Data & Knowledge Engineering 68 (2009) 481–498

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

Mining non-derivable frequent itemsets over data stream ☆

Haifeng Li [a], Hong Chen [a,b,*]

[a] School of Information, Renmin University of China, Beijing 100872, China
[b] Key Laboratory of Data Engineering and Knowledge Engineering, MOE, Beijing 100872, China

ARTICLE INFO

Article history:
Received 24 August 2008
Received in revised form 19 December 2008
Accepted 6 January 2009
Available online 22 January 2009

Keywords:
Stream
Non-derivable frequent itemsets
Data mining

ABSTRACT

Non-derivable frequent itemsets are one of several condensed representations of frequent itemsets, which store all of the information contained in frequent itemsets using less space, thus being more suitable for stream mining. This paper considers a problem that to the best of our knowledge has not been addressed, namely, how to mine non-derivable frequent itemsets in an incremental fashion. We design a compact data structure named *NDFIT* to efficiently maintain a dynamically selected set of itemsets. In *NDFIT*, the nodes are divided into four categories to reduce the redundant computational cost based on their properties. Consequently, an optimized algorithm named *NDFIoDS* is proposed to generate non-derivable frequent itemsets over stream sliding window. Our experimental results show that this method is effective and more efficient than previous approaches.

1. Introduction

Frequent itemset mining is a traditional and important problem in data mining. An itemset is frequent if its support is not less than a threshold specified by users. Traditional frequent itemset mining approaches have mainly considered the problem of mining static transaction databases. In these methods, transactions are stored in secondary storage so that multiple scans over the data can be performed. Three kinds of frequent itemset mining approaches over static databases have been proposed: reading-based [3], writing-based [23], and pointer-based [27]. Han et al. [22] presented a comprehensive survey of frequent itemset mining and discussed research directions.

With the growing number of data stream applications [4], such as stock analysis, wireless sensor networks management, web log analysis, and networks monitoring, more studies have begun to focus on stream mining. Data streams have posed new challenges: a data stream is continuous, unlimited, and high speed, which means that data cannot be stored in secondary storage. In addition, only one or limited scans can be performed when the data arrives, so the stream algorithm must operate incrementally. Furthermore, the data distribution may dynamically change with time, so a concept-drift problem [32] must be solved.

Many methods focusing on frequent itemset mining over a stream have been proposed. Giannella et al. [19] proposed *FP-Stream* to mine frequent itemsets, which was efficient when the average transaction length was small; Manku and Motwani [34] used lossy counting to mine frequent itemsets; Chang and Lee [12–14] focused on mining the recent itemsets, which used a regression parameter to adjust and reflect the importance of recent transactions; Teng et al. [39] presented the *FTP-DS* method to compress each frequent itemset; Cormode et al. [15], Asai et al. [1] separately focused on multiple-level frequent itemset

☆ This research is supported by the National High Technology Research and Development Program of China (2008AA01Z133), the National Science Foundation of China (60673138,60603046), Key Program of Science Technology Research of MOE (106006), Program for New Century Excellent Talents in University and Chinese National Programs for Science and Technology Development (2005BA112A02).

* Corresponding author. Address: School of Information, Renmin University of China, Beijing 100872, China.
E-mail addresses: mydlhf@ruc.edu.cn (H. Li), chong@ruc.edu.cn (H. Chen).

0169-023X/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2009.01.002

mining and semi-structure stream mining; Cormode and Muthukrishnan [17] proposed a group testing technique, and Jin et al. [26] proposed a hash technique to improve frequent itemset mining; Jin and Agarwal [24] proposed an in-core mining algorithm to speed up the runtime when distinct items are huge or minimum support is low; Koh and Shin [28] presented two methods separately based on the average time stamps and frequency-changing points of patterns to estimate the approximate supports of frequent itemsets; Calders et al. [6] focused on mining a stream over a flexible sliding window; Leung and Khan [29] was a block-based stream mining algorithm with *DSTree* structure; Mozafari et al. [35] used a verification technique to mine frequent itemsets over a stream when the sliding window is large; Cheng et al. [16] reviewed the main techniques of frequent itemset mining algorithms over data streams and classified them into categories to be separately addressed. Given these algorithms, the runtime could be reduced, but the mining results were huge when the minimum support was low; consequently, the condensed representations of frequent itemsets including closed itemsets [37,18,25,33], maximal itemsets [30,36,41], free itemsets [5], representative itemsets [42], approximate *k*-sets [2], weighted itemsets [40], and non-derivable itemsets [7–11,20] were proposed; in addition, [38] focused on discovering a minimal set of unexpected itemsets.

The concept of non-derivable itemsets (*NDI*) was first proposed in 2002 and is based on deduction rules [7]. An itemset is non-derivable if its support cannot be derived from the supports of its subsets; we will discuss the details in Section 2. Non-derivable itemset mining is a new and important problem. On the one hand, as one of the condensed representations, non-derivable itemsets only store the non-redundant cover of frequent itemsets and in many practical cases this cover is considerably less than all frequent itemsets, thus reducing the space cost in mining. On the other hand, compared to another condensed representation (closed itemsets), non-derivable itemsets make association rules more concise and understandable to the user [20]: closed itemset pruning is less efficient on association rule mining because using closed itemsets only prune out rules whose confidences are one; in comparison to that, the confidence can have any value to prune the derivable rules in non-derivable association rule mining, consequently, over 99–99.99% of rules are derivable and can be pruned as presented in the experimental results in [20], and it is much easier for user to understand the results. Furthermore, if X is a non-closed itemset, Y is an itemset in the closure of X, and $X \Rightarrow Z$ is an association rule, then rules $XY \Rightarrow Z$ and $X \Rightarrow YZ$ are redundant and can be pruned, but user cannot know the pruning reason without knowing the assumptions; in other words, association rule mining with closed itemsets makes assumptions in the form of fixing an inference system or an estimation method, whereas non-derivable association rule mining needs no such assumptions, so user can understand what was pruned more easily.

Recently, many non-derivable itemset mining algorithms over static datasets have been proposed. Calders and Goethals [7] used the *a priori*-based method *NDI* to mine non-derivable itemsets; Calders and Goethals [8] presented the depth-first non-derivable itemset mining method *dfNDI* to improve performance; in [9], quick inclusion–exclusion principles were proposed based on arrays to accelerate the non-derivable itemset mining speed; Calders and Goethals [10] surveyed non-derivable itemset mining in comparison to mining of the other condensed representations; in addition, [20] discussed the non-derivable association rules mining method.

As has been proven in [11], non-derivable itemsets are the smallest and lossless condensed representation, so they are very suitable for stream mining. However, little work has been done on this problem. In this paper, we aim to test the feasibility of online incremental mining of non-derivable itemsets. We propose an algorithm, *NDFIoDS*.

NDFIoDS is inspired by *MOMENT* [18], a novel algorithm mining closed frequent itemsets over streams. *MOMENT* builds a closed enumeration tree (*CET*), in which nodes are scheduled according to their lexicographical order to build partial monotonicity. Based upon that, nodes are divided into four categories for efficient pruning. Our method and *MOMENT* are similar in many ways: both mine frequent itemsets over data streams, and they both use the same model, i.e., the sliding window model; moreover, they aim to obtain the condensed representations, and they use the *a priori* property to prune infrequent itemsets. However, they are different in the details. First, *MOMENT* is an algorithm for mining closed frequent itemsets, and our method is intended for mining non-derivable frequent itemsets, which are more condensed. Second, even though both approaches use trees to maintain frequent itemsets and divide nodes into categories, the pruning method is different because they have different properties: *MOMENT* uses the monotonicity of the itemsets lexicographical order, and our method uses the monotonicity of derivable itemsets. Third, the bottlenecks of these methods are different: *MOMENT* needs to compare the newly generated itemset with the existing closed itemsets to decide whether it is closed, and our method has to compute the bounds of the new itemset to determine if it is non-derivable. Finally, although both methods are incremental, *MOMENT* is more intuitive than our method; we need to supply a theoretical guarantee for incremental mining. We will compare the performances of the two methods in our experimental results with a further discussions.

NDFIoDS addresses the problem of incremental non-derivable mining, and its main contributions are summarized as follows:

1. First, a simple and compact data structure named *NDFIT* is employed to store the information of non-derivable itemsets and other helpful itemsets. We divide the *NDFIT* nodes into four categories; thus, effective pruning is performed using the properties of infrequent and derivable nodes.
2. Second, when the stream window continues and a new transaction is added and an existing one is deleted, we theoretically prove the incremental properties of the nodes; i.e., some kinds of nodes will definitely not change their types, thus avoiding redundant computation.
3. Third, both transaction addition and deletion may cause the frequent shift of node types regardless of whether or not the intersection between the node and the transaction is null, resulting in redundant node generation, insertion and deletion. We combine these two operations to optimize the mining process, so that the running time is reduced.

4. Finally, we evaluate the *NDFIoDS* algorithm on two synthetic and four real-life datasets in comparison to the state-of-the-art non-derivable frequent itemset mining method *dfNDI*, the closed frequent itemset mining method *MOMENT* and the frequent itemset mining method *AFOPT*. The experimental results show that *NDFIoDS* is effective and efficient.

The rest of this paper is organized as follows: In Section 2, we present the preliminaries of frequent and non-derivable itemsets and define the mining problem. Section 3 illustrates the *NDFIoDS* algorithm in detail. Section 4 evaluates the performance of *NDFIoDS* with experimental results. Finally, Section 5 concludes this paper with a discussion of future research considerations.

2. Preliminaries and problem statement

A brief review of the relevant concepts of frequent and non-derivable itemsets is presented in this section; based upon this, we define the problem addressed in this paper.

2.1. Preliminaries

2.1.1. Frequent itemsets

Given a set of distinct items $\Gamma = \{i_1, i_2, \ldots, i_n\}$ where $|\Gamma| = n$ denotes the size of Γ, a subset $X \subseteq \Gamma$ is called an itemset; suppose $|X| = k$, we call X a k-itemset. A concise expression of itemset $X = \{x_1, x_2, \ldots, x_m\}$ is $x_1x_2 \cdots x_m$. A database $D = \{T_1, T_2, \ldots, T_v\}$ is a collection wherein each transaction is a subset of Γ, namely an itemset. Each transaction $T_i (i = 1 \cdots v)$ is related to an id, i.e., the id of T_i is i. The absolute support (*AS*) of an itemset α, also called the weight of α, is the number of transactions which cover α, denoted $\Lambda(\alpha) = \{|T||T \in D \wedge \alpha \subseteq T\}$; the relative support (*RS*) of an itemset α is the ratio of *AS* with respect to $|D|$, denoted $\Lambda_r(\alpha) = \frac{\Lambda(\alpha)}{|D|}$. Given a relative minimum support $\lambda (0 \leqslant \lambda \leqslant 1)$, itemset α is frequent if $\Lambda_r(\alpha) \geqslant \lambda$.

