

When the support threshold increases, some existing frequent generators become infrequent, and the frequent pattern space shrinks. The generator representation of frequent patterns can be maintained by first removing existing generators that are no longer frequent. The negative generator border is then reconstructed with the minimal patterns among the newly infrequent generators and the original negative border generators. The maintenance process is quite straightforward and can be efficiently accomplished with the developed systematic enumeration method. Details will be discussed later.

When the threshold decreases, new frequent generators may emerge, and the frequent pattern space expands. In this case, the maintenance problem becomes more challenging, as little is known about the newly emerged generators. We resolve this challenge efficiently based on the concept of negative generator border.

Negative generator border is defined based on the the idea of *negative border*. The notion of negative border is first introduced in [13]. The negative border of frequent patterns refers to the set of minimal infrequent patterns. Maintenance algorithm **Border** [2] is proposed based on the idea of negative border. In **Border**, newly emerged frequent patterns are enumerated level-by-level from the negative border. However, **Border** aims to maintain the whole set of frequent patterns and thus suffers from the tremendous size of frequent patterns.

On the other hand, the negative generator border, as formally defined in Definition 2, refers to the set of minimal infrequent generators. The negative generator border records the nodes, where the previous enumeration of generator stops, as shown in Figure 2 (b). It thus serves as a convenient starting point for further enumeration of newly emerged generators when the support threshold decreases. This allows us to utilize previously obtained information to avoid redundant generation of existing generator and enumeration of unnecessary candidates.

Proposition 1. *Given a dataset \mathcal{D} and a support threshold ms , let $\mathcal{FG}(\mathcal{D}, ms)$ denote the set of frequent generators and $NBd(\mathcal{FG}(\mathcal{D}, ms))$ be the corresponding negative generator border. Suppose the support threshold is adjusted to ms_{upd} , where $ms_{upd} < ms$. For every newly emerged generator G ($G \notin \mathcal{FG}(\mathcal{D}, ms) \wedge G \in \mathcal{FG}(\mathcal{D}, ms_{upd})$), there exists $G' \in NBd(\mathcal{FG}(\mathcal{D}, ms))$ and $G'' \in NBd(\mathcal{FG}(\mathcal{D}, ms_{upd}))$ such that $G' \subseteq G \subseteq G''$.*

Proof. The proposition can be proven easily based on the a priori property of generators, and thus it is not included here.

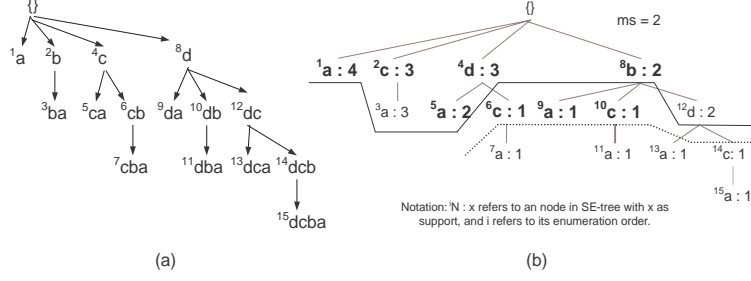


Fig. 2. (a) A set-enumeration tree of items $\{a, b, c, d\}$ with ordering $d <_0 c <_0 b <_0 a$. (b) The set-enumeration tree for the sample dataset with support threshold $ms = 2$; the solid line separates the frequent patterns from the infrequent ones; the patterns in bold form the generator representation; and the generators between the solid and the dotted lines form the negative generator border.

Proposition 1 shows that, when the support threshold decreases, every newly emerged generator falls in between the original and the updated negative generator border. This implies that all newly emerged generators can be generated without extra overhead, as we enumerate the updated negative generator border from the original border. This further simplifies the maintenance task to the update of the negative generator border.

We update the negative generator border based on the candidate-enumeration-elimination framework. Candidates of new frequent generators and border generators are enumerated iteratively based on the *a priori* characteristic of generators. Thus the next question is: how we can efficiently enumerate candidates?

Systematic pattern enumeration method is the answer to this question. In this paper, we employ the “Set-enumeration Tree”, a conceptual data structure, to facilitate the candidate enumeration for the update of the negative generator border.

