Expert Systems With Applications 105 (2018) 129–143

Contents lists available at ScienceDirect

Expert Systems With Applications

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

negFIN: An efficient algorithm for fast mining frequent itemsets

Nader Aryabarzan[a], Behrouz Minaei-Bidgoli[b,*], Mohammad Teshnehlab[c]

[a] Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran
[b] School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran
[c] Faculty of Electrical Engineering, Systems and Control Department, K. N. Toosi University of Technology, Tehran, Iran

ARTICLE INFO

Article history:
Received 3 December 2017
Revised 19 March 2018
Accepted 22 March 2018
Available online 24 March 2018

Keywords:
Prefix tree
Nodesets
Bitmap representation
Frequent itemsets

ABSTRACT

Frequent itemset mining is a basic data mining task and has numerous applications in other data mining tasks. In recent years, some data structures based on sets of nodes in a prefix tree have been presented. These data structures store essential information about frequent itemsets. In this paper, we propose another efficient data structure, *NegNodeset*. Similar to other such data structures, the basis of *NegNodeset* is sets of nodes in a prefix tree. *NegNodeset* employs a novel encoding model for nodes in a prefix tree based on the bitmap representation of sets. Based on the *NegNodeset* data structure, we propose negFIN, which is an efficient algorithm for frequent itemset mining. The efficiency of the negFIN algorithm is confirmed by the following three reasons: (1) the *NegNodesets* of itemsets are extracted using bitwise operators, (2) the complexity of calculating *NegNodesets* and counting supports is reduced to $O(n)$, where n is the cardinality of *NegNodeset*, and (3) it employs a set-enumeration tree to generate frequent itemsets and uses a promotion method to prune the search space in this tree. Our extensive performance study on a variety of benchmark datasets indicates that negFIN is the fastest algorithm, compared with previous state-of-the-art algorithms. However, our algorithm runs with the same speed as dFIN on some datasets.

1. Introduction

"Frequent itemset mining" is one of the important data mining tasks and has numerous applications in other data mining tasks, such as the discovery of association rules (Ceglar & Roddick, 2006), clustering (Wang, Wang, Yang, & Yu, 2002), and classification (Cheng, Yan, Han, & Yu, 2008). The original use of this task was for market basket analysis and was first proposed in (Agrawal, Imieliński, & Swami, 1993). It aims to find items in the customer transactions database that are frequently bought together.

1.1. Problem definition

Let $I=\{i_1,i_2,\ldots, i_{nit}\}$ be the set of all items in the transactional database; a transaction T be a set of some items ($T\subseteq I$), with a unique identifier *TID*; and a database $DB=\{T_1,T_2,\ldots, T_{nt}\}$ be the set of transactions. Each P where $P\subseteq I$ is called an "itemset." P is also called a k-itemset, where $|P|=k$. A transaction T contains an itemset P if and only if $P\subseteq T$; the support of P, which is denoted as *support*(P), is defined as the percentage of transactions in *DB* containing P. Let $min-support$ be the user-defined minimum support threshold. P is called a frequent itemset if and only if $min-support \leq support(P)$. Given the database *DB* and the $min-support$ threshold, the frequent itemset mining task is defined as "discovering all frequent itemsets with their supports." The number of itemsets that have to be checked to discover frequent itemsets is 2^{nit}, where $nit=|I|$. Therefore, the problem of discovering frequent itemsets is NP.

1.2. Motivation and contribution

Frequent itemset mining has been a hot research topic in the data mining field for the last two decades (Aliberti, Colantonio, Di Pietro, & Mariani, 2015; Calders, Dexters, Gillis, & Goethals, 2014; Deng, 2014; Deng, Gao, & Xu, 2011; Lan, Hong, Lin, & Wang, 2015; Lin, Hong, & Lin, 2015; Troiano & Scibelli, 2014; Vo, Le, Hong, & Le, 2015). In recent years, four types of data structures based on the sets of nodes in a "prefix tree" have been presented to enhance the efficiency of mining frequent itemsets. They are: (1) $Node-list$ (Deng & Wang, 2010), (2) $N-list$ (Deng, Wang, & Jiang, 2012), (3) *Nodeset* (Deng & Lv, 2014), and (4) *DiffNodeset* (Deng, 2016). All of these data structures employ a prefix tree with encoded nodes and associate a set of nodes with each itemset. The

* Corresponding author at: University Road, Hengam Street, Resalat Square, Narmak, Tehran 16846-13114, Iran.
E-mail addresses: aryabarzan@aut.ac.ir (N. Aryabarzan), b_minaei@iust.ac.ir (B. Minaei-Bidgoli), teshnehlab@eetd.kntu.ac.ir (M. Teshnehlab).

https://doi.org/10.1016/j.eswa.2018.03.041
0957-4174/

nodes in $Node-list$ and $N-list$ are encoded by the pre-order rank and post-order rank of the node. Two algorithms, PPV (Deng & Wang, 2010) and PrePost (Deng et al., 2012), have been proposed for mining frequent itemsets based on these two data structures, respectively. These two algorithms outperform their predecessors. However, they have a drawback: they consume a lot of memory (Deng & Lv, 2014). To overcome this problem, another data structure, called *Nodeset* (Deng & Lv, 2014), has been proposed. Unlike $N-list$ and $Node-list$, the nodes in a *Nodeset* are encoded only by the pre-order (or post-order) rank of the nodes (Deng & Lv, 2014). The *Nodeset* of each k-itemset ($3 \leq k$) is extracted by the intersection of the *Nodesets* of two (k-1)-itemsets (Deng & Lv, 2014). The FIN algorithm (Deng & Lv, 2014) has been proposed for frequent itemset mining based on this structure. The disadvantage of *Nodeset* is that the *Nodeset* cardinality becomes very large for some datasets (Deng, 2016). To overcome this problem, another data structure, *DiffNodeset* (Deng, 2016), has been proposed. In contrast to *Nodeset*, the *DiffNodeset* of each k-itemset ($3 \leq k$) is extracted by the difference between the *DiffNodesets* of two (k-1)-itemsets (Deng, 2016). Extensive experiments show that the cardinality of *DiffNodeset* is smaller than that of *Nodeset* (Deng, 2016). The dFIN algorithm (Deng, 2016) has been proposed for mining frequent itemsets based on the *DiffNodeset* data structure. Experimental results show that the dFIN algorithm is faster than its predecessors (Deng, 2016).

Despite the advantages of *DiffNodeset*, we find that calculating the difference between two *DiffNodesets* takes a long time on some databases. To overcome this problem, we propose a new data structure, *NegNodeset*, which employs a prefix tree as well as the previous four data structures. Unlike these data structures, *NegNodeset* employs a new encoding model for nodes. The node-encoding model of *NegNodeset* is based on the bitmap representation of sets. Consider a universal set U with cardinality n. We can represent each subset of U by a bitmap of size n. Each element of U is assigned to one of the bits in the bitmap. If an element is a member of a subset S ($S \subseteq U$), then its corresponding bit is 1; otherwise it is 0. Take the following example into account: let there be a universal set $U=\{a_3,a_2,a_1,a_0\}$ and subsets $A=\{a_3,a_2\}$ and $B=\{a_3,a_0\}$. With two bitmaps of size four, in which each a_i ($0 \leq i \leq 3$) is assigned to their ith bit, these subsets are represented as $A=1100$ and $B=1001$. With this representation of sets, some common set operators can be implemented faster using bitwise operators. For example, to calculate the intersection (union) of two given sets, we can use the bitwise operator AND (OR) on their corresponding bitmaps. Bitwise operators are implemented efficiently in CPUs and done in one CPU cycle.

Based on the *NegNodeset* data structure, we propose negFIN, a fast algorithm for mining frequent itemsets. The efficiency of the negFIN algorithm is confirmed by the following three reasons: (1) new *NegNodesets* are extracted using bitwise operators, which are fast; (2) the complexity of extracting new *NegNodesets* and counting their supports is reduced to $O(n)$, instead of $O(m+n)$ in previous algorithms, where m and n are the cardinality of two sets of nodes, and $n \leq m$; and (3) it employs a "set-enumeration tree" (Rymon, 1992) to generate frequent itemsets and uses a promotion method to prune search space in this tree. This pruning strategy generates the frequent itemsets, sometimes directly without candidate generation.

1.3. Performance of negFIN

We conducted several experimental studies to evaluate the performance of the negFIN algorithm. We compared the performance of negFIN against dFIN (Deng, 2016), Goethals's Eclat (Goethals & Zaki, 2004), and FP-growth* (Grahne & Zhu, 2005), which have been the leading algorithms in the field of frequent itemset mining so far. The experimental results show that negFIN has good performance and, compared to the above mentioned algorithms, runs faster or equally fast. It runs faster than Goethals's Eclat (Goethals & Zaki, 2004) and FP-growth* (Grahne & Zhu, 2005) on all datasets. It still runs faster than dFIN (Deng, 2016) on some datasets, but runs as fast as dFIN (Deng, 2016) on other datasets.

1.4. Structure of the paper

The rest of this paper is organized as follows: Section 2 discusses background and related work for frequent itemset mining. Section 3 introduces basic definitions and properties relevant to the *NegNodeset* structure and the negFIN algorithm. Section 4 explains the negFIN algorithm. Section 5 shows experimental results. Section 6 concludes the paper, and section 7 provides some future research directions.

2. Related work

Many algorithms have been proposed to discover all frequent itemsets efficiently. These algorithms are divided into two main categories: (1) algorithms that use the "candidate generation" method, and (2) algorithms that use the "pattern growth" method (Ceglar & Roddick, 2006). In the candidate generation method, the candidate itemsets are generated first, and then frequent itemsets are identified from these candidate itemsets. This method employs an anti-monotone property, called Apriori (Agrawal & Srikant, 1994), to prune search space. The Apriori property explains that if an itemset is not frequent, then none of its super-itemsets are frequent either. Algorithms like (Agrawal & Srikant, 1994; Deng et al., 2011; Savasere, Omiecinski, & Navathe, 1995; Shenoy et al., 2000; Zaki, 2000; Zaki & Gouda, 2003) employ the candidate generation method. The drawback of this method is that it is highly expensive, because it requires multiple database scans.