2.1.2. Deduction rules

We will revisit the deduction rules introduced in [7] because they support the concept of non-derivable itemsets.

Given an item a, a negation of a is $\bar{a}$. A transaction is said to cover $\bar{a}$ if it does not cover a. A generalized itemset is a conjunction of items and negations of items, which is denoted $G = X\overline{Y}$, where X contains the items and Y contains the negation of items. A transaction T is said to support G if $X \subseteq T$ and $T \cap Y = \phi$. Let $\Gamma = \{a, b, c, d\}$, an example of a generalized itemset is $G = ab\overline{cd}$, and for a transaction database $\{ab, abc\}$, ab is said to support G because it contains a and b and neither c nor d; on the contrary, abc does not support G because it contains c.

We say a generalized itemset $G = X\overline{Y}$ is based on an itemset I if $I = X \cup Y$; thus, the deduction rules are derived from the inclusion–exclusion principles [21], which are expressed as $\Lambda(X\overline{Y}) = \sum_{J:X \subseteq J \subseteq I} (-1)^{|J \setminus X|} \Lambda(J)$ for a generalized itemset $X\overline{Y}$ based on itemset I. Consequently, the following theorem [7] is obtained.

Theorem 1. *Let Γ be the distinct items set and I be an itemset; if $X \subseteq I \subseteq \Gamma$ and $Y = I \setminus X$, then the following inequalities hold:*

$$\begin{aligned} \Lambda(I) &\leqslant \sum_{J:X \subseteq J \subset I} (-1)^{|I \setminus J|+1} \Lambda(J) \quad \text{is odd} \\ \Lambda(I) &\geqslant \sum_{J:X \subseteq J \subset I} (-1)^{|I \setminus J|+1} \Lambda(J) \quad \text{is even} \end{aligned} \tag{1}$$

Example 1. The deduction rule in Theorem 1 is denoted $R_X(I)$ for each X. We use Table 1 to present the rules of an itemset $I = abc$.

2.1.3. Non-derivable itemsets

For an itemset I, the bound $ul_X(I) = \sum_{J:X \subseteq J \subset I} (-1)^{|I \setminus J|+1} \Lambda(J)$ can be obtained from the deduction rules for each $X \subseteq I$; when $I \setminus X$ is odd, $ul_X(I)$ is upper bound, denoted $u_X(I)$, and when $I \setminus X$ is even, $ul_X(I)$ is lower bound, denoted $l_X(I)$. If we use $ml(I)$ to

Table 1
Deduction rules of *abc*.

I	X	$R_X(I)$
abc	ϕ	$\Lambda(abc) \leqslant \Lambda(ab) + \Lambda(ac) + \Lambda(bc) - \Lambda(a) - \Lambda(b) - \Lambda(c) + \Lambda(\phi)$
abc	*a*	$\Lambda(abc) \geqslant \Lambda(ab) + \Lambda(ac) - \Lambda(a)$
abc	*b*	$\Lambda(abc) \geqslant \Lambda(ab) + \Lambda(bc) - \Lambda(b)$
abc	*c*	$\Lambda(abc) \geqslant \Lambda(ac) + \Lambda(bc) - \Lambda(c)$
abc	*ab*	$\Lambda(abc) \leqslant \Lambda(ab)$
abc	*ac*	$\Lambda(abc) \leqslant \Lambda(ac)$
abc	*bc*	$\Lambda(abc) \leqslant \Lambda(bc)$
abc	*abc*	$\Lambda(abc) \geqslant 0$

denote the maximal lower bound and use $mu(I)$ to denote the minimal upper bound, i.e., $ml(I) = \max\{l_X(I)|X \subseteq I \wedge |I \setminus X| \text{ is even}\}$ and $mu(I) = \min\{u_X(I)|X \subseteq I \wedge |I \setminus X| \text{ is odd}\}$, then the absolute support $\Lambda(I) \in [ml(I), mu(I)]$. If $ml(I) = mu(I)$, then $\Lambda(I) = ml(I) = mu(I)$; thus, I is called the derivable itemset. A derivable itemset is monotone [7], i.e., for two itemsets $S \subseteq T$, if S is derivable, then T is also derivable. On the other hand, if $ml(I) \neq mu(I)$, itemset I is non-derivable. In general, the length $|I|$ is inversely proportional to $mu(I) - ml(I)$, and thus if $|I|$ is large enough, I must be derivable; Calders and Goethals [9] gives the detailed properties as follows.

Lemma 2 [9]. *Given itemset $I \subset \Gamma$ and item $a \in \Gamma \setminus I$, then $mu(I \cup a) - ml(I \cup a) \leqslant \frac{mu(I)-ml(I)}{2}$.*

Lemma 3 [9]. *The width of the intervals exponentially shrinks with the size of the itemsets. Hence, for a dataset D and an itemset $I \subset \Gamma$, if $|I| > \log_2(|D|) + 1$, then I must be derivable in D.*

Example 2. Given a simple transaction database $\{abc, ab, ac, bc\}$, then

$$\Lambda(abc) \leqslant \begin{cases} u_{ab}(abc) = 2 \\ u_{ac}(abc) = 2 \\ u_{bc}(abc) = 2 \\ u_{\phi}(abc) = 1 \end{cases} \Rightarrow \Lambda(abc) \leqslant 1 \text{ and } \Lambda(abc) \geqslant \begin{cases} l_a(abc) = 1 \\ l_b(abc) = 1 \\ l_c(abc) = 1 \\ l_{abc}(abc) = 0 \end{cases} \Rightarrow \Lambda(abc) \geqslant 1, \text{ i.e., } \Lambda(abc) \in [1, 1];$$

thus, abc is a derivable itemset. On the other hand,

$$\Lambda(ab) \leqslant \begin{cases} u_a(ab) = 3 \\ u_b(ab) = 3 \end{cases} \Rightarrow \Lambda(ab) \leqslant 3 \text{ and } \Lambda(ab) \geqslant \begin{cases} l_{\phi}(ab) = 2 \\ l_{ab}(ab) = 0 \end{cases} \Rightarrow \Lambda(ab) \geqslant 2, \text{ i.e, } \Lambda(ab) \in [2, 3];$$

hence, ab is a non-derivable itemset; neither are bc and ac.

Furthermore, corresponding to Lemma 2, $ab \subset abc$, so $mu(abc) - ml(abc) = 0 \leqslant \frac{mu(ab)-ml(ab)}{2} = 1/2$ holds.

2.2. Problem definition

Generally, three kinds of models are used when mining streams: the landmark model, which uses the entire stream as a data source; the damped model, which uses a time-decaying weight for transactions; and the sliding window model, which continuously adds and deletes transactions to maintain a fixed window size. The number of non-derivable itemsets will increase with new transaction being added in the first two models. If we use the absolute minimum support to perform the mining, it will theoretically require unlimited memory; on the other hand, if we use the relative minimum support, the results will be rechecked once the number of transactions changes, and thus the computational cost will be huge. Consequently, we choose the sliding window model in this paper because it can reflect the time characteristic of a stream with limited memory, i.e., the problem addressed in this paper is to generate non-derivable frequent itemsets in the recent N transactions in a stream. Fig. 1 is an example with $\Gamma = \{a, b, c, d\}$ and window size $N = 4$.

3. *NDFIoDS* method

In this section, we introduce an in-memory structure, the *non-derivable frequent itemsets tree*, *NDFIT* for short, to maintain the dynamically selected set of itemsets. Then we theoretically prove the incremental properties of non-derivable itemsets during transaction addition and deletion. We further discuss the *NDFIT* node properties for computational optimization and space pruning. Finally, the incremental mining algorithm, *NDFIoDS*, is presented in detail.

3.1. Naive method

In streams, users are often interested in the new real-time frequent itemsets. Consequently, we immediately come up with a naive method for the window-based model: to regenerate non-derivable frequent itemsets from the entire window

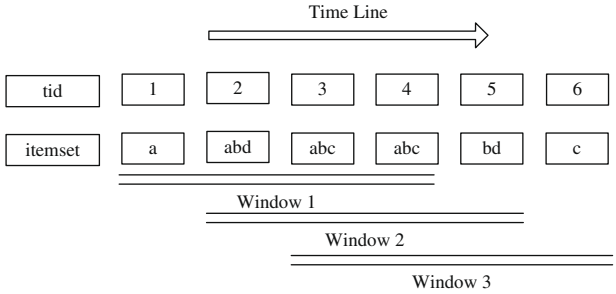


Fig. 1. A running example of sliding window.

whenever a new transaction is added and an old transaction is deleted. In this paper, we implement the naive method with calling the traditional state-of-the-art algorithm *dfNDI*.

3.2. Non-derivable frequent itemsets tree

3.2.1. NDFIT description

To compute the bounds of an itemset, we need to know all of the supports of its subsets, thus a quick itemset search is necessary in the mining process. An in-memory data structure, the *non-derivable frequent itemsets tree* (*NDFIT*), is proposed to store the itemsets relationship. Fig. 2 shows the initial *NDFIT* for the first window in Fig. 1, with relative minimum support $\lambda = 0.5$. In *NDFIT*, nodes are divided into four categories: we initially divide nodes by their supports, i.e., infrequent nodes and frequent nodes; then, we divide frequent nodes by their derivable property, that is, derivable nodes and non-derivable nodes; we finally divide non-derivable nodes according to their children node types, that is, intermediate nodes and steady nodes, so that transaction addition and deletion can be optimized separately. The four types are described as follows.

Infrequent node. A node n_I is an infrequent node if its relative support is smaller than the minimum support, i.e., $\Lambda_r(I) < \lambda$. An infrequent node has no children according to the *a priori* property [3]. In Fig. 2, $d : 1$ is an infrequent node.

Promising node. A node n_I is a promising node if it is frequent and derivable and its parents are non-derivable. In Fig. 2, $ab : 3$ and $ac : 2$ are promising nodes (represented by parentheses) because $mu(ab) = ml(ab) = 3$ and $mu(ac) = ml(ac) = 2$, and their parents, *a*, *b* and *c*, are all non-derivable nodes. A promising node has no children according to the monotonicity of derivable itemset; thus, *abc*, the child of *ab* and *ac*, is pruned (hence, it does not appear in the tree) even though it is frequent.

Intermediate non-derivable node. A node n_I is an intermediate non-derivable node (briefly: *intermediate node*) if it is frequent and non-derivable and it has no non-derivable child. In Fig. 2, $a : 4$ is an intermediate node (represented by square brackets) because $ml(a) = 0$ and $mu(a) = 4$, and its children *ab* and *ac* are derivable nodes; in addition, $bc : 2$ is also an intermediate node, because $ml(bc) = 1$ and $mu(bc) = 2$, and it has no children.