Let the set $I = \{i_1, \dots, i_m\}$ of items be ordered according to an arbitrary ordering $<_0$ so that $i_1 <_0 i_2 <_0 \dots <_0 i_m$. For itemsets $X, Y \subseteq I$, we write $X <_0 Y$ iff X is lexicographically “before” Y according to the order $<_0$. We say an itemset X is a “prefix” of an itemset Y iff $X \subseteq Y$ and $X <_0 Y$. We write $last(X)$ for the item $\alpha \in X$, if the items in X are $\alpha_1 <_0 \alpha_2 <_0 \dots <_0 \alpha$. We say an itemset X is the “precedent” of an itemset Y iff $X = Y - last(Y)$.

A set-enumeration tree (*SE-Tree*) is a conceptual organization on the subsets of I so that $\{\}$ is its root node; for each node X such that Y_1, \dots, Y_k are all its children from right to left, then $Y_1 <_0 \dots <_0 Y_k$; for each node X in the set-enumeration tree such that X_1, \dots, X_k are siblings to its left, we make $X \cup X_1, \dots, X \cup X_k$ the children of X ; $|X \cup X_i| = |X| + 1 = |X_i| + 1$; and $|X| = |X_i| = |X \cap X_i| + 1$.

We also induce an enumeration ordering on the nodes of this *SE-Tree* so that given two nodes X and Y , we say $X <_1 Y$ iff X would be visited before Y when we visit the set-enumeration tree in a left-to-right top-down manner. Since this visit order is a bit unusual, we illustrate it in Figure 2 (a). Here, the number besides the node indicates the time at which the node is visited. Note that; although *SE-tree* is defined with an arbitrary item order $<_0$, to reduce the number of nodes to be generated and visited in the *SE-tree* and thus the time complexity of the enumeration, we, as shown in Figure 2 (b), organize items in ascending frequency order.

When the support threshold decreases, the *SE-tree* effectively ensures that the enumeration of the new frequent generators and negative border generators is complete and non-redundant. Another advantage of *SE-tree* is that: for every pattern P , all its subsets are enumerated before it. This allows us to judge whether P is a generator at the same time as we enumerate P . Take pattern $\{a, c, d\}$ in Figure 2 (b) as an example. As shown, the subsets of $\{a, c, d\}$, including $\{a\}$, $\{c\}$, $\{d\}$, $\{a, c\}$, $\{a, d\}$ and $\{c, d\}$, are enumerated before $\{a, c, d\}$. When $\{a, c, d\}$ is enumerated, we can decide immediately that it is not a generator, for one of its subset, $\{a, c\}$, is not a generator.

In addition, the *SE-tree* can also serve as an efficient storage structure for the generator representation, as shown in Figure 2 (b). When the support threshold increases, the *SE-tree* greatly facilitates the scanning through of the existing frequent generators and the update of the negative generator border. Take Figure 2 (b) as an example again. Suppose only the frequent generators and the negative border generators are stored and ms is increased to 3. According to the enumeration order, we first check through generators $\{a\}$, $\{c\}$ and $\{d\}$, and we find that all of them remain frequent. We then check generator $\{a, d\}$. We find that $\{a, d\}$ is no longer frequent but it is a minimal infrequent generator (all subsets of $\{a, d\}$ are frequent). Thus $\{a, d\}$ is removed from the set of frequent generators and included in the updated negative generator border. Applying the similar logic, the entire generator representation can be effectively updated. With the *SE-tree*, the maintenance of the generator representation for support threshold raise is quite straightforward, and thus it is omitted in the subsequent discussions.

Combining the above findings, a novel algorithm is proposed to maintain the generator representation of frequent patterns for support threshold adjustment. The proposed algorithm is named as the ‘‘Support Threshold Update Maintainer’’ (STUM), and it is discussed in Section 4.

3.2 Generalization & extension

It is discovered that the proposed maintenance method can be generalized to other types of frequent pattern representations, as far as the representation follows the following two characteristics:

- the representation is composed with both the frequent representation pattern (e.q. frequent generator) and its corresponding negative border (e.q. negative generator border).