Unlike the candidate generation method, the pattern growth method does not generate the candidate itemsets and avoids multiple database scans by storing essential information about frequent itemsets into special data structures. The classic and basic algorithm in this category is the FP-growth algorithm (Han, Pei, & Yin, 2000). It stores essential information about frequent itemsets in a tree-based data structure, namely frequent pattern tree (FP-tree). Similar to the FP-growth algorithm, other algorithms, like (Grahne & Zhu, 2005; Jian et al., 2001; Liu, Lu, Lou, Xu, & Yu, 2004), employ the pattern growth method to discover frequent itemsets. Despite the above benefits of the pattern growth method, this method has some weaknesses, which are as follows: (1) it is inefficient on sparse datasets (Deng et al., 2012) and (2) the data structures employed by pattern growth algorithms are complex (Woon, Ng, & Lim, 2004).

In recent years, four types of data structures based on prefix trees have been proposed to store essential information about frequent itemsets, which are as follows: (1) $Node-list$ (Deng & Wang, 2010), (2) $N-list$ (Deng et al., 2012), (3) *Nodeset* (Deng & Lv, 2014), and (4) *DiffNodeset* (Deng, 2016). Both $Node-list$ and $N-list$ are based on a tree structure called PPC-tree (Deng & Wang, 2010; Deng et al., 2012). A PPC-tree is a prefix tree in which each node is encoded by its pre-order rank and post-order rank. The $Node-list$ or $N-list$ of an itemset is a set of nodes in the PPC-tree. $N-list$ has two advantages over $Node-list$: (1) the cardinality of the $N-list$ of an itemset is much smaller than the cardinality of its $Node-list$. (2) $N-list$ employs a property, called the "single path property," to directly discover frequent itemsets without candidate generation in some cases. Two algorithms, PPV (Deng & Wang, 2010) and PrePost (Deng et al., 2012), have been proposed for discovering all frequent itemsets, based on $Node-list$ and $N-list$ respectively. In recent years,

Table 1
A sample transaction database. min – support = 0 · 4.

TID	Items	Ordered frequent items
1	e, b, g, d	b, d, e
2	c, e, b, a	a, b, c, e
3	c, b, a, i	a, b, c
4	a, d, h	a, d
5	a, d, c, b, f	a, b, c, d

(Deng & Lv, 2015; Vo, Coenen, Le, & Hong,) have employed highly efficient pruning techniques to enhance PrePost performance. Despite the mentioned advantages of *Node – list* and *N – list*, these data structures consume a lot of memory, because they need to store both the pre-order and post-order ranks of nodes. To overcome this problem, the *Nodeset* (Deng & Lv, 2014) data structure has been proposed, which only holds one of either the pre-order or the post-order rank of nodes. The *Nodeset* of each k-itemset ($3 \leq k$) is extracted from the intersection of the *Nodeset*s of two (k – 1)-itemsets (Deng & Lv, 2014). The FIN (Deng & Lv, 2014) algorithm has been proposed for discovering frequent itemsets based on the *Nodeset* data structure. Although *Nodeset* is an efficient structure for discovering frequent itemsets, the *Nodeset* cardinality becomes very large on some datasets (Deng, 2016). To overcome this problem, the *DiffNodeset* (Deng, 2016) data structure has been proposed. In contrast to *Nodeset*, the *DiffNodeset* of each k-itemset ($3 \leq k$) is extracted from the difference between the *DiffNodeset*s of two (k – 1)-itemsets (Deng, 2016). The dFIN (Deng, 2016) algorithm has been proposed for discovering frequent itemsets based on the *DiffNodeset* data structure. Extensive experiments show that the dFIN algorithm runs faster than its state-of-the-art predecessors (Deng, 2016).

3. Basic terminologies

In addition, similar data structure named ***PUN – list*** (Deng, 2018) has been proposed to discover "high utility itemsets," a new kind of mining task that is different from frequent itemset mining. In this task, each item has a utility value and can occur more than once in a transaction. A high utility itemset is an itemset that its utility is not less than a given minimum threshold. In addition to storing an information about frequent itemsets, ***PUN – list*** data structure also stores an information about utilities. The MIP (Deng, 2018) algorithm has been proposed for efficiently discovering high utility itemsets based on the ***PUN – list*** data structure. The experimental results show that MIP algorithm is very efficient and runs faster than its state-of-the-art predecessors (Deng, 2018).

In this section, the basic terminologies and properties related to the *NegNodeset* structure and the negFIN algorithm will be introduced. Here, some notations and terminologies are similar to the notations and terminologies that are used in (Deng & Wang, 2010; Deng et al., 2012). Most notations and terminologies are illustrated by examples. These examples are based on Example 1.

Example 1. Consider a sample transaction database, which is shown in Table 1, and $min - support = 0 \bullet 4$. In this table, the first column shows the transaction ID (*TID*), the second column shows the items in each transaction, and the third column shows the frequent items in each transaction, which are sorted in non-ascending order with respect to $support(\alpha)$, where α is an item.

Definition 1. ($\succ$relation). $\forall i_1, i_2 \in F_1$ (The set of frequent items); $i_2 \succ i_1$ if and only if $support(i_2) \geq support(i_1)$. In Example 1, $a \succ b \succ c \succ d \succ e$.

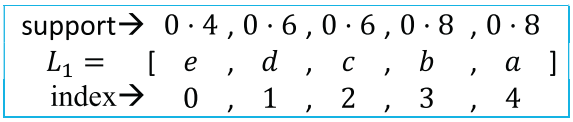

Fig. 1. The zero-based vector L_1, and the index of each frequent item in Example 1.

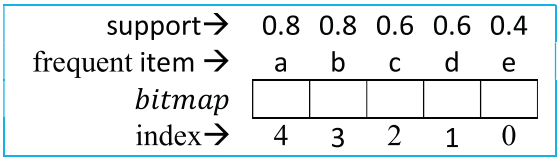

Fig. 2. The bit assigned to each frequent item for Example 1.

Definition 2. (L_1). Given F_1, L_1 is the zero-based vector of ordered frequent items, where items are sorted in non-descending order with respect to $support(\alpha)$, where α is an item.

In this study, L_1 is denoted as $L_1 = [i_0, i_1, \ldots, i_{nf-1}]$, where $nf = |F_1|$ (the abbreviation for **n**umber of **f**requent items), and $i_{nf-1} \succ \ldots \succ i_1 \succ i_0$. Furthermore, a k-itemset P is denoted as $P_k = i_k \ldots i_2 i_1$ or $P_k = i_k P_{k-1}$, where $i_k \succ \ldots \succ i_2 \succ i_1$, and $P_{k-1} = i_{k-1} \ldots i_2 i_1$.

In Example 1, $F_1 = \{e, b, a, c, d\}$, $L_1 = [e, d, c, b, a]$(Fig. 1), and a sample itemset $P = \{e, b, d\}$ is denoted as $P_3 = bde$.

Definition 3. ($\acute{P}_k$). Let $P_k = i_k P_{k-1} (2 \leq k)$. $\acute{P}_k$ is defined as $\acute{P}_k = \neg i_k P_{k-1}$, where $\neg i_k$ means the absence of item i_k.

Definition 4. ($index(item\ i)$). For any item i, $i \in L_1$, $index(i)$ is defined as the index of item i in the zero-based vector L_1. Fig. 1 shows the index of each frequent item in Example 1.

Definition 5. ($BMC(itemset\ P_k)$: the abbreviation for **b**it**m**ap **c**ode of itemset). Each itemset P_k can be represented by a bitmap code $BMC(P_k) = b_{nf-1} \ldots b_1 b_0$ of size nf as follows: the jth item in the zero-based vector L_1 is assigned to the bit b_j in this bitmap. If an item i ($i \in L_1$) is a member of P_k, then its corresponding bit is 1; otherwise it is 0.

The bit assigned to each frequent item for Example 1 is shown in Fig. 2. In Example 1, $BMC(ade) = 10011$.

*NegNodeset*s are based on the BMC-tree. Here, BMC-tree is the abbreviation for **B**it**M**ap **C**oding tree and is defined as follows:

Definition 6. (BMC-tree). A BMC-tree is a kind of tree that:

(1) Its root holds $\emptyset$ (means no item) and has a number of item prefix subtrees, as the children of the root.
(2) Each node in the item prefix subtree holds an item i ($i \in L_1$). If the father of this node represents an item j, then $j \succ i$ (we suppose that $\forall i,\ i \in L_1, \emptyset \succ i$). The portion of the path reaching this node is represented by the itemset $node - path$.
(3) Each node has four fields: $item - name$, $count$, $bitmap - code$, and $children - list$. $item - name$ holds an item i ($i \in L_1$). $count$ holds the number of transactions that contain the itemset $node - path$. $bitmap - code$ holds $BMC(node - path)$ (Definition 5). *children-list* holds all children of this node.

The BMC-tree for Example 1 is shown in Fig. 3.

Definition 7. (The "main section" and "don't-care section" of $BMC(node - path)$). Let a node N hold an item i_1, $N.node - path = i_k \ldots i_2 i_1$, where $i_k \succ \ldots \succ i_2 \succ i_1$, $BMC(node - path) = b_{nf-1}, \ldots, b_1, b_0$, and the item i_1 is assigned to the bit b_m ($m = index(i_1)$). The bits $b_{nf-1}, \ldots, b_m$ are defined as the main section of $BMC(node - path)$, and the bits $b_{m-1}, \ldots, b_1, b_0$ are defined as the don't-care section of $BMC(node - path)$.

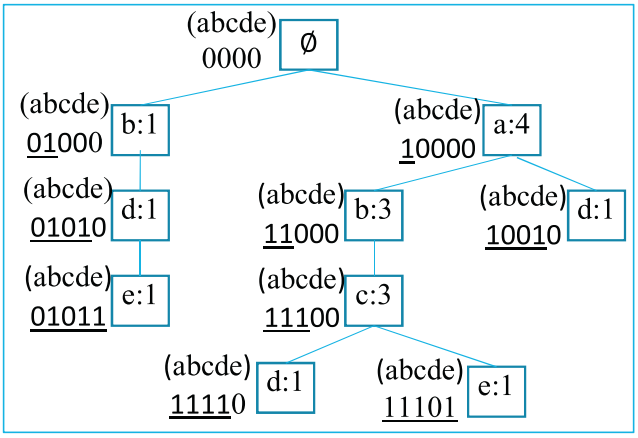

Fig. 3. The BMC-tree for Example 1. Each node label represents the $item-name$ field. The number in each node represents the *count* field. A binary number on the left side of each node represents the $bitmap-code$ field. Items in parentheses represent the item assigned to each bit of $bitmap-code$. The underlined digits in $bitmap-code$ represent the main section, and other bits represent the don't-care section.