Steady non-derivable node. A node n_I is a steady non-derivable node (*steady node* in abbreviation) if it is frequent and non-derivable and it has at least one non-derivable child. In Fig. 2, $b : 3$ and $c : 2$ are both steady nodes (represented by a rectangle) because they have a non-derivable child *bc*.

3.2.2. NDFIT implementation

In *NDFIT*, each node n_I, a representation of itemset *I*, is a 3-tuple $\langle iss, wt, tp\rangle$, where *iss*, on the left of ":", is the itemset *I*; *wt* is the absolute support of *I* on the right of ":"; and the node status is recorded by *tp*. In addition, nodes are arranged in lexicographical order for convenient computation. For example, in Fig. 2, *a* is smaller than *ab*, *ab* is smaller than *ac*, and *ac* is smaller than *b*, denoted as $a \prec ab \prec ac \prec b$.

We frequently compute node bounds to determine type, which means we need to quickly locate subsets for efficient computation. As a result, we also build the double linked list at each tree level. When using these pointers between parent and children, as well as the pointers between siblings, the speed of bounds computation increases significantly.

3.3. NDFIT initialization

Originally, the sliding window is empty and the *NDFIT* has the root node ϕ; transactions are added one by one until the sliding window is full. In such a case, our method actually works under the landmark model, and the *NDFIT* is updated once for each new transaction added. In order to maintain consistency with the following *NDFIT* maintenance process, we still compute the absolute minimum support as $\Lambda = \Lambda_r \times |D|$, where $|D|$ is the window size. We call this process the *NDFIT Initialization*.

3.4. NDFIT maintenance

When the sliding window is full, old transactions will be deleted once new transactions are added. In this phase, we will update node supports. In addition, to determine the node types, we have to compute the bounds of each node once a

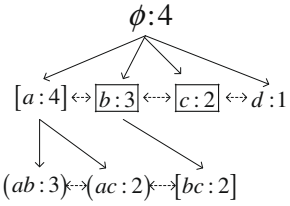


Fig. 2. Initial *NDFIT* of window 1.

transaction is added or deleted, which is less efficient in practice. Consequently, if we can predict which nodes will change their type and which nodes will not, the computational cost can be reduced.

Lemma 4 [10] suggests that the bound may become less tight when a new transaction is added, and may become more tight when an existing transaction is deleted. However, this suggestion is still unproven and ambiguous, requiring further studies to prove and test its feasibility. New problems are therefore posed: Is this suggestion right? How much will the bound change? Is that useful for our node type prediction? We answer these questions separately for transaction addition and deletion.

Lemma 4. *Let I be an itemset, $X \subseteq I$ and $Y = I \setminus X$, and let $ul_X(I)$ be the bound of I, then $|\Lambda(I) - ul_X(I)| = \Lambda(X\overline{Y})$.*

3.4.1. Adding a new transaction

Theorem 5. *Given an added transaction T and a node n_I, $N_{new} = T \cap I$ and $X \subset I$. If $l_X(I)$ and $u_X(I)$ denote the original lower and upper bounds of I, and $l'_X(I)$ and $u'_X(I)$ denote the updated ones. The following conclusions hold:*

if $N_{new} \subset I$ and $N_{new} = X$, then

$$\begin{cases} l'_X(I) = l_X(I) - 1 & |I \setminus X| \quad \text{is even} \\ u'_X(I) = u_X(I) + 1 & |I \setminus X| \quad \text{is odd} \end{cases} \tag{2}$$

if $N_{new} \subset I$ and $N_{new} \neq X$, then

$$\begin{cases} l'_X(I) = l_X(I) & |I \setminus X| \quad \text{is even} \\ u'_X(I) = u_X(I) & |I \setminus X| \quad \text{is odd} \end{cases} \tag{3}$$

if $N_{new} = I$, then

$$l'_X(I) = l_X(I) + 1 \quad \text{and} \quad u'_X(I) = u_X(I) + 1 \tag{4}$$

Proof. From the proposition, N_{new} is definitely covered by or equal to I, i.e. $N_{new} \subseteq I$. According to Theorem 1, when $X = I$, the lower or upper bounds of I will never be changed, which is 0, so we ignore this aspect. When $X \subset I$, for any J satisfies $X \subseteq J \subseteq N_{new}$, the support $\Lambda(J)$ will be increased by 1. We use $ul_X(I)$ to denote the original bound of I for any X, and use $ul'_X(I)$ to denote the updated one; besides, C_x^y is used to denote the combination(number of ways to select y items from x items), then the discussion is divided into two parts.

1. If $N_{new} \subset I$, then

$$ul_X(I) = \sum_{J:X \subseteq J \subset I} (-1)^{|I\setminus J|+1} \Lambda(J),$$

$$ul'_X(I) = \sum_{J:X \subseteq J \subseteq N_{new} \subset I} (-1)^{|I\setminus J|+1} (\Lambda(J) + 1) + \sum_{J:X \subseteq J \not\subseteq N_{new} \subset I} (-1)^{|I\setminus J|+1} (\Lambda(J)).$$

1.1. If $|I \setminus X|$ is even, the following equation of updated lower bound holds

$$l'_X(I) = l_X(I) + \sum_{J:X \subseteq J \subseteq N_{new} \subset I} (-1)^{|I\setminus J|+1} = l_X(I) - C^0_{|N_{new}\setminus X|} + C^1_{|N_{new}\setminus X|} - \cdots (+/-) C^{|N_{new}\setminus X|}_{|N_{new}\setminus X|} = l_X(I) - (1-1)^{|N_{new}\setminus X|}.$$

Consequently, (1) if $X = N_{new}$, i.e., $|N_{new} \setminus X| = 0$, we can get $l'_X(I) = l_X(I) - 0^0 = l_X(I) - 1$ (An example is shown in Fig. 3a.) (2) if $X \subset N_{new}$, we can get $l'_X(I) = l_X(I) - 0^{t(t\neq 0)} = l_X(I)$ (An example is shown in Fig. 3c.)

1.2. If $|I \setminus X|$ is odd, the following equation of updated upper bound holds

$$u'_X(I) = u_X(I) + \sum_{J:X \subseteq J \subseteq N_{new} \subset I} (-1)^{|I\setminus J|+1} = u_X(I) + C^0_{|N_{new}\setminus X|} - C^1_{|N_{new}\setminus X|} + \cdots (+/-) C^{|N_{new}\setminus X|}_{|N_{new}\setminus X|} = u_X(I) + (1-1)^{|N_{new}\setminus X|}.$$

Consequently, (1) if $X = N_{new}$, i.e., $|N_{new} \setminus X| = 0$, we can get $u'_X(I) = u_X(I) + 0^0 = u_X(I) + 1$ (An example is shown in Fig. 3b.). (2) if $X \subset N_{new}$, we can get $u'_X(I) = u_X(I) + 0^{t(t\neq 0)} = u_X(I)$ (An example is shown in Fig. 3d.)

2. If $N_{new} = I$, then

$$ul_X(I) = \sum_{J:X \subseteq J \subset I} (-1)^{|I\setminus J|+1} \Lambda(J),$$

$$ul'_X(I) = \sum_{J:X \subseteq J \subset N_{new} = I} (-1)^{|I\setminus J|+1} (\Lambda(J) + 1).$$

2.1. If $|I \setminus X|$ is even, the following equation of updated lower bound holds.

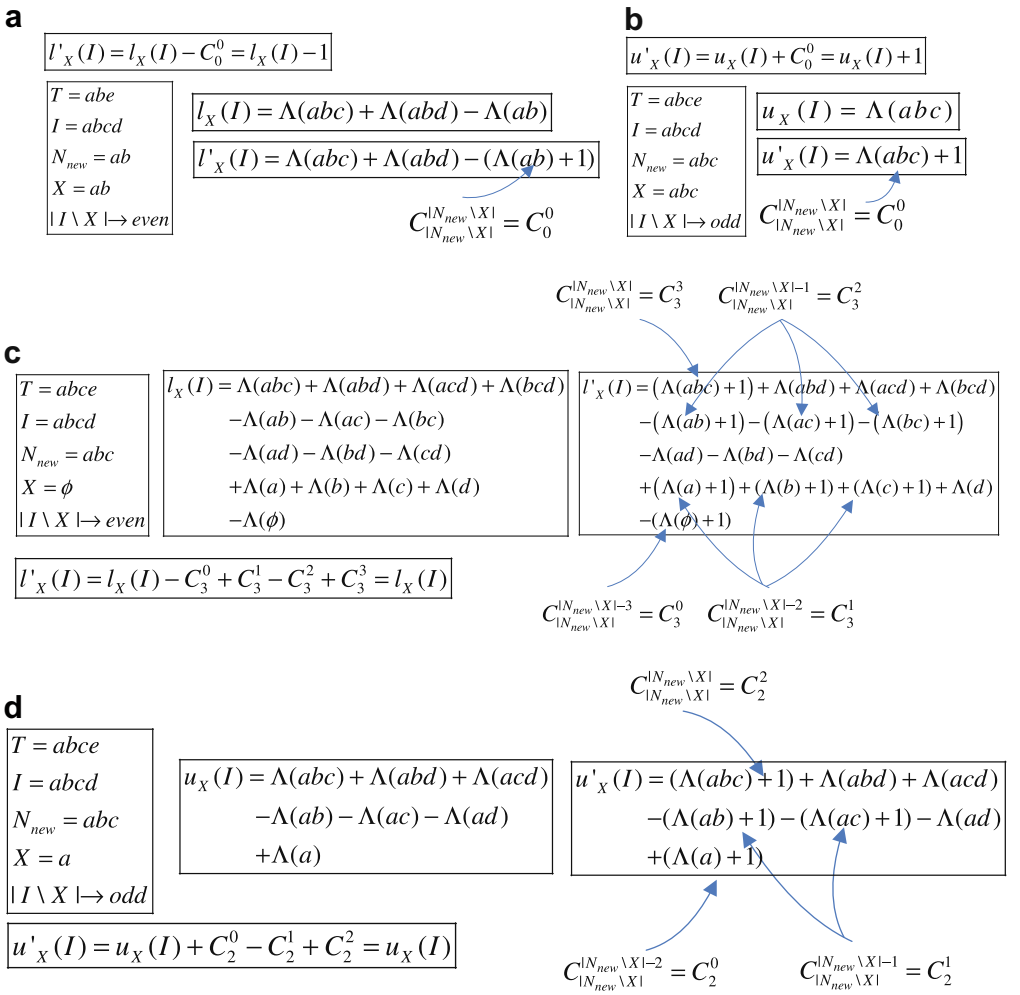


Fig. 3. Example for $N_{\text{new}} \subset I$ (a) $N_{\text{new}} = X$ and $|I \setminus X|$ is even (b) $N_{\text{new}} = X$ and $|I \setminus X|$ is odd (c) $X \subset N_{\text{new}}$ and $|I \setminus X|$ is even (d) $X \subset N_{\text{new}}$ and $|I \setminus X|$ is odd.