Algorithm 1 Proposed algorithm STUM

Input: $\mathcal{N} = \{G_1, G_2, \dots, G_m\}$, the negative generator border, where $G_1 >_0 G_2 >_0 \dots >_0 G_m$; \mathcal{FG} , the existing frequent generators; and ms' , the new support threshold.
Output: \mathcal{FG}' the updated frequent generators; and \mathcal{N}' the updated negative generator border.
Method:
1: $\mathcal{FG}' := \mathcal{FG}$; {Initialization.}
2: **for all** $G_i \in \mathcal{N}$ **do**
3: ExpandNBGenerator(G_i, ms');
 {Expand from the negative border generators.}
4: **end for**
5: **return** \mathcal{FG}' and \mathcal{N}' ;

Procedure 2 ExpandNBGenerator

Input: G , a negative border generator or a newly emerged frequent generator; ms' , the new support threshold.
Output: \mathcal{FG}' the updated frequent generators; and \mathcal{N}' the updated negative generator border.
Method:
1: **if** $sup(G) \geq ms'$ **then**
2: $G \rightarrow \mathcal{FG}'$ {Newly emerged generator}
 {Enumerate new generators from G }
3: **for all** $i >_0 last(G)$ **do**
4: $G' := G \cup i$;
5: **if** G' is a generator **then**
6: ExpandNBGenerator(G', ms')
7: **end if**
8: **end for**
9: **else**
10: $G \rightarrow \mathcal{N}'$ {Update negative generator border.}
11: **end if**
12: **return** \mathcal{FG}' and \mathcal{N}' ;

– the representation pattern follows the *a priori* property.

The free-sets [4] and the disjunctive-free sets [10] are two representations that follow the above characteristics. The detailed definitions of these two representations are included in the Appendix (<http://www.ntu.edu.sg/home5/feng0010/appendix.pdf>).

4 Support Threshold Update Maintainer (STUM)

The proposed maintenance algorithm, STUM, is presented in Algorithm 1 and Procedure 2. When the support threshold decreases, some negative border generators emerge to be frequent. We treat these border generators as starting points. The basic idea of STUM is to expand the frequent pattern space from these starting points. For a particular negative border generator G , the expand process is to enumerate new frequent generators from G . The enumeration follows the enumeration order of *SE-tree*, which ensures to be complete and efficient.

4.1 Complexity analysis

According to Algorithm 1 and Procedure 2, the time complexity of STUM is proportional to the number of candidates enumerated during the generation of

Table 1. Approximate number of enumerated candidates by STUM and Border when the support threshold is adjusted to half of the original one. Here ms denotes the original support threshold.

	accidents $ms = 50\%$	gazelle $ms = 0.5\%$	mushroom $ms = 0.5\%$	T1014D100K $ms = 1\%$	BMS-POS $ms = 0.5\%$	pumsb_star $ms = 20\%$
STUM	5K	58	27K	173	8K	100K
Border	36K	3K	533K	16K	30K	122K

newly emerged frequent generators. Thus the complexity of the proposed algorithm can be modelled as $O(N_{GenCan})$, where N_{GenCan} refers to the number of enumerated generator candidates. According to the complexity study, we foresee that STUM is more efficient than some of the previous algorithms, such as Border [2]. The computational complexity of Border is $O(N_{FreqCan})$, where $N_{FreqCan}$ refers to the number of candidates enumerated during the generation of newly emerged frequent patterns. In general, $N_{GenCan} \ll N_{FreqCan}$, as shown in Table 1.

4.2 Implementation

As shown in Procedure 2, for every enumerated generator candidates, we need to retrieve its support value. To avoid multiple scans of datasets, we employ a prefix-tree and a header table to summarize the dataset. Figure 3 (a) demonstrates how the sample dataset is compressed and stored in a prefix tree. (Details on the construction of prefix tree can be found in [9].) With the prefix tree and the header table, support values of patterns can be retrieved without data scanning. Let us take the sample dataset in Figure 3 as an example. Suppose we need to obtain the support of pattern $\{a, b\}$. We first need to look for all the paths that contain item b based on the linked list pointers in the header table. Then, for each path that contains b , we travel up and search for item a . In this case, only one path contains both items b and a , and the support of the path is 1. Therefore, we have $sup(\{a, b\}, \mathcal{D}) = 1$.

The prefix tree structure also facilitates effective candidate pruning. According to Procedure 2, the generator candidates are produced based on the enumeration order of *SE-tree*. Given a generator G , only items $i >_0 last(G)$ are enumerated. This greatly reduces the number of unnecessary enumerations. On top of that, with the concept of local prefix tree, we can completely avoid generating unnecessary candidates. For example, Figure 3 (b) shows the local prefix tree for generator $\{d\}$. Suppose $ms = 2$. Based on the local prefix tree, we know immediately that the enumeration of candidate $\{c, d\}$ is not necessary, for its support is below ms .