For example, see Fig. 3.

Property 1. Bit values in the don't-care section of $BMC(node-path)$ have don't-care values; hence, we can set these bits to 0.

Rationale. Let a node N hold an item i_1, $N.node-path=i_k...i_2i_1$, and $B(node-path)=b_{nf-1}, ..., b_1,b_0$. Each bit b in the don't-care section of $BMC(node-path)$ is assigned to an item like i, where $i_1 \succ i$ (Definition 7). According to the definition of a BMC-tree (Definition 6), such items will or will not be registered in the descendant nodes of N, and we do not have any information about the presence or absence of them. Therefore, the values of these bits are not important. Later, we are going to find out that the don't-care section is useless.□

Property 2. The 0 bit in the main section of $BMC(node-path)$ means that its corresponding item does not exist in $node-path$.

Rationale. Let a node N hold an item i_1, $N.node-path=i_k...i_2i_1$, and $B(node-path)=b_{nf-1}, ..., b_1,b_0$. Each bit b in the main section of $B(node-path)$ is assigned to an item like i, where $i \succ i_1$ (Definition 7). Let $b=0$. We prove by contradiction that $i \notin N.node-path$. Suppose that $i \in N.node-path$. According to the definitions of bitmap codes (Definition 5) and BMC-trees (Definition 6), b must be 1. This contradicts the supposition that $b=0$. Hence, the supposition $i \in N.node-path$ is false, and $i \notin N.node-path$.□

Property 3. All the bits in the $bitmap-code$ of the root of the BMC-tree are 0.

Property 4. Let a node N hold an item i_1, and a node $N.father$ be its father node. $N.bitmap-code = N.father.bitmap-code \vee 2^{index(i_1)}$.

Rationale. Let $N.father.node-path=i_k...i_2$. According to the definition of a BMC-tree (Definition 6), $N.node-path=i_k...i_2i_1$. Therefore, all bits in $N.bitmap-code$ and $N.father.bitmap-code$ are the same, except the bit assigned to the item i_1 ($index(i_1)^{th}$ bit). This bit in $N.father.bitmap-code$ is a don't-care bit, but in $N.bitmap-code$ it is 1. In the binary number $2^{index(i_1)}$, the $index(i_1)^{th}$ bit is 1 and other bits are 0. Hence, the binary operator $\vee$ turns the $index(i_1)^{th}$ bit of $N.bitmap-code$ to 1.□

Algorithm 1 (constructing_BMC_tree).

Input: A transactional database *DB*, and a threshold $min-support$.
Output: A BMC-tree (Definition 6), and L_1 (Definition 2).
1. Scan *DB* to find F_1;
2. L_1 = Sort the items of F_1 in non-descending order, with respect to $support(\alpha)$, where α is an item.
3. Create *Tr* as the root of a BMC-tree, and do the flowing assignments:
4. $Tr.item-name=\emptyset$;
5. $Tr.count=0$;
6. $Tr.bitmap-code=b_{nf-1}, ..., b_1,b_0$, where $b_i=0$, *and* $0 \le i \le nf-1$; //(Property 3)
7. **For each** transaction *T* in *DB* **do**:
8. Remove all infrequent items from T.
9. Sort T according to the order of items in $L_1^{reverse}$ (reverse order of L_1).
//Insert *T* into BMC-tree
10. $current-root=Tr$;
11. **For** each item *i* in *T* **do**:
12. Let *N* be a child of $current-root$, in such a way that $N.item-name=i$;
13. **If** such node does not exist **then**:
14. Create the new node *N*;
15. $N.item-name=i$;
16. $N.count=0$;
17. Add *N* to $current-root.children-list$;
18. **Endif**
19. $N.count=N.count+1$;
20. $N.bitmap-code=current-root.bitmap-code \vee 2^{index(i)}$; //(Property 4)
21. $current-root=N$;
22. **Endfor**
23. **Endfor**
24. **Return** A BMC-tree T_r, and a zero-based vector L_1;

Based on Definition 6, Property 3 and Property 4, the BMC-tree construction algorithm is described in Algorithm 1.

According to Definition 6, the structure of a BMC-tree is almost the same as the structure of a POC-tree (Deng & Lv, 2014), except that in a BMC-tree, each node is encoded by $BMC(node-path)$, while in a POC-tree, each node is encoded by its pre-order rank. This difference is displayed in the lines 6 and 20 of Algorithm 1.

A BMC-tree is only used to build the set of nodes associated with each frequent 1-itemset. Later, we will find that after building these sets of nodes, the BMC-tree is useless and can be deleted.

Definition 8. ($N-info$). Let N be a node in a BMC-tree. The $N-info$ of N is the pair of its $bitmap-code$ and *count* fields ($bitmap-code$, *count*). This definition is similar to definition of $N-info$ in Deng and Lv (2014).

Definition 9. ($Nodeset(itemset\ P_k)$). The *Nodeset* of itemset P_k is a set of all the $N-infos$ of nodes like N in the BMC-tree in such a way that N holds the item i_1, and each item in $i_ki_{k-1}...i_2$ is held in one of the ancestor nodes of N. By considering Definition 5 and Definition 6, the *Nodeset* of itemset P_k is defined as follows:

$$Nodeset(P_k) = \left\{ The\ N-info\ of\ N | \begin{matrix} N\ holds\ i_1,\ and\ \forall i_j, 1 \le j \le k, \\ the\ bit\ assigned\ to\ i_j\ in\ N.bitmap-code\ is\ 1 \end{matrix} \right\}$$

The *Nodeset* of each frequent 1-itemset for Example 1 is shown in Fig. 4. These *Nodesets* are extracted from Fig. 3. As two other examples, the *Nodeset* of itemsets *bd* and *abd* are shown in Fig. 5.

Definition 10. ($NegNodeset(itemset\ P_k)$). Let $2 \le k$. The *NegNodeset* of itemset $P_k=i_kP_{k-1}$ is equal to $Nodeset(P'_k=\neg i_kP_{k-1})$. Therefore,

$$NS_a \rightarrow \left\{ \begin{matrix} abcde \\ (10000, 4) \end{matrix} \right\}$$
$$NS_b \rightarrow \left\{ \begin{matrix} abcde & abcde \\ (01000, 1), & (11000, 3) \end{matrix} \right\}$$
$$NS_c \rightarrow \left\{ \begin{matrix} abcde \\ (11100, 3) \end{matrix} \right\}$$
$$NS_d \rightarrow \left\{ \begin{matrix} abcde & abcde & abcde \\ (01010, 1), & (11110, 1), & (10010, 1) \end{matrix} \right\}$$
$$NS_e \rightarrow \left\{ \begin{matrix} abcde & abcde \\ (01011, 1), & (11101, 1) \end{matrix} \right\}$$

Fig. 4. The *Nodeset* of each frequent 1-itemset for Example 1. Here, *NS* is the abbreviation for *Nodeset*.

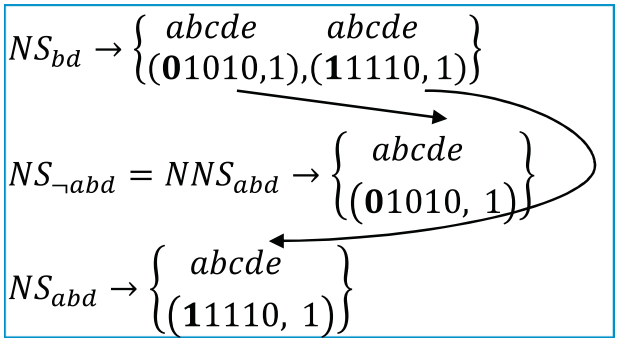

Fig. 5. The *Nodeset* of itemset *bd*. Furthermore, the *Nodeset* and *NegNodeset* of itemset *abd*, in Example 1. Here, *NNS* is the abbreviation for *NegNodeset*.

the *NegNodeset* of itemset P_k is a set of all the $N-info$s of nodes like N in the BMC-tree in such a way that N holds the item i_1, each item in $i_{k-1}...i_2$ is held in one of the ancestor nodes of N, and the item i_k is not held in any ancestor nodes of N. By considering Definitions 5 and 6, the *NegNodeset* of itemset P_k is defined as follows:

$$NegNodeset(P_k = i_k i_{k-1} \dots i_2 i_1) = Nodeset\left(P'_k = \neg i_k i_{k-1} \dots i_2 i_1\right)$$
$$= \left\{ The\ N-info\ of\ N \middle| \begin{matrix} N \text{ holds } i_1,\ \forall i_j,\ 1 \le j \le k-1, \\ \text{the bit assigned to } i_j \text{ in } N.bitmap-code \text{ is } 1. \\ \text{and the bit assigned to } i_k \text{ is } 0 \end{matrix} \right\}$$

NegNodeset and *Nodeset* (Definition 9) have similar definitions, except that in a *NegNodeset*, the item i_k does not appear in any ancestor nodes of N, but in *Nodeset*, the item i_k appears in one of the ancestor nodes of N. For example, see Fig. 5.

Property 5. $support(P_k) = \sum_{ni \in Nodeset(P_k)} ni.count$ (Deng & Lv, 2014).

For example, $Nodeset(b) = \{(01000, 1), (11000, 3)\}$ (Fig. 4), and $Nodeset(bd) = \{(01010, 1), (11110, 1)\}$ (Fig. 5). According to Property 5, $support(b) = 4$ $(1+3)$, and $support(bd) = 2$ $(1+1)$.