$$l'_X(I) = l_X(I) + \sum_{J:X \subseteq J \subset N_{\text{new}}=I} (-1)^{|I \setminus J|+1} = l_X(I) - C^0_{|N_{\text{new}} \setminus X|} + C^1_{|N_{\text{new}} \setminus X|} - \cdots (+/-) C^{|N_{\text{new}} \setminus X|-1}_{|N_{\text{new}} \setminus X|},$$

$C^{|N_{\text{new}} \setminus X|-1}_{|N_{\text{new}} \setminus X|}$ is positive because its coefficient is $(-1)^{|I|-(|N_{\text{new}}|-1)+1}$, then

$$l'_X(I) = l_X(I) - C^0_{|N_{\text{new}} \setminus X|} + C^1_{|N_{\text{new}} \setminus X|} - \cdots + C^{|N_{\text{new}} \setminus X|-1}_{|N_{\text{new}} \setminus X|} - C^{|N_{\text{new}} \setminus X|}_{|N_{\text{new}} \setminus X|} + C^{|N_{\text{new}} \setminus X|}_{|N_{\text{new}} \setminus X|} = l_X(I) - (1-1)^{|N_{\text{new}} \setminus X|} + 1.$$

Because $X \subset I = N_{\text{new}}$, i.e., $|N_{\text{new}} \setminus X| \neq 0$, then $l'_X(I) = l_X(I) - 0^{t(t \neq 0)} + 1 = l_X(I) + 1$ holds (An example is shown in Fig. 4a.)

2.2. If $|I \setminus X|$ is odd, the following equation of updated upper bound holds

$$u'_X(I) = u_X(I) + \sum_{J:X \subseteq J \subset N_{\text{new}}=I} (-1)^{|I \setminus J|} = u_X(I) + C^0_{|N_{\text{new}} \setminus X|} - C^1_{|N_{\text{new}} \setminus X|} - \cdots (+/-) C^{|N_{\text{new}} \setminus X|-1}_{|N_{\text{new}} \setminus X|},$$

$C^{|N_{\text{new}} \setminus X|-1}_{|N_{\text{new}} \setminus X|}$ is positive because its coefficient is $(-1)^{|I|-(|N_{\text{new}}|-1)+1}$, then

$$u'_X(I) = u_X(I) + C^0_{|N_{\text{new}} \setminus X|} - C^1_{|N_{\text{new}} \setminus X|} + \cdots + C^{|N_{\text{new}} \setminus X|-1}_{|N_{\text{new}} \setminus X|} - C^{|N_{\text{new}} \setminus X|}_{|N_{\text{new}} \setminus X|} + C^{|N_{\text{new}} \setminus X|}_{|N_{\text{new}} \setminus X|} = u_X(I) + (1-1)^{|N_{\text{new}} \setminus X|} + 1.$$

Because $X \subset I = N_{\text{new}}$, i.e., $|N_{\text{new}} \setminus X| \neq 0$, then $u'_X(I) = u_X(I) + 0^{t(t \neq 0)} + 1 = u_X(I) + 1$ holds (An example is shown in Fig. 4b.) □

Lemma 6. *For a node n_I, adding a new transaction T will not decrease $mu(I) - ml(I)$, and it will increase $mu(I) - ml(I)$ by at most 1.*

Proof. For each $X \subset I$, we use $l_X(I)$ and $u_X(I)$ to denote the original lower and upper bounds of I, and use $l'_X(I)$ and $u'_X(I)$ to denote the updated ones. Similarly, we use $ml(I)$ and $mu(I)$ to denote the original maximal lower and minimal upper bounds,

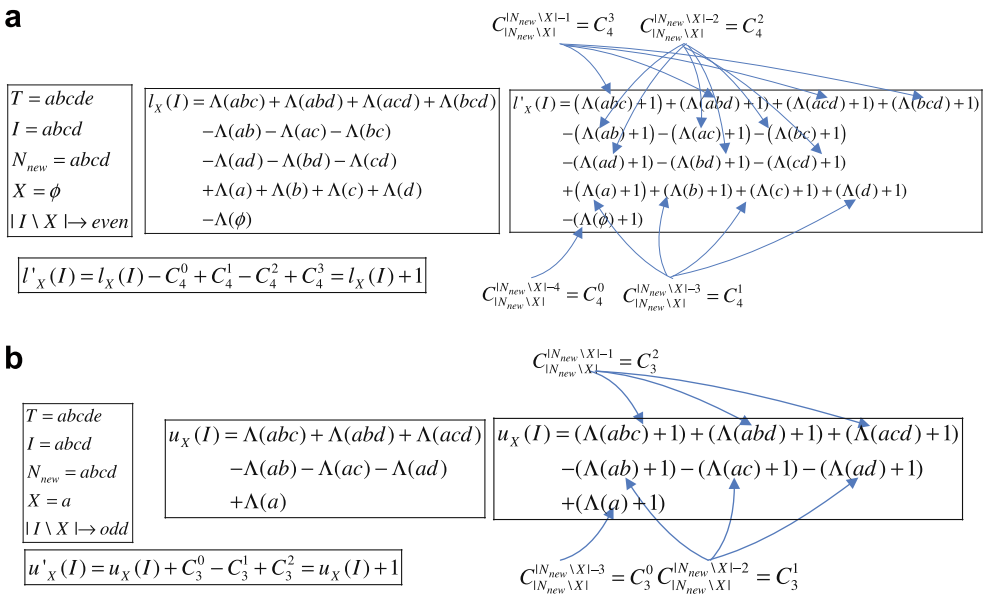


Fig. 4. Example for $N_{\text{new}} = I$ (a) $|I \setminus X|$ is even; (b) $|I \setminus X|$ is odd.

and use $ml'(I)$ and $mu'(I)$ to denote the updated bounds. When a new transaction T is added, $N_{\text{new}} = I \cap T$, then $N_{\text{new}} = I$ and $N_{\text{new}} \subset I$ will not happen at the same time, so we discuss these two aspects separately.

If $N_{\text{new}} = I$, for each X, $l_X(I)$ and $u_X(I)$ are increased by 1 according to Eq. (4); thus, $ml(I)$ and $mu(I)$ are both increased by 1, i.e., $mu'(I) - ml'(I) = mu(I) - ml(I)$, so in this condition, $mu(I) - ml(I)$ is unchanged.

If $N_{\text{new}} \subset I$, since N_{new} is a fixed itemset, for each X, only one X value x satisfies $x = N_{\text{new}}$, so only one of the formulas in Eq. (2) holds; without a loss of generality, we suppose $|I \setminus x|$ is even, in which case $l'_x(I) = l_x(I) - 1$ holds; if $l_x(I) = ml(I)$ and no other values of $X(X \neq x$ and $I \setminus X$ is even) satisfy $l_X(I) = ml(I)$, then $ml'(I) = ml(I) - 1$, otherwise, $ml'(I) = ml(I)$; furthermore, for the other values of X, i.e., $X \neq x = N_{\text{new}}$, then $l_X(I)$ and $u_X(I)$ keep their original values according to Eq. (3), and thus $ml(I)$ and $mu(I)$ are not changed for these X. To summarize, for all X, $mu(I)$ will keep its value, and $ml(I)$ will be decreased by 0 or 1, as a result, $mu(I) - ml(I)$ will be increased by at most 1. The same conclusion can be obtained when $|I \setminus x|$ is odd. □

Example 3. Fig. 5 shows the *NDFIT* with transaction *bd* added to window 1. Since $|D|$ changes to 5, the minimal upper bound of n_a is updated from 4 to 5, and the maximal lower bound of n_a is still 0; hence, $mu'(a) - ml'(a) = mu(a) - ml(a) + 1$, and this also holds true for n_b,n_c, and n_d. Furthermore, for n_{ab} in Fig. 2, its original absolute support is $\Lambda(ab) \leqslant \begin{cases} u_a(ab) = 4 \\ u_b(ab) = 3 \end{cases} \Rightarrow \Lambda(ab) \leqslant 3$ and $\Lambda(ab) \geqslant \begin{cases} l_{ab}(ab) = 0 \\ l_\phi(ab) = 3 \end{cases} \Rightarrow \Lambda(ab) \geqslant 3$, i.e., $\Lambda(ab) \in [3,3]$. After *bd* is added, $N_{\text{new}} = ab \cap bd = b \subset ab$; according to Eq. (2), $u'_b(ab) = u_b(ab) + 1$, so its updated absolute support is $\Lambda'(ab) \leqslant \begin{cases} u'_a(ab) = 4 \\ u'_b(ab) = 4 \end{cases} \Rightarrow \Lambda'(ab) \leqslant 4$ and $\Lambda'(ab) \geqslant \begin{cases} l'_{ab}(ab) = 0 \\ l'_\phi(ab) = 3 \end{cases} \Rightarrow \Lambda'(ab) \geqslant 3$, i.e., $\Lambda'(ab) \in [3,4]$, so n_{ab} changes from derivable to non-derivable. Similarly, n_{ac} becomes non-derivable and n_{bc} is still non-derivable. As can be seen, after *bd* is added for each node n_I, $mu(I) - ml(I)$ is not decreased, and it is increased by at most 1.

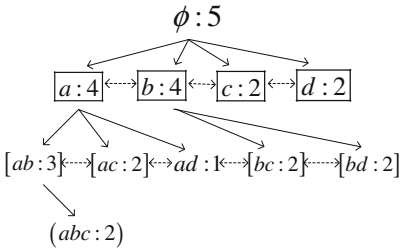


Fig. 5. *NDFIT* with *bd* being added to window 1.