5 Experimental Evaluation

The computational effectiveness of the proposed algorithm, STUM, is tested on several benchmark datasets from the *FIMI* Repository [8]. STUM is evaluated with various degrees of support threshold adjustment.

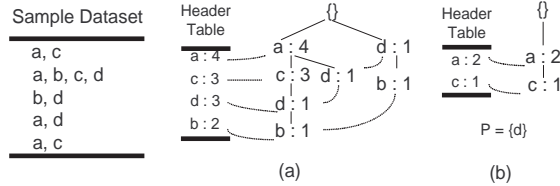


Fig. 3. (a) Global prefix tree and header table for the sample dataset with ordering $b <_0 d <_0 c <_0 a$; and (b) local prefix tree and header table for pattern $\{d\}$.

STUM is compared with some state-of-the-art frequent pattern discovery and maintenance methods, including GC-growth [12], ZIGZAG [15] and Border [2]. GC-growth is an effective algorithm that generates frequent generators. ZIGZAG is one of the recently proposed frequent maximal pattern maintenance algorithm. Border is a frequent pattern maintenance algorithm proposed based on the concept of negative border. The original implementation of Border requires multiple data scans. This induces heavy I/O overhead. To better justify the effectiveness of the proposed method, we improved the implementation of Border. We employ a prefix-tree structure in Border to summarize the dataset and thus to avoid multiple data scans. We name the improved implementation of Border as Border(prefixTree). All the experiments are run on a PC with 2.8 GHz processor and 2 GB RAM.

Figure 4 compares the computational time of STUM against the one of other methods. We observe that, in general, STUM outperforms the rest considerably. However, it is also observed that, compared to the frequent generator discovery algorithm GC-growth, the advantage of STUM drops as the change of support threshold gets larger. This is because large variation in support threshold logically leads to dramatic changes in the frequent pattern space. Thus it is more expensive to update, and the advantage of STUM is found to diminish when the change of support threshold gets larger. It is inevitable that when the support threshold is adjusted to a certain extent, the change induced to the pattern space becomes so significant that it becomes more efficient to re-discover the patterns than to maintain and update them.

We also measure the "speed-up" achieved by STUM against other methods. The speed-up is calculated as the ratio between the computational time of the comparing method and that of the proposed method. Table 2 summarizes the average speed-up we have achieved on various datasets. Since Border suffers from heavy I/O overhead, STUM outperforms Border significantly. It can also be seen that, by employing the prefix-tree structure, the improved implementation of Border is much faster than the original implementation. STUM performs the best on dataset *T10I4D100K*. It is faster than the other methods by at least an order of magnitude.