Rationale. Let T be a transaction that contains itemset P_k. According to the construction algorithm of BMC-trees (Algorithm 1), T must register i_k, i_{k-1},...,i_2, and i_1 to a series of nodes, N_k, N_{k-1},...,N_2, and N_1, respectively. In addition, N_j must be an ancestor of N_l, if $j > l$. According to Definition 9, the $N-info$ of N_1 must be one element in $Nodeset(P_k)$, which we denote as *ni*. Furthermore, according to the construction algorithm of BMC-trees, *ni.count* is the number of transactions that contain the itemset P_k and register the item i_1 in such a node N_1. Hence, $support(P_k) = \sum_{ni \in Nodeset(P_k)} ni.count$ (Deng & Lv, 2014).□

Property 6. Let $2 \le k$. $support(P'_k) = \sum_{ni \in NegNodeset(P_k)} ni.count$.

For example, in Fig. 5, $NegNodeset(abd) = Nodeset(\neg abd) = \{(01010, 1)\}$. According to Property 6, $support(\neg abd) = 1$.

Rationale. According to Property 5, $support(P'_k) = \sum_{ni \in Nodeset(P'_k)} ni.count$. Furthermore, according to Definition 10, $Nodeset(P'_k) = NegNodeset(P_k)$. Hence, $support(P'_k) = \sum_{ni \in NegNodeset(P_k)} ni.count$.□

Property 7. Let itemsets $P_k = i_k i_{k-1} P_{k-2}$ and $Q_{k-1} = i_k P_{k-2}$, and $3 \le k$. The *NegNodeset* of k-itemset P_k can be directly extracted from the *NegNodeset* of (k-1)-itemset Q_{k-1}, as follows:

$$NegNodeset(P_k = i_k i_{k-1} P_{k-2})$$
$$= \left\{ ni \middle| \begin{matrix} ni \in NegNodeset(Q_{k-1} = i_k P_{k-2}) \wedge \\ the\ bit\ assigned\ to\ item\ i_{k-1}\ in\ ni.bitmap-code\ is\ 1 \end{matrix} \right\}$$

In Example 1, Let $P_3 = bcd$, $Q_2 = bd$, $P_1 = d$, and $NegNodeset(bd) = Nodeset(\neg bd) = \{(10010, 1)\}$. According to Property 7, $NegNodeset(bcd)$ $(= Nodeset(\neg bcd))$ can be extracted from $NegNodeset(bd)$ $(= Nodeset(\neg bd))$ as follows: $N-info$ (10010, 1) is a member of $NegNodeset(bd)$, and its $bitmap-code$ is 10010. The bit assigned to the item c in this $bitmap-code$ (the third bit from the right) is 0. Therefore, (10010, 1) is not a member of $NegNodeset(bcd)$, and $NegNodeset(bcd) = \emptyset$.

Rationale. Let $ni_P \in NegNodeset(P_k)$, and $ni_Q \in NegNodeset(Q_{k-1})$. According to Definition 10, all bits in the $ni_P.bitmap-code$ and $ni_Q.bitmap-code$ are the same, except the bit assigned to i_{k-1}. This bit in $ni_P.bitmap-code$ is 1, but in $ni_Q.bitmap-code$, it may be 0 or 1. Therefore, if this bit in $ni_Q.bitmap-code$ is 1, then ni_Q is also a member of $NegNodeset(P_k)$.□

Property 8. Let itemsets $P_k = i_k P_{k-1}$ and $P'_k = \neg i_k P_{k-1}$, and $2 \le k$. $support(P_k) = support(P_{k-1}) - support(P'_k)$.

For example, in Table 1, $support(bd) = 2$, $support(abd) = 1$, and $support(\neg abd) = 1$. We find that $support(abd) = support(bd) - support(\neg abd)$.

Rationale. All transactions in $DB(P_{k-1})$ are divided into two groups: (1) those which contain the item i_k and (2) those which do not contain the item i_k. They have no transaction in common. Hence, $|DB(P_{k-1})| = |DB(i_k P_{k-1})| + |DB(\neg i_k P_{k-1})|$. Consequently $support(P_{k-1}) = support(i_k P_{k-1}) + support(\neg i_k P_{k-1})$, or $support(P_k) = support(P_{k-1}) - support(P'_k)$.□

Property 9. (Superset equivalence (Deng & Lv, 2014)). Given itemsets P and Q and an item i, where $P \cap Q = \emptyset$, $i \notin P$, and $i \notin Q$, if $support(P) = support(P \cup \{i\})$, then $support(P \cup Q) =$

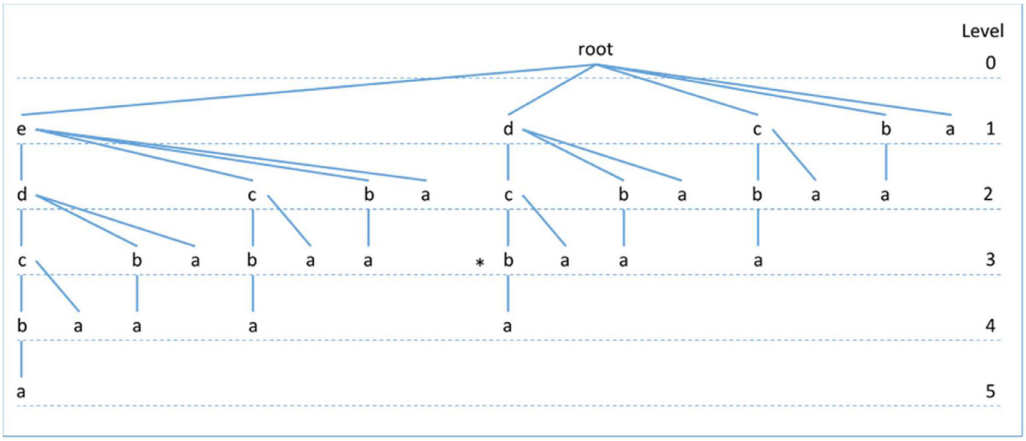

Fig. 6. The set-enumeration tree for Example 1.

$support(P \cup^{Q} \cup^{\{} i\})$. In Table 1, let $P = ab$, $Q = e$, and $i = c$. $support(ab) = support(abc) = 3$. Hence, $support(abe) = support(abce) = 1$.

Rationale. Please see Deng and Lv (2014).□

Definition 11. (Set-enumeration tree (Rymon, 1992)). Given L_1 (Definition 2), a set-enumeration tree is a tree structure such that:

(1) Each node N in the set-enumeration tree has two fields: $item - name$ and $children - list$. $N.item - name$ holds an item i ($i \in L_1 \cup^{\{} \emptyset\}$). $N.children - list$ holds all children of node N. Moreover, the node N represents an itemset $N.itemset$.
(2) The root holds the item $\emptyset$ ($root.item - name = \emptyset$) and represents the itemset $\emptyset$ ($root.itemset = \emptyset$). The child nodes of the root hold items i, where $i \in L_1$.
(3) For other nodes N, the child nodes of N hold items i, where $i \in L_1 \wedge i \succ N.item - name$, respectively. Further, the itemset of N is defined as $N.itemset = N.father.itemset \cup^{\{} N.item - name\}$, where the node $N.father$ is the father of node N.

Based on Definition 11, the pseudo-code of the set-enumeration tree construction algorithm is described in Algorithm 2.

The set-enumeration tree for Example 1 is shown in Fig. 6. For example, in Fig. 6, the node marked with an asterisk holds the item b and represents the itemset bcd.

Algorithm 2 (Set-enumeration tree construction).

Input: The zero-based vector L_1 (Definition 2).
Output: A set-enumeration tree (Definition 11).
1. Create the node *root*;
2. $root.level = 0$; // The root is at level 0;
3. $root.children - list = \emptyset$;
4. $root.item - name = \emptyset$;
5. $root.itemset = \emptyset$;
6. **for each** item $i \in L_1$ **do**:
7. Create the node $child_i$;
8. $child_i.level = root.level + 1$;
9. $child_i.item - name = i$;
10. $child_i.itemset = \{i\}$;
11. Append $child_i$ into $root.children - list$;
12. **call** constructing_set_enumeration_tree ($child_i$); //Line 15
13. **end for**
14. **return** *root*;
15. **procedure** constructing_set_enumeration_tree (N)
16. $P = N.itemset$;
17. $N.children - list = \emptyset$;
18. **for each** item $i \in L_1 \wedge i \succ N.item - name$ **do**:
19. $R = P \cup \{i\}$
20. Create the node $child_i$;
21. $child_i.level = N.level + 1$;
22. $child_i.item - name = i$;
23. $child_i.itemset = R$;
24. Append $child_i$ into $N.children - list$;
25. **call** constructing_set_enumeration_tree ($child_i$); //Line 15
26. **end for**
27. **end procedure**

4. negFIN: the proposed algorithm

negFIN employs a set-enumeration tree (Definition 11) to represent the search space. The framework of negFIN consists of three steps. In the first step, the BMC-tree is constructed, all frequent 1-itemsets and their *Nodesets* are identified, and level 1 of the set-enumeration tree is constructed. In the second step, all frequent 2-itemsets and their *NegNodesets* are identified, and level 2 of the set-enumeration tree is constructed. In the third step, all frequent k-itemsets ($3 \leq k$) and their *NegNodesets* are identified, and other levels of the set-enumeration tree are constructed. negFIN employs the superset equivalence property (Property 9) as a pruning strategy.

We demonstrate the negFIN algorithm through Example 1. For example, in the first step, $Nodeset(d)$ and $support(d)$ are computed as follows:

$$Nodeset(d) = \{(01010,\ 1),\ (11110,\ 1),\ (10010, 1)\ (\text{Figure } 4)\}. \quad (1)$$

$$support(d) = 3\ (\text{Eq. } (1) \text{ and Property } 5). \quad (2)$$

In the second step, based on the *Nodeset* of 1-itemset d (Eq. (1)) and according to Definition 10, the *NegNodeset* of 2-itemset ad is extracted as follows:

$$NegNodeset(ad) = Nodeset(\neg ad) = \{(01010,\ 1)\}. \quad (3)$$

$support(\neg ad)$ is computed as follows:

$$support(\neg ad) = 1\ (\text{Eq. } (3),\ \text{and Property } 6). \quad (4)$$

Hence, the *support* of 2-itemset ad is computed as follows:

$$support(ad) = support(d) - support(\neg ad) = 3 - 1 = 2\ (\text{Eqs. } (2) \text{ and } (4), \text{and Property } 8). \quad (5)$$

Similarly, $NegNodeset(bd) = Nodeset(\neg bd) = \{(10010,\ 1)\}$, and the *suport* of 2-itemset bd is computed as follows:

$$support(bd) = support(d) - support(\neg bd) = 3 - 1 = 2. \quad (6)$$

Algorithm 3 (negFIN algorithm).