3.4.2. Deleting an existing transaction

Theorem 7. *Given a deleted transaction T and node n_I, $N_{\text{old}} = T \cap I$ and $X \subset I$. If $l_X(I)$ and $u_X(I)$ denote the original lower and upper bounds of I, and $l'_X(I)$ and $u'_X(I)$ denote the updated ones, then the following conclusions hold:*

if $N_{\text{old}} \subset I$ and $N_{\text{old}} = X$, then

$$\begin{cases} l'_X(I) = l_X(I) + 1 & |I \setminus X| \quad \text{is even} \\ u'_X(I) = u_X(I) - 1 & |I \setminus X| \quad \text{is odd} \end{cases} \tag{5}$$

if $N_{\text{old}} \subset I$ and $N_{\text{old}} \neq X$, then

$$\begin{cases} l'_X(I) = l_X(I) & |I \setminus X| \quad \text{is even} \\ u'_X(I) = u_X(I) & |I \setminus X| \quad \text{is odd} \end{cases} \tag{6}$$

if $N_{\text{old}} = I$, then

$$l'_X(I) = l_X(I) - 1 \quad \text{and} \quad u'_X(I) = u_X(I) - 1. \tag{7}$$

Proof. Be similar to that of Theorem 5. □

Lemma 8. *For a node n_I, deleting an existing transaction will not increase $mu(I) - ml(I)$, and it will decrease $mu(I) - ml(I)$ by at most 1.*

Proof. For each $X \subset I$, we use $l_X(I)$ and $u_X(I)$ to denote the original lower and upper bounds of I, and use $l'_X(I)$ and $u'_X(I)$ to denote the updated ones. Similarly, we use $ml(I)$ and $mu(I)$ to denote the original maximal lower and minimal upper bounds, and use $ml'(I)$ and $mu'(I)$ to denote the updated bounds. When an existing transaction T is deleted, $N_{\text{old}} = I \cap T$, then $N_{\text{old}} = I$ and $N_{\text{old}} \subset I$ will not happen at the same time, so we discuss these two aspects separately.

If $N_{\text{old}} = I$, for each X, $l_X(I)$ and $u_X(I)$ are decreased by 1 according to Eq. (7); thus, $ml(I)$ and $mu(I)$ are both decreased by 1, i.e., $mu'(I) - ml'(I) = mu(I) - ml(I)$, so in this condition, $mu(I) - ml(I)$ is unchanged.

If $N_{\text{old}} \subset I$, since N_{old} is a fixed itemset, for each X, only one X value x satisfies $x = N_{\text{old}}$, so only one of the formulas in Eq. (5) holds; without a loss of generality, we suppose $|I \setminus x|$ is even, in which case $l'_x(I) = l_x(I) + 1$ holds; if $l_x(I) = ml(I)$, then $ml'(I) = ml(I) + 1$, otherwise, $ml'(I) = ml(I)$; furthermore, for the other values of X, i.e., $X \neq x = N_{\text{old}}$, then $l_X(I)$ and $u_X(I)$ keep their original values according to Eq. (6), and thus $ml(I)$ and $mu(I)$ are not changed for these X. To summarize, for all X, $mu(I)$ will keep its value, and $ml(I)$ will be increased by 0 or 1, as a result, $mu(I) - ml(I)$ will be decreased by at most 1. The same conclusion can be obtained when $|I \setminus x|$ is odd. □

Example 4. Fig. 6 shows the *NDFIT* after transaction a is deleted from window 1. Since $|D|$ backs to 4, the minimal upper bound of n_a becomes from 5 to 4, and the maximal lower bound of n_a is still 0; hence, $mu'(a) - ml'(a) = mu(a) - ml(a) - 1$, and so do n_b,n_c, and n_d. Furthermore, for n_{bc} in Fig. 5, its original absolute support is $\Lambda(bc) \leqslant \begin{cases} u_b(bc) = 4 \\ u_c(bc) = 2 \end{cases} \Rightarrow \Lambda(bc) \leqslant 2$ and $\Lambda(bc) \geqslant \begin{cases} l_{bc}(bc) = 0 \\ l_\phi(bc) = 1 \end{cases} \Rightarrow \Lambda(bc) \geqslant 1$, i.e., $\Lambda(bc) \in [1,2]$. After a is deleted, $N_{\text{old}} = bc \cap a = \phi \subset bc$, according to Eq. (5), $l'_\phi(bc) = l_\phi(bc) + 1$, so its updated absolute support is $\Lambda'(bc) \leqslant \begin{cases} u'_b(bc) = 4 \\ u'_c(bc) = 2 \end{cases} \Rightarrow \Lambda'(bc) \leqslant 2$ and $\Lambda'(bc) \geqslant \begin{cases} l'_{bc}(bc) = 0 \\ l'_\phi(bc) = 2 \end{cases} \Rightarrow \Lambda'(bc) \geqslant 2$, i.e., $\Lambda'(bc) \in [2,2]$, so n_{bc} changes from non-derivable to derivable. Similarly, n_{ab} becomes derivable and n_{ac} is still non-derivable. As a result, after a is deleted, for each node n_I, $mu(I) - ml(I)$ is not increased, and it is decreased by at most 1.

3.5. Node properties

A node n_I becomes non-derivable once $mu(I) - ml(I) > 0$; and it becomes derivable when $mu(I) - ml(I) = 0$. However, from Lemmas 6 and 8, each transaction addition or deletion changes $mu(I) - ml(I)$ by at most 1. That means most of the nodes will not change their types, which enables us to save a large amount of computation.

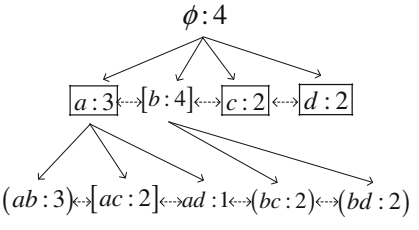


Fig. 6. *NDFIT* with a being deleted from window 1.

Lemma 9. *Both transaction addition and deletion will not change the steady nodes from non-derivable to derivable.*

Proof. Given a steady node n_I, it has at least one non-derivable child n_J, because $mu(J) - ml(J) > 0$, i.e., the minimal value of $mu(J) - ml(J)$ is 1, according to Lemma 2, and $mu(I) - ml(I) \geqslant 2 \times (mu(J) - ml(J)) \Rightarrow mu(I) - ml(I) \geqslant 2$. After a transaction is added, $mu(I) - ml(I)$ will not be decreased according to Lemma 6, i.e., $mu(I) - ml(I) \geqslant 2$ holds; thus, n_I is still non-derivable. Similarly, after a transaction is deleted, $mu(I) - ml(I)$ will be decreased by at most 1 according to Lemma 8, so $mu(I) - ml(I) \geqslant 1$ holds; thus, n_I is still non-derivable. □

Lemma 10. *Adding a transaction will not change the intermediate nodes from non-derivable to derivable, and it may only change the promising nodes from derivable to non-derivable.*

Proof. According to Lemma 6, after a transaction is added, for a node n_I, $mu(I) - ml(I)$ will not be decreased. If n_I is intermediate, i.e., $mu(I) - ml(I) > 0$, then the updated $mu(I) - ml(I)$ is still larger than 0, so n_I is still non-derivable. On the other hand, if n_I is promising, i.e., $mu(I) - ml(I) = 0$, since $mu(I) - ml(I)$ will be increased by at most 1, then $mu(I) - ml(I)$ may be larger than 0, so n_I may become non-derivable. □

Lemma 11. *Deleting a transaction will not change the promising nodes from derivable to non-derivable, and it may only change the intermediate nodes from non-derivable to derivable.*

Proof. According to Lemma 8, after a transaction is deleted, for a node n_I, $mu(I) - ml(I)$ will not be increased. If n_I is promising, i.e., $mu(I) - ml(I) = 0$, then the updated $mu(I) - ml(I)$ is still 0, so n_I is still derivable. On the other hand, if n_I is intermediate, it has no non-derivable child, i.e., $mu(I) - ml(I) \geqslant 1$, since $mu(I) - ml(I)$ will be decreased by at most 1, then $mu(I) - ml(I)$ may become 0, so n_I may become derivable. □

Lemma 9 enables us to ignore the bounds computation of steady nodes; Lemmas 10 and 11 enable us to only compute the bounds of promising nodes when a new transaction is added, and to only compute the bounds of intermediate nodes when an existing transaction is deleted.

3.6. Computational optimization

From Examples 3 and 4 we intuitively find that although a node n_I has no intersection with the added and deleted transaction, the bound computation based on ϕ may change the lower or upper bound when $|D|$ is briefly changed. Thus, it may change the node type. In fact, however, after the window finishes one sliding step, $|D|$ returns to its original value; hence, the computation based on ϕ for n_I and the operations based on the node type change are all redundant. Therefore, if we consider combining transaction addition and deletion, the nodes which are not relevant to the added and deleted transaction will be ignored in processing. Furthermore, we extend this idea and obtain the following conclusion.

Lemma 12. *Given a node n_I, an added transaction T and a deleted transaction S, if $I \cap T = I \cap S$, the bounds computation of n_I can be ignored.*

Proof. The type of n_I is determined by $mu(I)$ and $ml(I)$, which are computed by the support of its subsets. If $I \cap T = I \cap S$, i.e., all of the ancestors of n_I including ϕ do not change their support; hence, n_I will keep its original bounds, and thus its bounds computation can be ignored. □

3.7. NDFIoDS algorithm

Algorithm 1 shows the pseudo code of the *NDFIoDS* algorithm. Algorithm 2 is the exploration process, and is called when new frequent nodes are generated.

Based on the properties of different node types, various operations are performed after the window moves on at each step. First of all, the supports of the relevant nodes are updated.

If a node is a steady node, according to Lemma 9, it is still non-derivable; also, if the intersection between a node and the added transaction is equal to the one between it and the deleted transaction, according to Lemma 12, its bounds are unchanged. Consequently, we will ignore its bounds computation in these conditions.

If an infrequent node changes to frequent, we will generate its descendants and decide their types. In Fig. 5, the support of n_d is updated from 1 to 2; thus, n_d changes to frequent and its children n_{ad} and n_{bd} are generated; since n_{bd} is non-derivable, n_d is remarked the steady node. On the contrary, if a frequent node turns to infrequent, we will prune all of its descendants.

If a node is a promising node, it may change from derivable to non-derivable when a new transaction is added according to Lemma 10; thus, we will compute its bounds to determine whether it changes into an intermediate node; if yes, then we will check its siblings to decide if new children will be generated, and all of its parents become steady nodes. In Fig. 5, since n_{ab} and n_{ac} change to non-derivable, n_a is identified the steady node, and n_{abc}, a derivable node, is generated as the child of n_{ab}, n_{ac}, and n_{bc}.

If a node is an intermediate node, it may change from non-derivable to derivable when an existing transaction is deleted according to Lemma 11; thus, we will compute its bounds to decide whether it turns into a promising node; if yes, then all of its children are pruned, and all of its parents are rechecked to decide whether they become intermediate. In Fig. 6, n_{ab} and n_{bc} are changed from non-derivable to derivable, so n_b is considered the intermediate node, and n_{abc} is pruned.