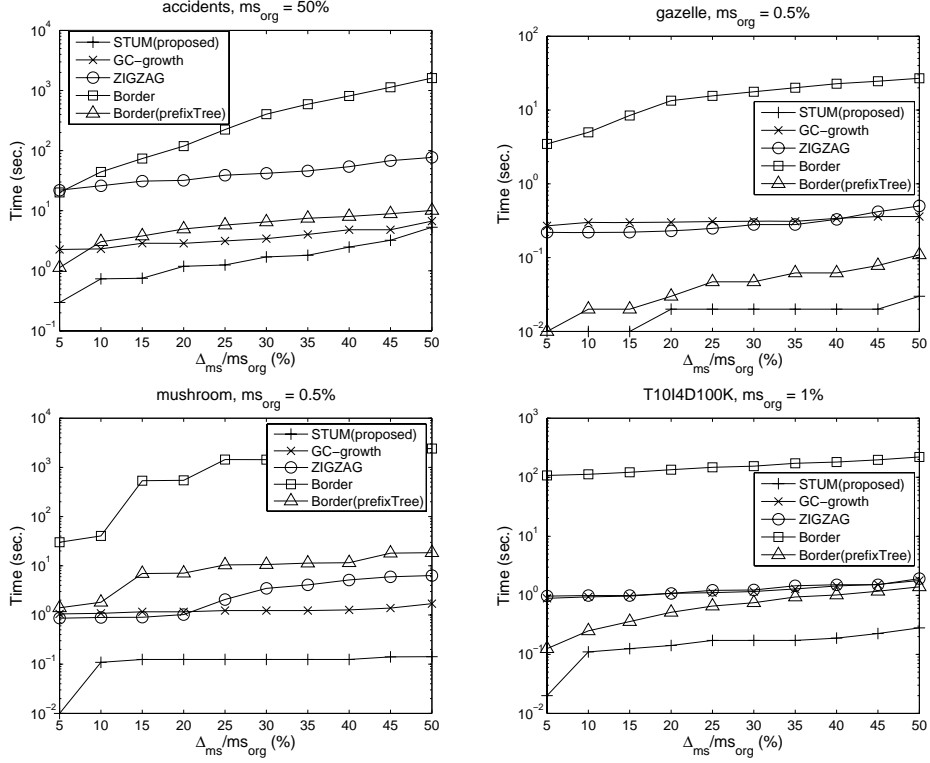


Fig. 4. Comparison of computation time of the proposed algorithm — STUM, GC-growth, ZIGZAG, Border and Border(prefixTree).

6 Closing remarks

In this paper, we investigated the maintenance of the generator representation of frequent patterns. We studied the characteristics of the generator representation and found that generators follow the *a priori* property. Using this property, we proposed to maintain the generator representation by expanding the negative generator border.

Based on the concept of negative generator border, a new algorithm, STUM, is proposed to maintain the generator representation for support threshold adjustment. Extensive experiments are conducted to evaluate the effectiveness of the proposed algorithm. The experimental results show that, in general, STUM outperforms the other methods significantly. For some particular datasets, STUM is faster than the state-of-the-art methods by more than an order of magnitude.

In addition, we theoretically demonstrated that the concept of negative generator border can also be applied to the maintenance of other data updates. We also show that the proposed method can be extended to two other frequent

Table 2. Average speed-up achieved by STUM. T_{comp} denotes the computational time of the comparing algorithms, and T_{STUM} is that of the proposed algorithm.

T_{comp}/T_{STUM}	accidents $ms\% = 50\%$	gazelle $ms\% = 0.5\%$	mushroom $ms\% = 0.5\%$	T1014D100K $ms\% = 1\%$
ZIGZAG	2.8	19.4	10.8	19.5
GC-growth	31	17.2	11.6	31.5
Border	230	828	1334	927
Border(prefixTree)	3.7	2.5	4.4	87.8

pattern representations — the free-sets and disjunctive-free sets. The realization of these theoretical ideas could serve as potential future works.

References

- [1] R. Agrawal, T. Imielinski, A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.
- [2] Y. Aumann, R. Feldman, O. Lipshtat, H. Manilla. Borders: An efficient algorithm for association generation in dynamic databases. In *JIIS*, (12) page 61–73, 1999.
- [3] R. J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD*, pages 85–93, 1998.
- [4] A. Bykowski, C. Rigotti. A condensed representation to find frequent patterns. In *PODS*, 2001.
- [5] C. Chang, et al. Enhancing SWF for incremental association mining by itemset maintenance. In *PAKDD*, pages 301–312, 2003.
- [6] D. Cheung, J. Han, V. T. Y. Ng, C. Y. Wong. Maintenance of discovered association rules in large databases: an incremental update techniq. In *ICDE*, pages 106–114, 1996.
- [7] Y. Chi, H. Wang, P. S. Yu, R. R. Muntz. Moment: Maintaining closed frequent itemsets over a stream sliding window. In *ICDM*, pages 59–66, 2004.
- [8] Frequent Itemset Mining Dataset Repository. <http://fimi.cs.helsinki.fi>
- [9] J. Han, J. Pei, Y. Yin. Mining frequent patterns without candidates generation. In *SIGMOD*, pages 1–12, 2000.
- [10] M. Kryszkiewicz. Concise representation of frequent patterns based on disjunction-free generators. In *ICDM*, pages 305–312, 2001
- [11] C-H. Lee, C-R. Lin, M-S. Chen. Sliding window filtering: An efficient method for incremental mining on a time-variant database. *Information Systems*, 30(3):227-244, 2005.
- [12] H. Li, J. Li, L. Wong, M. Feng, Y-P. Tan. Relative risk and odds ratio: A data mining perspective. In *PODS*, pages 368–377, 2005.
- [13] H. Mannila, H. Toivonen. Levelwise search and borders of theories in knowledge discovery. In *Data Mining and Knowledge Discovery*, 1(2):241–258,1997.
- [14] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, pages 398–416, 1999.
- [15] A.A. Veloso, W.Meira Jr., M.B. de Carvalho B. Possas, S. Parthasarathy, M.J. Zaki. Mining frequent itemsets in evolving databases. In *SIAM*, 2002.