Input: A transactional database *DB* and a threshold $min-support$.
Output: The set of all frequent itemsets, *F*.
1. $F=\emptyset$;
//First step
2. **call** constructing_BMC_tree (*DB*, $min-support$)(Algorithm 1) to construct the BMC-tree and find L_1(Definition 2);
3. $F=F\cup L_1$;
4. **for each** node *N* in the BMC-tree **do**: //Traverse the BMC-tree in an arbitrary order.
5. Append the $N-info$ of *N* into the *Nodeset* of item $N.item-name$;
6. **end for**
7. Create the node *root*;
8. $root.level=0$; // The root is at level 0;
9. $root.children-list=\emptyset$
10. $root.item-name=\emptyset$;
11. $root.itemset=\emptyset$;
12. **for each** item $i\in L_1$ **do**:
13. Create the node $child_i$;
14. $child_i.level=root.level+1$;
15. $child_i.item-name=i$;
16. $child_i.itemset=\{i\}$;
17. Append $child_i$ into $root.children-list$;
18. **call** constructing_frequent_itemset_tree ($child_i,\emptyset$); //Algorithm 4
19. **end for**
20. **return** *root*;

In the third step, based on the *NegNodeset* of 2-itemset *ad* (Eq. (3)) and according to Property 7, the *NegNodeset* of 3-itemset *abd* is extracted as follows:

$$NegNodeset(abd)=Nodeset(\neg abd)=\{(01010,\ 1)\}. \tag{7}$$

$$support(\neg abd)=1\ (\text{Eq. (7), and Property 6}). \tag{8}$$

The *support* of 3-itemset *abd* is computed as follows:

$$\begin{aligned} support(abd) &= support(bd)-support(\neg abd)=2-1\\ &=1\ (\text{Eqs. (6) and (8), and Property 8}). \end{aligned} \tag{9}$$

Algorithm 3 shows the pseudo-code of the negFIN algorithm. *F* in line (1) holds frequent itemsets and is initialized by an empty set. Line (2) builds the BMC-tree and L_1, by calling Algorithm 1. Line (3) inserts all frequent 1-itemsets into *F*. Lines (4) to (6) generate the *Nodesets* of all frequent 1-itemsets by traversing the BMC-tree in an arbitrary order. Lines (7) to (19) build a "frequent itemset tree," which is similar to a set-enumeration tree (Definition 11). Lines (7) to (11) build level 0 of the tree (the root). Lines (12) to (17) build level 1 of the tree through all frequent 1-itemsets in L_1. Line (18) builds levels k ($2\le k$) of the tree and generates all frequent k-itemsets by recursively calling the procedure **constructing_frequent_itemset_tree**() (Algorithm 4). This procedure is similar to the procedure **constructing_set_enumeration_tree**(), which is presented in Algorithm 2.

Procedure **constructing_frequent_itemset_tree** () has two parameters: *N* and FIS_{parent}. *N* is the current node in frequent itemset tree. FIS_{parent} is used to hold the frequent itemsets generated on the parent of *N*. *P* in line (2) holds the itemset represented by *N*. Lines (5) to (38) extend *P* by the item *i*. The extended itemset is denoted as *R* in line (6). Lines (8) to (24) generate the *NegNodeset* of *R*. If *R* is a 2-itemset (*N* is at level 1), then the *NegNodeset* of *R* is extracted from the *Nodeset* of *P* (Definition 10), as lines (8) to (15) do. Line (11) checks whether the condition specified in Definition 10 is true. If *R* is a k-itemset ($3\le k$), then the *NegNodeset* of *R* is extracted from the *NegNodeset* of *P* (Property 7), as lines (15) to (24) do. Line (20) checks whether the condition specified in Property 7 is true. Line (25) employs Property 6 to compute the support of Ŕ. Line (26) employs Property 8 to compute the support of *R*. Lines (27) to (37) look for items that can be used to build

Algorithm 4 (**Procedure** constructing_frequent_itemset_tree).

1. **procedure** constructing_frequent_itemset_tree (*N*, FIS_{parent})
2. $P=N.itemset$;
3. $N.children-list=\emptyset$;
4. $N.equivalent\_items=\emptyset$;
5. **for each** item $i\in L_1\wedge i\succ N.item-name$ **do**:
6. $R=iP$; //$R=P\cup^{\{}i\}$
7. $R.NegNodeset=\emptyset$;
8. **if** $N.level=1$ **then**:
//Second step
9. **for each** N-info $ni\in Nodeset(P)$ **do**:
10. //Checks whether the bit assigned to the item *i* in $ni.bitmap-code$ is 0? (Definition 10)
11. **if** $ni.bitmap-code\wedge 2^{index(i)}=0$ **then**:
12. $R.NegNodeset=R.NegNodeset\cup^{\{}ni\}$;
13. **end if**
14. **end for**
15. **else**
//Third step
16. $jX=P$; //*j* is the leftmost item in *P* and *X* is the remaining itemset. Hence, $R=ijX$
17. $Q=iX$; //Replace the first item in *P* with the item *i* ($i\succ j$).
18. **for each** N-info $ni\in NegNodeset(Q)$ **do**:
19. //Checks whether the bit assigned to the item *j* in $ni.bitmap-code$ is 1? (Property 7)
20. **if** $ni.bitmap-code\wedge 2^{index(j)}=1$ **then**:
21. $R.NegNodeset=R.NegNodeset\cup^{\{}ni\}$;
22. **end if**
23. **end for**
24. **end if**
25. $R'.support=\sum_{ni\in NegNodeset(R)} ni.count$;//Property 6
26. $R.support=P.support-R'.support$;//Property 8
27. **if** $R.support=P.support$ **then**:
28. $N.equivalent\_items=N.equivalent\_items\cup^{\{}i\}$;
29. **else**
30. **if** $R.support\ge|DB|\times min-support$ **then**:
31. Create the node $child_i$;
32. $child_i.level=N.level+1$;
33. $child_i.item-name=i$;
34. $child_i.itemset=R$;
35. Append $child_i$ into $N.children-list$;
36. **end if**
37. **end if**
38. **end for**
39. *SS* = the set of all subsets of $N.equivalent\_items$;
40. $PSet\leftarrow\{A|A=\{N.item-name\}\cup\acute{A},\acute{A}\in SS\}$;
41. **if** $FIS_{parent}=\emptyset$ **then**:
42. $FIS_N=PSet$;
43. **else**
44. $FIS_N=\{P'|P'=P_1\cup^{P_2},\ P_1\in PSet\wedge\ P_2\in FIS_{parent}\}$;
45. **end if**
46. $F=F\cup^{F} IS_N$;
47. **if** $N.children-list\neq\emptyset$ **then**:
48. **for each** $child_i\in N.children-list$ **do**:
49. **call** constructing_frequent_itemset_tree ($child_i, FIS_N$); //Algorithm 4
50. **end for**
51. **else**
52. **return**;
53. **end if**
54. **end procedure**

the child nodes of *N*. Line (27) checks whether the condition specified in the "superset equivalence property" (Property 9) is true. If this condition is true, then the item *i* is a promoted item (Deng & Lv, 2014). A promoted item is held in $N.equivalent\_items$, for future use, in line (28); the promoted items are not used to build the child nodes of *N*, because all information about the frequent itemsets related to these items are held in *N*. This pruning strategy is called promotion (Deng & Lv, 2014). Line (30) checks whether the itemset *R* is frequent. If so, then lines (31) to (35) use the item *i* to create a child node of *N*. Lines (39) to (45) identify all frequent itemsets in *N*, denoted as FIS_N. If FIS_{parent} is empty, then FIS_N is the same as *PSet*. Otherwise, FIS_N is extracted from *PSet* and FIS_{parent}, as line (44) does. Lines (47) to (51) extend the child nodes of *N* by

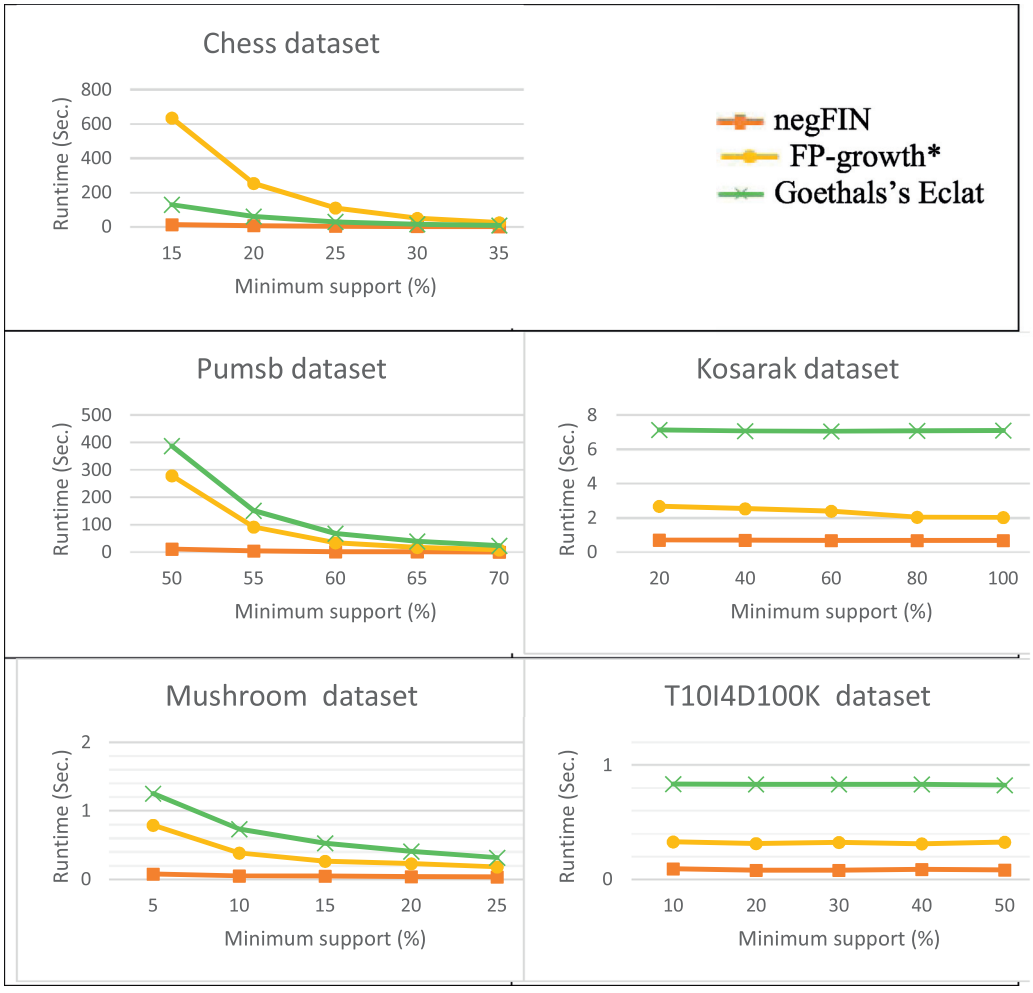

Fig. 7. Runtime of three algorithms, negFIN, FP-growth*, and Goethals's Eclat, on different datasets, depending on the minimum support.

calling **constructing_frequent_itemset_tree**() (Algorithm 4) recursively.