Algorithm 1. NDFIoDS function

Require: n_I: *NDFIT* node; S: deleted transaction; T: added transaction; D: sliding window; λ: minimum support;
1: **if** $I \cap T \neq I \cap S$ **then**
2: update n_I's support;
3: **if** n_I is not the steady node **then**
4: **if** n_I is new frequent **then**
5: CALL Explore(n_I, D, λ);
6: **else if** n_I is new infrequent **then**
7: remark n_I the infrequent node;
8: prune n_I's descendants $n_{I'}$ and update $n_{I'}$'s parents' type;
9: **else**
10: compute $ml(I)$ and $mu(I)$;
11: **if** n_I is intermediate and $ml(I) = mu(I)$ **then**
12: mark n_I the promising node;
13: prune n_I's children $n_{I'}$ and update $n_{I'}$'s parents' type;
14: update n_I's parents' type;
15: **else if** n_I is promising and $ml(I) \neq mu(I)$ **then**
16: generate its children and mark them promising nodes;
17: mark n_I the intermediate node;
18: mark n_I's parents the steady node;
19: **for** each child $n_{I'}$ of n_I **do**
20: CALL NDFIoDS($n_{I'}$, S, T, D, λ);

The runtime cost of each sliding step is determined by the number of newly generated and pruned nodes. If a node n_I will be pruned, nothing will be performed except the type update of n_I parents, whose time complexity is linear with the help of the pointers between the siblings; consequently, the asymptotic time complexity mainly depends on the time complexity of new node generating. If a node n_I will be generated, its supports will be counted and its bounds will be computed. Given the sliding window D, as mentioned before, its bounds computation is exponential with regard to $|I|$, if $|I|$ is very large, the efficiency is low. Nevertheless, we argue that the practical performance is much better. First, the computational cost is not unlimited when $|I|$ increases: from Lemma 3 we can see that if $|I| > \log_2(|D|) + 1$, then I is derivable, that is, even if in the worst case, the time complexity of computing one node bounds is $O(2^{\log_2(|D|)+2}|I|) = O(|D||I|)$, and this worst case almost never happens because of the minimum support. Second, the bounds need not be entirely computed with the help of Lemma 2. Third, since our algorithm only considers the nodes that are necessary for discovering the non-derivable frequent itemsets, the number of nodes need to be computed will be very small. As can be seen, the overall time complexity of the *NDFIoDS* algorithm depends on the sliding window size, the minimum support, the number and the length of *NDFIT* nodes and the data distribution.

Algorithm 2. Explore function

Require n_I: *NDFIT* node;
D: sliding window; λ: minimum support;
1: **if** $n_I.wt < \lambda \times |D|$ **then**
2: mark n_I the infrequent node;
3: **else**
4: compute ml(I) and mu(I);
5: **if** ml(I) = mu(I) **then**
6: mark n_I the promising node;
7: **else**
8: check and generate n_I's children;
9: **for** each child $n_{I'}$ of n_I **do**
10: CALL Explore($n_{I'}$, D, λ);
11: **if** n_I has non-derivable child **then**
12: mark n_I the steady node;
13: **else**
14: mark n_I the intermediate node;

4. Experimental results

We conducted a series of experiments to evaluate the performance of *NDFIoDS*. Three algorithms were used as the evaluation methods: The state-of-the-art mining algorithm, *dfNDI* [8], is used to generate the non-derivable frequent itemsets for naive incremental mining; the stream mining algorithm, *MOMENT* [18], is used to mine the closed itemsets; and the traditional mining algorithm, *AFOPT* [31], is used to generate the frequent itemsets.

4.1. Running environment and datasets

All experiments were implemented with *C♯*, compiled with *Visual Studio 2005* running on *Windows Server 2003* and executed on a *Xeon 2.0GHz* PC with *2GB* RAM.

We used 2 synthetic datasets and 4 real-life datasets, which are well-known benchmarks for frequent itemset mining. The *T10I4D100K* and *T40I10D100K* datasets are generated with the IBM synthetic data generator. The *BMS-POS* and *BMS-Webview-1* datasets are used for *KDDCUP 2000*; the former contains several years of point-of-sale data and the latter records months of click-stream data from an e-commerce web site. The *KOSARAK* dataset contains the click-stream data of a Hungarian online news portal. The *MUSHROOM* dataset contains characteristics from different species of mushrooms. The data characteristics are summarized in Table 2.

For stream mining, we evaluated the performance over n consecutive sliding windows, where n equals the size of the sliding window. To reflect the entire dataset feature, we randomly selected n transactions from the rest of the data in each dataset, i.e., n randomly selected transactions are added into the sliding window, and the initial n transactions are deleted. For example, in the *BMS-POS* dataset, we chose the first 50,000 transactions as the initial window, then we randomly selected 50,000 transactions from the rest of the 515,597 − 50,000 = 465,597 transactions to add into the window.

4.2. Running time cost evaluation

To evaluate the running time cost, we compared our method with the closed itemset mining algorithm *MOMENT*, as well as the naive non-derivable itemset mining method, which calls the static mining algorithm *dfNDI* once the sliding window is updated.

We first compared the average runtime of these three algorithms under different window sizes when the relative minimum support was fixed. As shown in all of the images marked (a) in Figs. 7–12, the running time cost of *NDFIoDS* and *MOMENT* are stable; these results verify that neither algorithm is sensitive to window size. This is clearly due to their incremental updating fashion; in addition, the use of relative minimum support results in growing absolute minimum support when the window size increases – in turn this causes the number of frequent nodes to decrease. Therefore, although the increasing window size increases the scanning time, the runtime as a whole does not change significantly; in comparison, even though the naive algorithm also uses the relative minimum support, it has to be performed for each window update; thus, the runtime cost grows when the sliding window size increases.

MOMENT divides the data into four categories; when infrequent nodes become frequent, and unpromising nodes become promising, it calls the *explore* function to generate all of the descendants. *NDFIoDS* also classifies the data into four types, and it also calls the *explore* function to generate descendants when the infrequent nodes become frequent. However, in *NDFIoDS*, when the promising nodes become intermediate, their children must be promising or infrequent nodes, i.e., only one tree level will be extended, that is to say, we need not call the *explore* function any more in this situation, and thus the performance is improved.

All of the images marked (b) in Figs. 7–12 represent plots of the average running time cost for consecutive sliding windows under different relative minimum support. It is clear that *NDFIoDS* outperforms *MOMENT* except in the most-dense *MUSHROOM* dataset at low minimum support levels. That is because data in a dense dataset is highly correlated, and a large number of long itemsets are generated when the minimum support is low; on the other hand, deciding whether an itemset is non-derivable requires computation according to all of its subsets. As a result, even though we use *NDFIT* to speed up the itemset search, the time complexity is exponential with regard to the length of the frequent itemsets; consequently, *NDFIoDS* performs worse in this condition.

Table 2
Data characteristics.

DataSet	No. of trans.	Avg. trans. length	Min. trans. length	Max. trans. length	No. of items	Trans. corr.
T10I4D100K	100,000	10.1	1	29	870	86.1
T40I10D100K	100,000	39.6	4	77	942	23.8
BMS-POS	515,597	6.5	1	164	1657	254.9
BMS-Webview-1	59,601	2.5	1	267	497	198.8
KOSARAK	990,002	7.1	1	2497	36,841	5188.8
MUSHROOM	8124	23	23	23	119	5.2

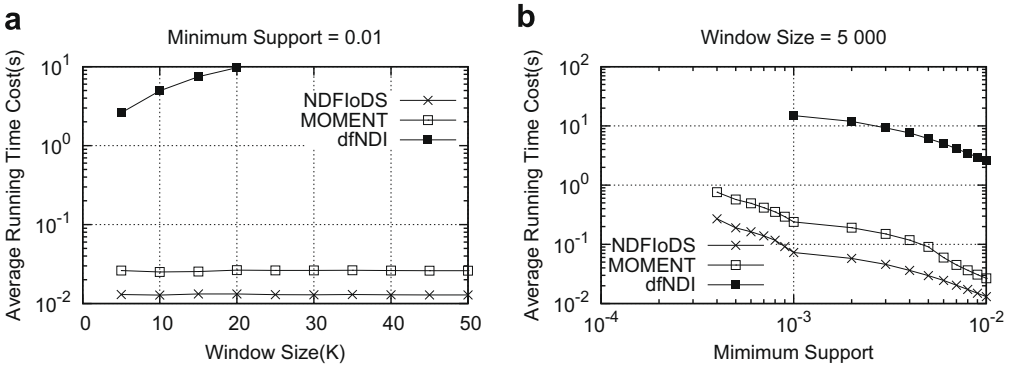


Fig. 7. T10I4D100K: (a) runtime cost vs. number of records; (b) runtime cost vs. minimum support.

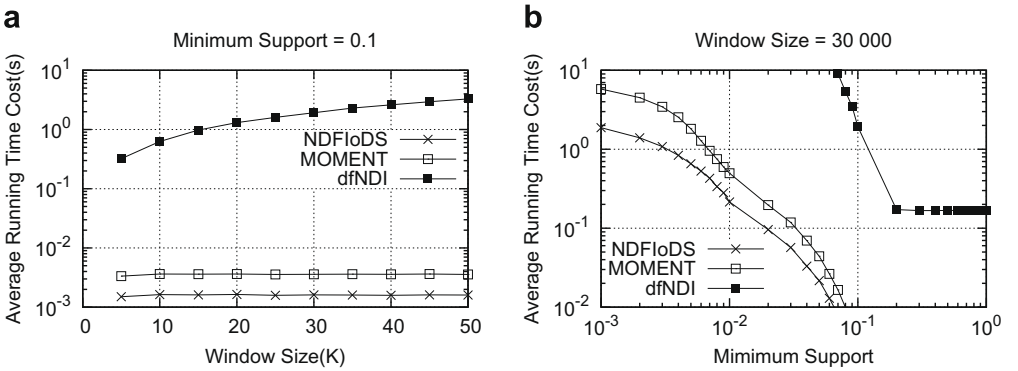


Fig. 8. T40I10D100K: (a) runtime cost vs. number of records; (b) runtime cost vs. minimum support.

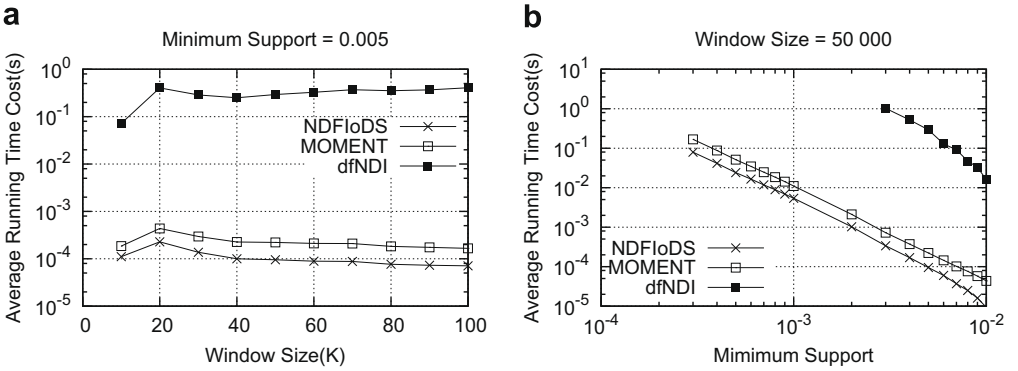


Fig. 9. BMS-POS: (a) runtime cost vs. number of records; (b) runtime cost vs. minimum support.

Our method is not as efficient as *dfNDI* when there are no incremental updates. First, *NDFIoDS* is implemented in a breadth-first fashion, that is, in such a case, it has a similar asymptotic time complexity to the *NDI* algorithm [7]. Second, *NDFIoDS* has to maintain and update data structures in memory, which also requires extra cost. On the other hand, *NDFIoDS* is an incremental algorithm, and it is adept in processing dynamic datasets. In comparison, *dfNDI* has to be performed repeatedly once the dataset has changed because it does not maintain the data synopsis to handle the incremental situation. Consequently, *NDFIoDS* is much more efficient in stream scenarios, as shown in the figures, it achieves speedups of hundreds-fold over the naive method.

In a dataset, given the distinct item number μ and the average transaction length v, an item will occur once in at most $\frac{\mu}{v}$ transactions on average, and we can thus demonstrate the approximate correlation among transactions with $\frac{\mu}{v}$. The smaller

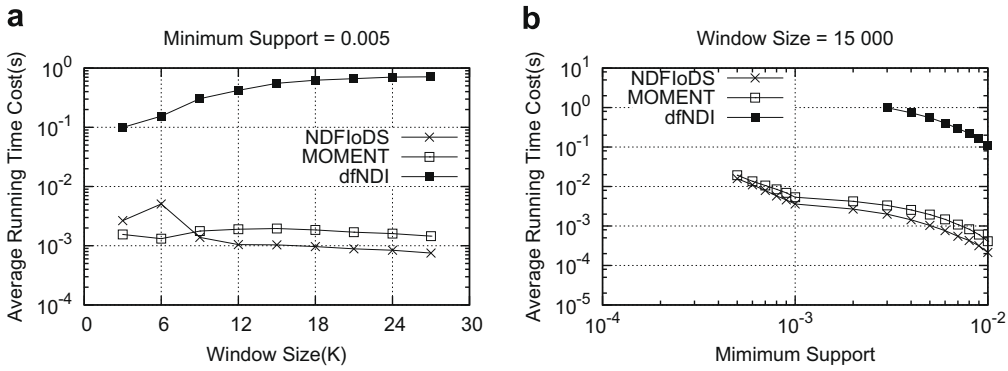


Fig. 10. BMS-Webview-1: (a) runtime cost vs. number of records; (b) runtime cost vs. minimum support.

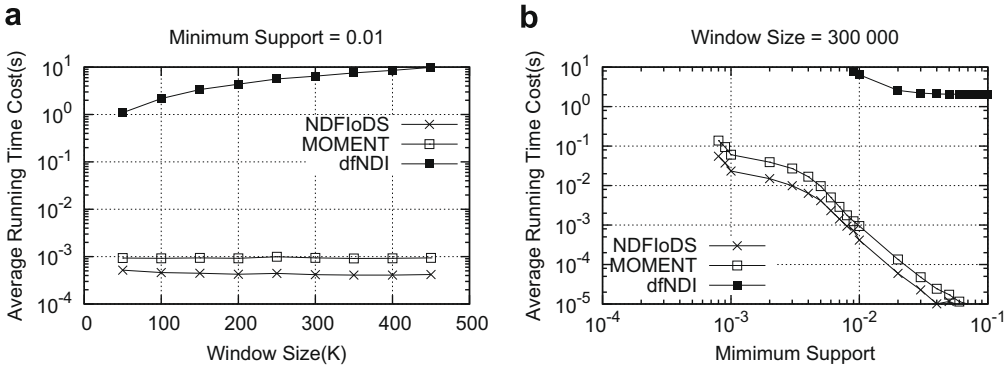


Fig. 11. KOSARAK: (a) runtime cost vs. number of records; (b) runtime cost vs. minimum support.

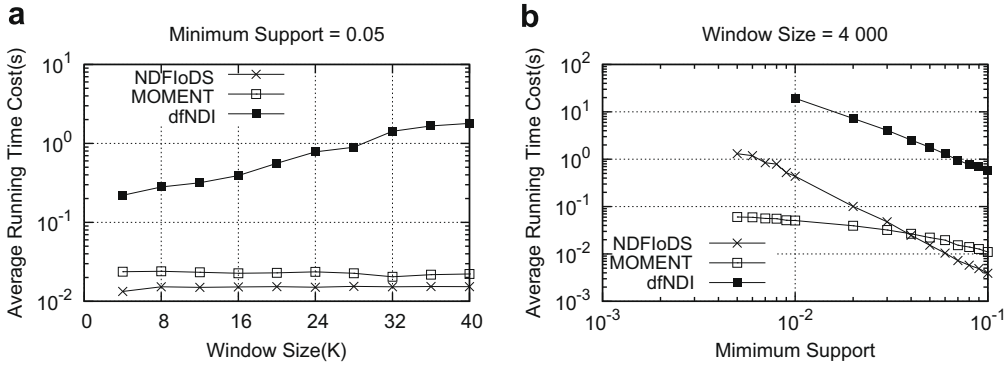


Fig. 12. MUSHROOM: (a) runtime cost vs. number of records; (b) runtime cost vs. minimum support.

the value of $\frac{\mu}{v}$, the more highly correlated the transactions. As shown in the last column of Table 2, data in *MUSHROOM* is the most highly correlated, that is, *MUSHROOM* is the most dense of the six datasets; in contrast, *KOSARAK* is the most sparse dataset. As we can see from all of the images marked (b) in Figs. 7–12, given same minimum support, the running time cost of our algorithm nearly correlates with the density of each dataset, i.e., *NDFIoDS* performs better when datasets become sparse. These results imply that data density has an important effect on the runtime cost of *NDFIoDS*. Nevertheless, it should be noted that data density is not the only factor; as mentioned before, the data distribution and the minimum support also have an effect on the performance of *NDFIoDS*. As an example, although *KOSARAK* is the most sparse dataset, *NDFIoDS* performs worse than expected.

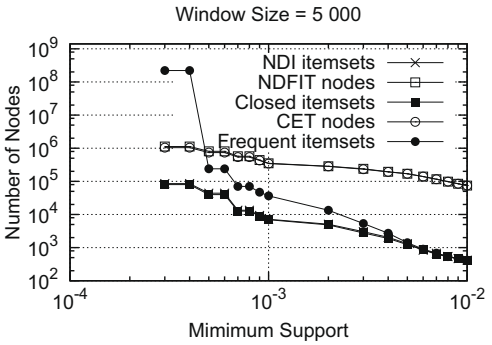


Fig. 13. Itemsets number of T10I4D100K.

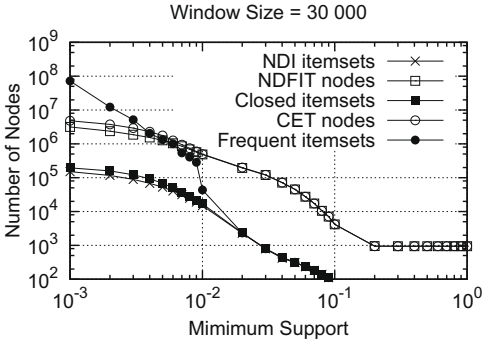


Fig. 14. Itemsets number of T40I10D100K.

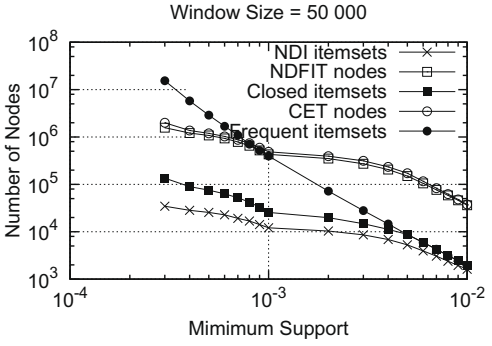


Fig. 15. Itemsets number of BMS-POS.

4.3. Itemsets number comparison

Non-derivable frequent itemsets are one of the condensed representations of frequent itemsets, with a size much smaller than that of all frequent itemsets. The target of mining non-derivable frequent itemsets is to give users more concise results, and to save on space cost when resources are limited. In generating non-derivable frequent itemsets over streams, however, we have to maintain certain infrequent and derivable itemsets, which raises the space cost. To evaluate the space cost performance, we present plots of the number of non-derivable frequent itemsets, the number of *NDFIT* nodes, the number of closed frequent itemsets, the number of *CET* nodes generated by *MOMENT*, and all of the frequent itemsets generated by *AFOPT* in Figs. 13–18.

Generally, the number of closed itemsets is larger than that of non-derivable itemsets, which can be verified with the figures. As can be seen, non-derivable frequent itemsets are less than closed frequent itemsets in the *T10I4D100K* and *T40I10D100K* synthetic datasets and the *BMS-POS* and *MUSHROOM* real-life datasets, although in the *BMS-Webview-1* and *KOSARAK* datasets, closed itemsets are less than non-derivable itemsets.

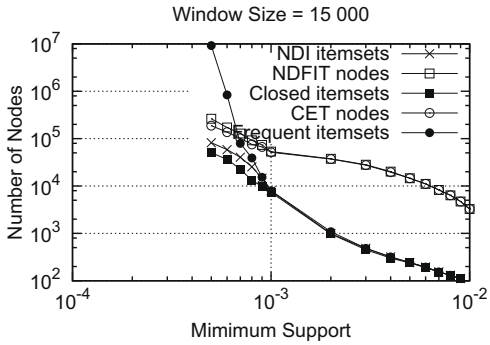


Fig. 16. Itemsets number of BMS-Webview-1.

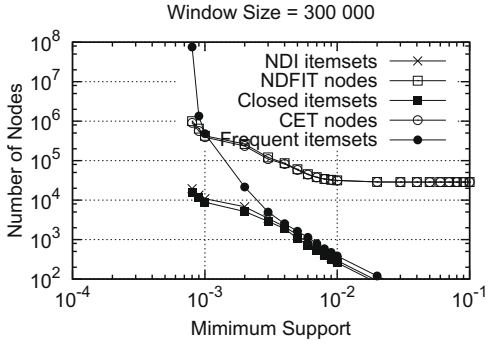


Fig. 17. Itemsets number of KOSARAK.