The time-consuming part of the negFIN algorithm (Algorithm 3) is the construction of the frequent itemset tree. The first part of the negFIN algorithm is the construction of the BMC-tree (Algorithm 1). In the worst case, the time complexity of this part is $O(nt \times nit \times \log nit)$ (the time complexity of the loop in line (7) of Algorithm 1), where $nt = |DB|$ and $nit = |I|$. The second part is the generation of the *Nodesets* of all frequent 1-itemsets. In the worst case, the time complexity of this part is $O(2^{nit})$ (the time complexity of traversing the BMC-tree). The third part is the construction of the frequent itemset tree. Levels k ($2 \leq k$) of this tree are constructed by Algorithm 4. To construct each node at these levels, first, the *NegNodeset* of the itemset assigned to that node is generated from one set of nodes with cardinality n, as the loops in lines (9) and (18) of Algorithm 4 do. The time complexity of these loops is $O(n)$. Second, the support of the itemset assigned to that node is computed. In the worst case, the time complexity of this operation is $O(n)$ (the time complexity of line (25) of Algorithm 4). Third, it is checked whether the itemset assigned to that node is frequent. The time complexity of this operation is $O(1)$. Hence, in the worst case, the time complexity of the third part of the negFIN algorithm is $O(2^{nit}n)$, where 2^{nit} is the maximum number of nodes in the frequent itemset tree.

The time complexity of the negFIN algorithm is equal to the time complexity of the third part since this part has the greatest time complexity among other parts. Let l be the number of nodes at levels k ($2 \leq k$) of the frequent itemset tree. Hence, the time complexity of the negFIN algorithm is $O(ln)$. Parameter l is the same for negFIN and the previous works (Deng, 2016; Deng & Lv, 2014; Deng & Wang, 2010; Deng et al., 2012). The time complexity of the previous works (Deng, 2016; Deng & Lv, 2014; Deng & Wang, 2010; Deng et al., 2012) is $O(l(x+y))$, where x and y are the cardinality of two sets of nodes and $O(x+y)$ is the time complexity of generating a new set of nodes.

5. Results of experiment and analysis

In order to evaluate the performance of the negFIN algorithm, we conducted two groups of experiments. The purpose of the first group of experiments is to compare the performance of the negFIN algorithm against the following algorithms: (1) Goethals's Eclat (Goethals & Zaki, 2004), which is the state-of-the-art algorithm in the family of vertical mining algorithms (Deng et al., 2012), and (2) FP-growth* (Grahne & Zhu, 2005), which is the state-of-the-art algorithm in the family of FP-tree-based pattern growth algorithms (Deng et al., 2012). Both of these algorithms are used as comparison algorithms in (Deng, 2016). In the second group of experiments, we conducted comprehensive experiments to compare the performance of the negFIN algorithm against the dFIN algorithm (Deng, 2016) separately, since (1) both algorithms belong to the same family of algorithms (nodeset-based algorithms), and (2) dFIN is the fastest algorithm among its family and other families of frequent itemset mining algorithms at present (Deng, 2016). The

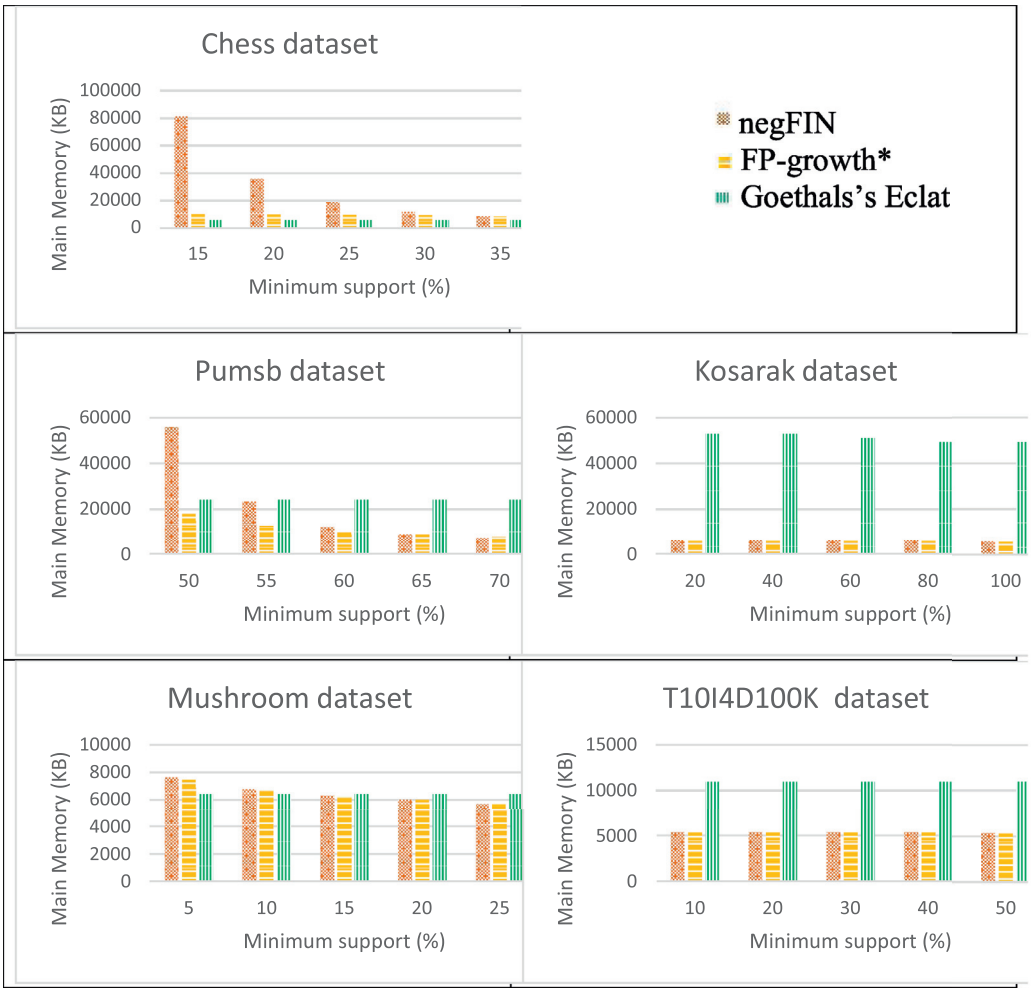

Fig. 8. Memory consumption of three algorithms, negFIN, FP-growth*, and Goethals's Eclat, on different datasets, depending on the minimum support.

Table 2
Description of the datasets used.

Dataset	Type	*#Items*	*#Transactions*	*#Avg. Length*
accidents	Real	468	340,183	33.8
chess	Real	75	3,196	37
connect	Real	129	67,557	43
kosarak	Real	41,270	990,002	8.1
mushroom	Real	119	8,124	23
pumsb	Real	2113	49,046	74
retail	Real	16,469	88,162	10.3
T10I4D100K	Synthetic	949	98,487	10

results generated by all these algorithms are the same. But these algorithms are different with regards to runtime and memory consumption.

5.1. Datasets

We ran the comparison algorithms on seven real datasets, which are common datasets from previous frequent itemset mining studies, and one synthetic dataset. These datasets can be downloaded from the FIMI repository (http://fimi.ua.ac.be). The description of these datasets is shown in Table 2. In this table, *#Items* is the number of items, *#Transactions* is the number of transactions, and *#Avg.Length* is the average transaction length. These seven real datasets are usually very dense. The synthetic dataset T10I4D100K is much sparser than these real datasets. This dataset is generated by the IBM generator, which can be downloaded from http://www.almaden.ibm.com/cs/quest/syndata.html. To generate this dataset, the average transaction size, the average maximal potentially frequent itemset size, the number of transactions in the dataset, and the number of different items used in the dataset are set to 10, 4, 98487, and 949 respectively.

5.2. Running environment

In order to make a fair comparison, all these experiments were conducted in the same software and hardware conditions. We used a computer with 8 GB memory and an Intel Core i5 3.0 GHz processor, with the Windows 10 x64. Standard Edition operating system. All these algorithms are coded in C/C++. The implementation of FP-growth* and Goethals's Eclat are available at http://fimi.ua.ac.be/src/ and http://adrem.ua.ac.be/~goethals/software/ respectively (available since August 2017) (Deng, 2016). Also we have made the source codes of dFIN and negFIN algorithms publicly available on GitHub via https://github.com/aryabarzan/dFIN and https://github.com/aryabarzan/negFIN/ respectively.