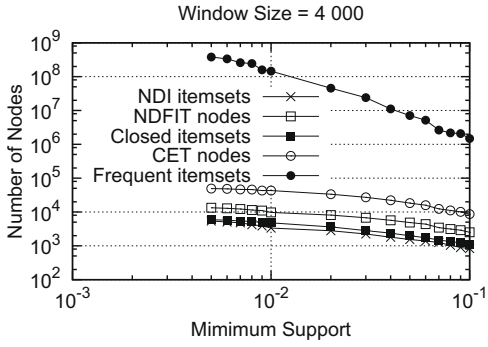


Fig. 18. Itemsets number of MUSHROOM.

In addition, other than the necessary infrequent and unpromising nodes in *CET*, *MOMENT* has to store intermediate nodes to maintain monotonicity, i.e., the redundant nodes exist not only at leaves, but also at internal nodes, which is less efficient than *NDFIoDS*. As shown in the figures, except in the *BMS-Webview-1* dataset, *NDFIT* nodes are less than *CET* nodes. In *MUSHROOM* in particular, *NDFIT* nodes are much less than *CET* nodes.

Moreover, the frequent itemsets are less than *NDFIT* nodes when the minimum support is high, and they become greater when the minimum support becomes low. In addition, in the *MUSHROOM* dataset, frequent itemsets have much higher numbers despite the minimum support.

5. Conclusions and future work

5.1. Conclusions

In this paper, we considered non-derivable frequent itemset mining over the sliding window of a stream, which, to the best of our knowledge, has not been looked at before. A compact structure named *NDFIT* was introduced to maintain the data

synopsis in memory. In *NDFIT*, we divided nodes into various categories; based upon this division, a large volume of computation was saved as a result of the theoretical proof of some of the node properties. Thus, an efficient algorithm named *NDFIoDS* was developed to incrementally update the *NDFIT* when newly arrived transactions changed the content of the sliding window. Furthermore, we discussed computational optimization opportunities for better performance. Our experimental studies showed that *NDFIoDS* not only outperforms the state-of-the-art non-derivable frequent itemset mining algorithm, but it is also more efficient than the closed frequent itemset mining algorithm.

5.2. Future work

Non-derivable itemset incremental mining is still a new problem, and during our work, we found that there are many important directions that are in need of further study.

5.2.1. Algorithm optimization

As mentioned before, *NDFIoDS* is a breadth-first algorithm, which is generally less efficient than a depth-first method. Consequently, how to generate non-derivable frequent itemsets over stream with a depth-first implementation is an interesting and challenging topic for future work. Moreover, developing new properties of non-derivable itemsets to achieve earlier pruning is another optimization problem.

5.2.2. Variable window size

In our method, we use relative minimum support to generate the results. That is to say, we have to fix the window size after the algorithm starts, as any window shrinking or growing may change the node types and will require a large adjustment of the *NDFIT*, which is not efficient. We will handle this problem in our future work.

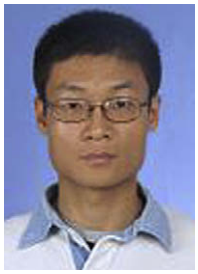
5.2.3. Approximate method

As shown in the experimental results, our method is not efficient when performed on dense datasets at low relative minimum support. In [10], the rules of limited depth are proved with good bounds and better performance, which can also be considered a part of our method that needs for further optimization. How to decide the depth to achieve the best efficiency with limited computational resources is one of our future areas of study.

References

[1] T. Asai, H. Arimura, K. Abe, S. Kawasoe, S. Arikawa, Online algorithms for mining semi-structured data stream, in: Proceedings of the ICDM'2002.
[2] F. Afrati, A. Gionis, H. Mannila, Approximating a collection of frequent sets, in: Proceedings of the SIGKDD'2004.
[3] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, in: Proceedings of the VLDB'1994.
[4] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the SIGMOD'2002.
[5] J. Boulicaut, A. Bykowski, C. Rigotti, Free-sets: a condensed representation of boolean data for the approximation of frequency queries, Data Mining and Knowledge Discovery 7 (2003) 5–22.
[6] T. Calders, N. Dexters, B. Goethals, Mining frequent itemsets in a stream, in: Proceedings of the ICDM'2007.
[7] T. Calders, B. Goethals, Mining all non-derivable frequent itemsets, in: Proceedings of the PKDD'2002.
[8] T. Calders, B. Goethals, Depth-first non-derivable itemset mining, in: Proceedings of the SDM'2005.
[9] T. Calders, B. Goethals, Quick inclusion–exclusion, in: Workshop KDID of PKDD'2005.
[10] T. Calders, B. Goethals, Non-derivable itemset mining, Data Mining and Knowledge Discovery 14 (1) (2007) 71–206.
[11] T. Calders, B. Goethals, Minimal *k*-free representations of frequent sets, in: Proceedings of the PKDD'2003.
[12] J.H. Chang, W.S. Lee, Decaying obsolete information in finding recent frequent itemsets over data stream, IEICE Transaction on Information and Systems 87 (6) (2004) 1588–1592.
[13] J.H. Chang, W.S. Lee, A sliding window method for finding recently frequent itemsets over online data streams, Journal of Information Science and Engineering 20 (4) (2004) 753–762.
[14] J.H. Chang, W.S. Lee, Finding recent frequent itemsets adaptively over online data streams, in: Proceedings of the SIGKDD'2003.
[15] G. Cormode, F. Korn, S. Muthukrishnan, Divesh Srivastava, Finding hierarchical heavy hitters in data streams, in: Proceedings of the VLDB'2003.
[16] J. Cheng, Y. Ke, W. Ng, A survey on algorithms for mining frequent itemsets over data streams, Knowledge and Information Systems 16 (1) (2006) 1–27.
[17] G. Cormode, S. Muthukrishnan, What's hot and what's not: tracking most frequent items dynamically, in: Proceedings of the PODS'2003.
[18] Y. Chi, H. Wang, P.S. Yu, R.R. Muntz, Moment: maintaining closed frequent itemsets over a stream sliding window, in: Proceedings of the ICDM'2004.
[19] C. Giannella, J. Han, J. Pei, X. Yan, P. Yu, Mining frequent patterns in data streams at multiple time granularities, in: Proceedings of the AAAI/MIT'2003.
[20] B. Goethals, J. Muhonen, H. Toivonen, Mining non-derivable association rules, in: Proceedings of the SDM'2005.
[21] J. Galambos, I. Simonelli, Bonferroni-type Inequalities with Applications, Springer, 1996.
[22] J. Han, H. Cheng, D. Xin, X. Yan, Frequent pattern mining: current status and future directions, Data Mining and Knowledge Discovery 17 (2007) 55–86.
[23] J. Han, J. Pei, Mining frequent patterns by pattern-growth: methodology and implications, in: Proceedings of the SIGKDD'2000.
[24] R. Jin, G. Agrawal, An algorithm for in-core frequent itemset mining on streaming data, in: Proceedings of the ICDM'2005.
[25] N. Jiang, L. Gruenwald, CFI-stream: mining closed frequent itemsets in data streams, in: Proceedings of the SIGKDD'2006.
[26] C. Jin, W. Qian, C. Sha, J.X. Yu, A. Zhou, Dynamically maintaining frequent items over a data stream, in: Proceedings of the CIKM'2003.
[27] S. Kevin, R. Ramakrishnan, Bottom-up computation of sparse and iceberg CUBEs, in: Proceedings of the SIGMOD'1999.
[28] J. Koh, S. Shin, An approximate approach for mining recently frequent itemsets from data streams, in: Proceedings of the DaWak'2006.
[29] C.K. Leung, Q.I. Khan, DSTree: a tree structure for the mining of frequent sets from data streams, in: Proceedings of the ICDM'2006.
[30] D. Lee, W. Lee, Finding maximal frequent itemsets over online data streams adaptively, in: Proceedings of the ICDM'2005.
[31] G. Liu, H. Lu, J.X. Yu, W. Wang, X. Xiao, AFOPT: an efficient implementation of pattern growth approach, in: Workshop on FIMI'2003.
[32] C. Lucchese, S. Orlando, R. Perego, Fast and memory efficient mining of frequent closed itemsets, IEEE Transactions on Knowledge and Data Engineering 18 (1) (2006) 21–36.
[33] A.J.T. Lee, C.S. Wang, W.Y. Wen, Y.A. Chen, H.W. Wu, An efficient algorithm for mining closed inter-transaction itemsets, Data & Knowledge Engineering 66 (1) (2008) 68–91.
[34] G.S. Manku, R. Motwani, Approximate frequency counts over streaming data, in: Proceedings of the VLDB'2002.

[35] B. Mozafari, H. Thakkar, C. Zaniolo, Verifying and mining frequent patterns from large windows over data streams, in: Proceedings of the ICDE'2008.
[36] G. Mao, X. Wu, X. Zhu, G. Chen, Mining maximal frequent itemsets from data streams, Journal of Information Science 33 (3) (2007) 251–262.
[37] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Discovering frequent closed itemsets for association rules, in: Proceedings of the ICDT'1999.
[38] B. Padmanabhan, A. Tuzhilin, On characterization and discovery of minimal unexpected patterns in rule discovery, IEEE Transactions on Knowledge and Data Engineering 18 (2) (2006) 202–216.
[39] W. Teng, M. Chen, P.S.Yu, A regression-based temporal pattern mining scheme for data streams, in: Proceedings of the VLDB'2003.
[40] F. Tao, F. Murtagh, M. Farid, Weighted association rule mining using weighted support and significance framework, in: Proceedings of the SIGKDD'2003.
[41] H.J. Woo, W.S. Lee, estMax: tracing maximal frequent itemsets over online data streams, in: Proceedings of the ICDM'2007.
[42] D. Xin, J. Han, X. Yan, H. Cheng, On compressing frequent patterns, Data & Knowledge Engineering 60 (1) (2007) 5–29.



Haifeng Li received the B.S. in Computer Science from Shandong University of Science and Technology, China, and the M.S. degree from Beijing University of Chemical Technology in 1996 and 2005, respectively. He is a Ph.D. candidate under the supervision of Prof. Hong Chen, in Renmin University of China. He has published more than 10 research papers in journals and proceedings of international conferences and workshops, mainly in the field of database. His research interests include data stream and data mining.

Hong Chen received the B.S. and M.S. degrees in Computer Science from Rennin University of China, and the Ph.D. degree in Computer Science from Institute of Computing Technology, Chinese Academy of Sciences in 1986, 1989 and 2000, respectively. In 1989, she joined the faculty of the school of Information at Renmin University of China. Since 2002, she is a professor in the Department of Computer Science and Technology at Renmin University of China, and is the director of the Data Warehousing and Business Intelligence research group. She is the (co-)author of more than 100 research papers, mainly in the field of database, and served as PC member in several national and international Conferences. Her research interest include data warehousing and data mining, data management in wireless sensor network and query processing in database system.