5.3. negFIN versus FP-growth* and Goethals's Eclat

The purpose of this group of experiments is to compare the runtime and memory consumption of negFIN algorithm against the

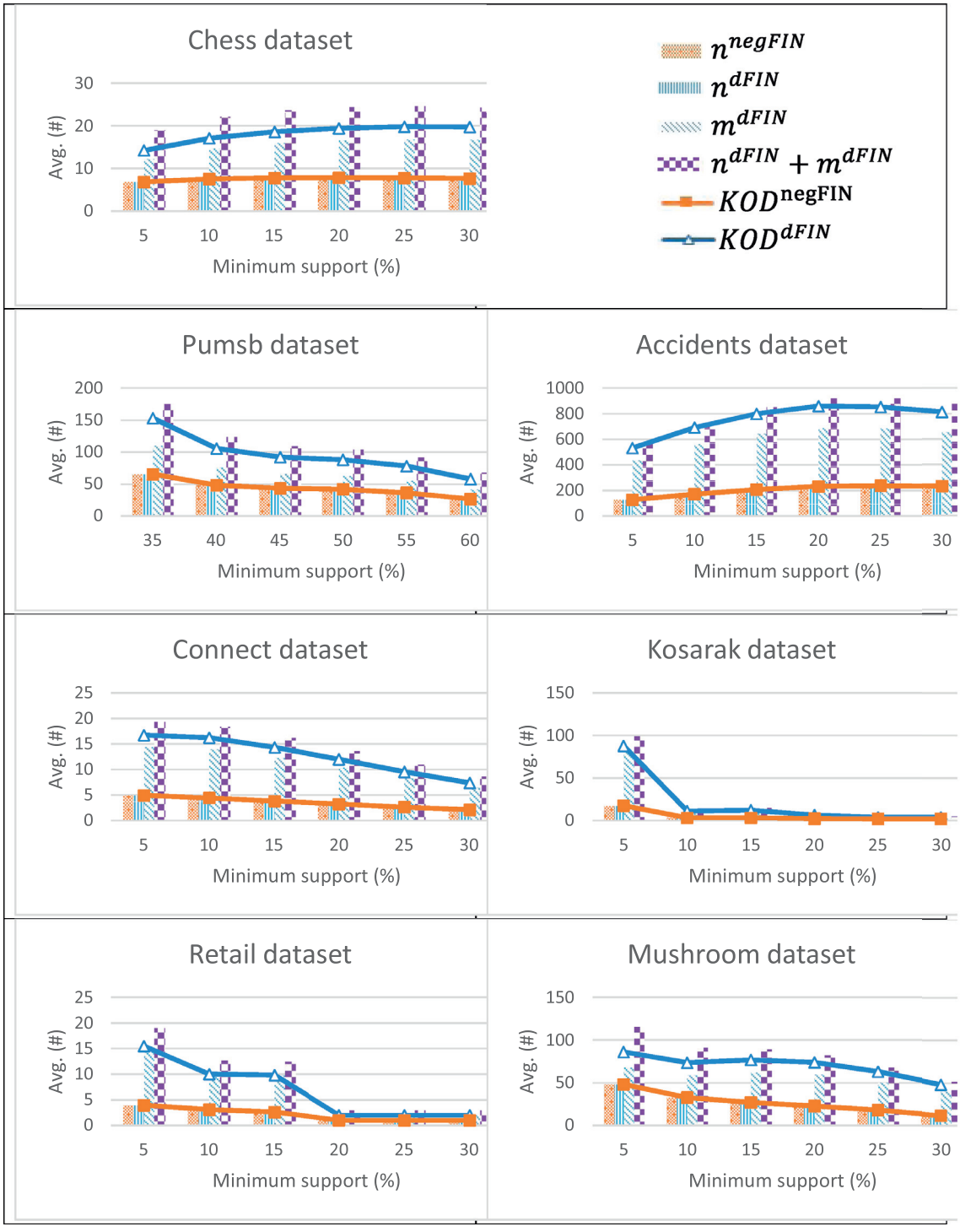

Fig. 9. The average cardinality of sets of nodes such that each *NegNodeset* and *DiffNodest* of k-itemset ($2 \leq k$) is derived from them and the average number of key operations required to derive each *NegNodeset* and *DiffNodest*, which is denoted as *KOD*, for different datasets, depending on the minimum support. Here, *KOD* is the abbreviation for key operations in each derivation.

FP-growth* and Goethals's Eclat algorithms. We conducted these experiments on five datasets—chess, pumsb, kosarak, mushroom, and T10I4D100 K—with various values of minimum support.

5.3.1. Runtime comparison

The runtime comparison of negFIN against FP-growth* and Goethals's Eclat are shown in Fig. 7. In this figure, the X and Y axes are minimum support and runtime, respectively. The runtime is the time for which the algorithm ran.

As we can see in Fig. 7, negFIN substantially surpasses FP-growth* and Goethals's Eclat on three datasets: chess, pumsb, and kosarak. Although negFIN runs faster than these algorithms on two datasets—mushroom and T10I4D100K—there is no significant difference between negFIN and these two algorithms.

5.3.2. Memory consumption comparison

The memory consumption comparison of negFIN against FP-growth* and Goethals's Eclat are shown in Fig. 8. In this figure,

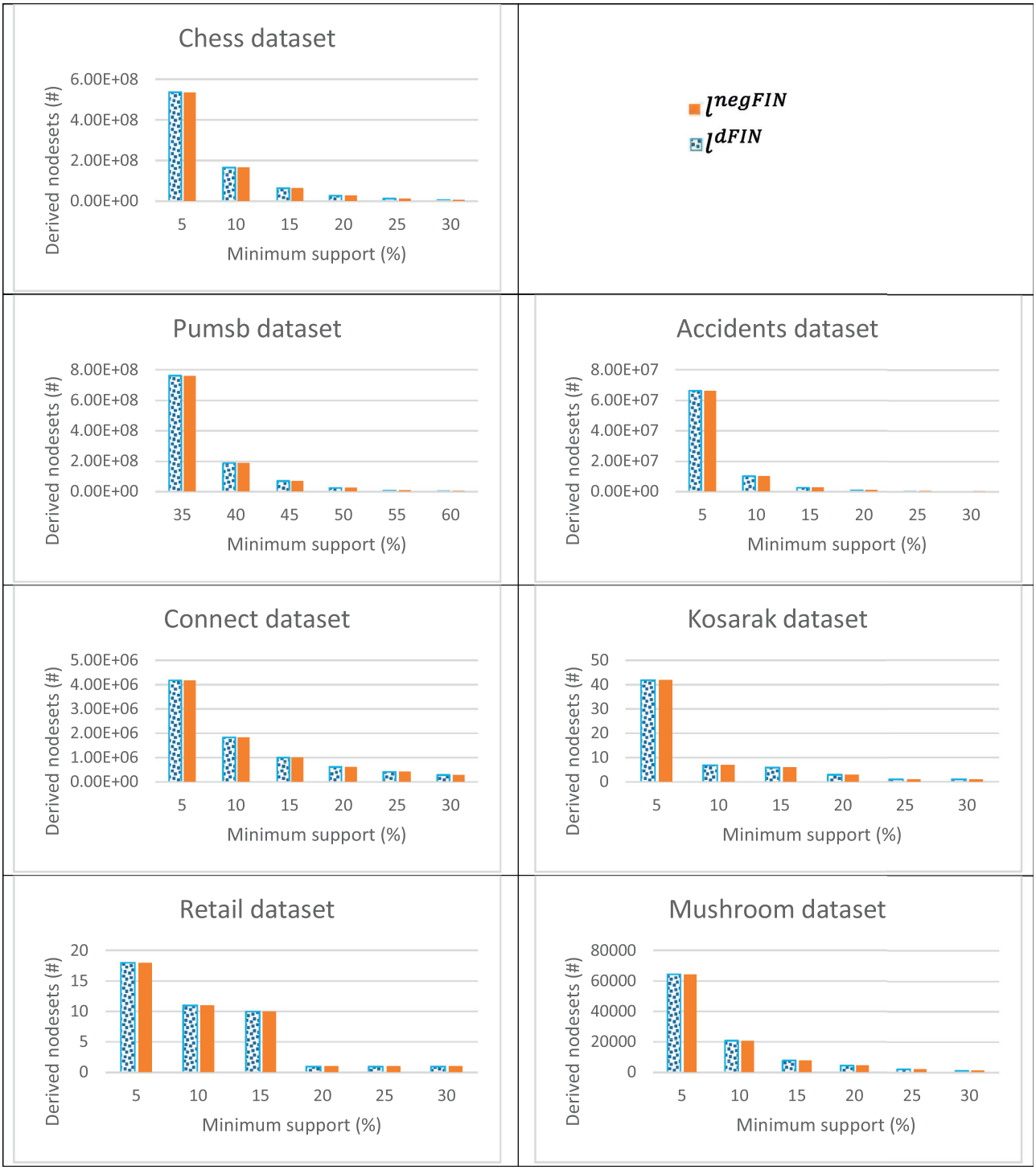

Fig. 10. The number of derived *NegNodesets* and *DiffNodesets* for different datasets, depending on the minimum support.

the Y axis is the peak memory consumption, which is measured by the *PeakWorkingSetSize* function in C/C++.

As we can see in this figure, negFIN consumes more memory than these two algorithms on the chess and pumsb datasets when minimum support is low. The reason is that the main components of memory consumption in negFIN and FP-growth* are BMC-tree and FP-tree, respectively. Since the node of the BMC-tree is a little bigger than the node of the FP-tree, it holds more information (the *bitmap – code* field) than the node of the FP-tree. Hence, the BMC-tree consumes a little more memory than the FP-tree. In addition, negFIN maintains a BMC-tree while generating the *NegNodesets* of frequent 1-itemsets.

Again, take Fig. 8 into account. We observe that negFIN and FP-growth* consume almost the same amount of memory for high minimum support on the chess and pumsb datasets, and for all minimum support on the kosarak, mushroom, and T10I4D100K datasets. Goethals's Eclat consumes more memory than negFIN and FP-growth* for high minimum support on the pumsb and mushroom datasets, and for all the minimum support on the kosarak and T10I4D100K datasets.

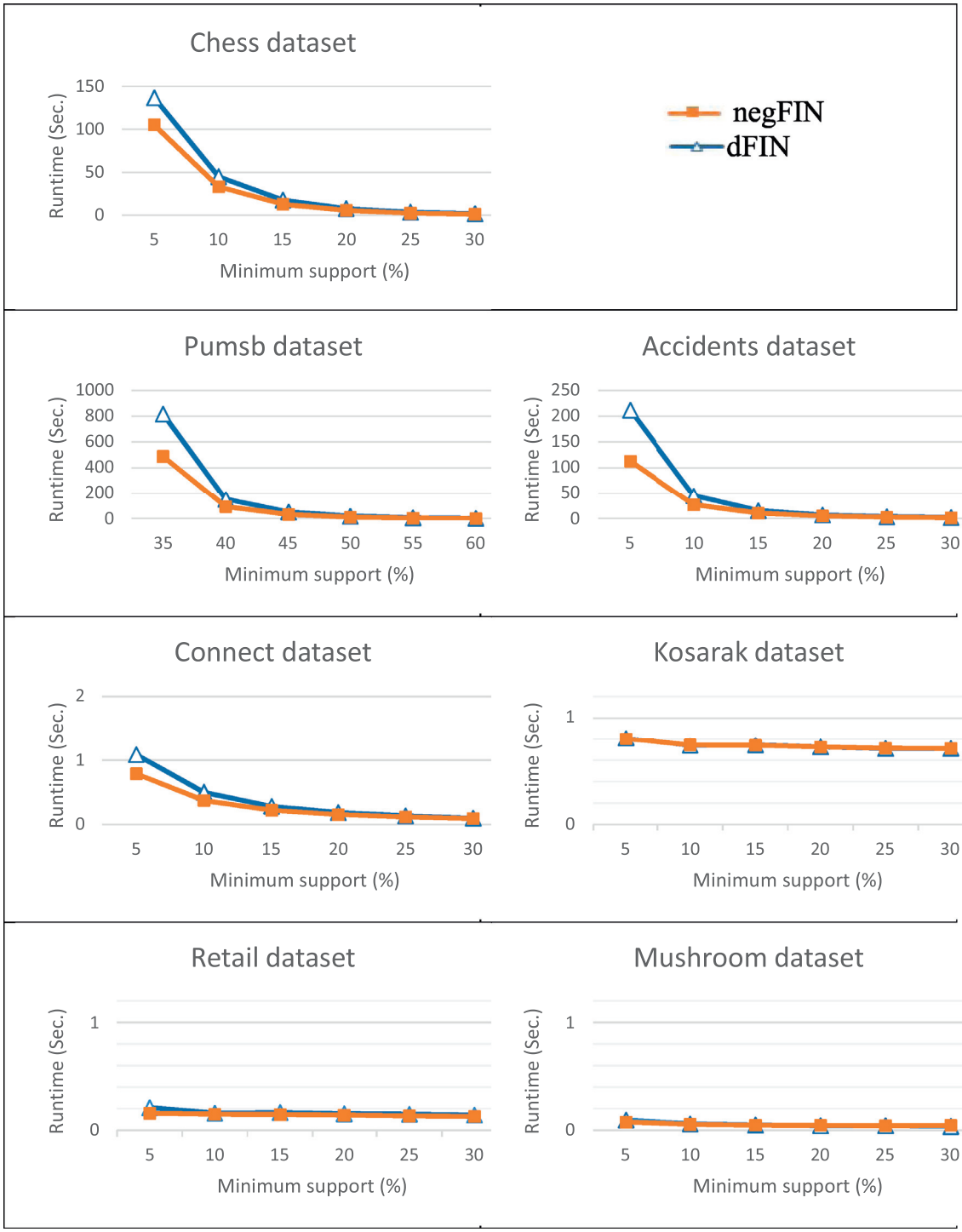

Fig. 11. Runtime comparison of negFIN against dFIN for different datasets, depending on the minimum support.

5.4. negFIN versus dFIN

In this section, we compare negFIN with dFIN based on three aspects: (1) the number of key operations, (2) the runtime, and (3) the memory consumption.

5.4.1. Number of key operations

In the negFIN (dFIN) algorithm, each *NegNodeset* (*DiffNodeset*) of k-itemset ($k \geq 2$) P is derived from one set (two sets) of nodes. Let $S_1^{NegNodeset}$ be a set of nodes such that the *NegNodeset* of P is derived from it, and $|S_1^{NegNodeset}| = n^{negFIN}$. Furthermore, let $S_1^{DiffNodeset}$ and $S_2^{DiffNodeset}$ be two sets of nodes such that the *DiffNodeset* of P is derived from them, $|S_1^{DiffNodeset}| = n^{dFIN}$, and $|S_2^{DiffNodeset}| = m^{dFIN}$. The time complexity of deriving the *NegNodeset* and *DiffNodeset* of P are $O(n^{negFIN})$ and $O(n^{dFIN} + m^{dFIN})$ respectively. The time-consuming component of negFIN (dFIN) is the derivation of these *NegNodesets* (*DiffNodesets*). Let l^{negFIN} and l^{dFIN} be the number of derived *NegNodesets* and *DiffNodesets* respectively. Consequently, the time complexity of negFIN and dFIN are $O(l^{negFIN} n^{negFIN})$ and $O(l^{dFIN}(n^{dFIN} + m^{dFIN}))$ respectively.

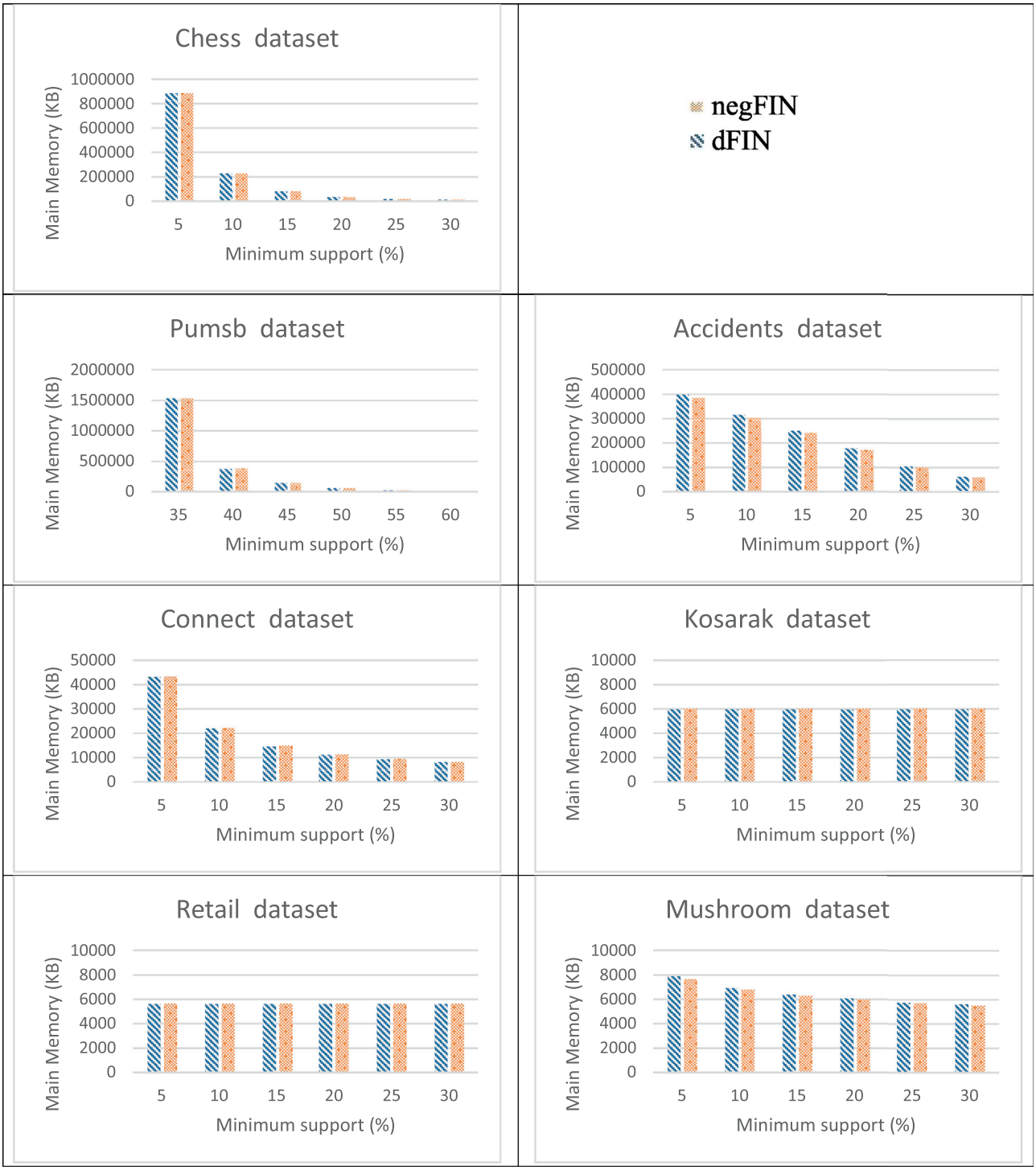

Fig. 12. Memory consumption comparison of negFIN against dFIN for different datasets, depending on the minimum support.

In Fig. 9, the average of n^{negFIN}, n^{dFIN}, m^{dFIN}, and the average number of required key operations to drive the *NegNodeset* (*DiffNodeset*) of each k-itemset ($k \geq 2$) are shown. The average number of key operations is denoted as KOD (the abbreviation for **k**ey **o**perations in each **d**erivation). Here, the key operation is the loop execution. Therefore, KOD is the average number of times when the loop is executed.

By examining Fig. 9, the following results are obtained: (1) The average number of key operations to drive *NegNodeset* is equal to n^{negFIN}. Therefore, the derivation of *NegNodeset* has a time complexity of $O(n^{negFIN})$. (2) The average number of key operations to drive *DiffNodest* is between n^{dFIN} and $(n^{dFIN} + m^{dFIN})$. Thus, the derivation of *DiffNodeset* has a time complexity of $O(n^{dFIN} + m^{dFIN})$. (3) $n^{dFIN} \leq m^{dFIN}$. (4) $n^{negFIN} = n^{dFIN}$. For simplicity, we use the notation n instead of n^{negFIN} and n^{dFIN}, and the notation m instead of m^{dFIN}. We conclude from (1) to (4) that: (5) the time complexity of the derivation of each *NegNodeset* is $O(n)$, (6) the time complexity of the derivation of each *DiffNodeset* is $O(n+m)$, and (7) $n \leq m$. Hence, the overall result is that the *NegNodeset* of the itemset is

generated about two orders of magnitude faster than its *DiffNodeset*.

In Fig. 10, l^{negFIN} and l^{dFIN} are presented for different datasets. As we can see in this figure, $l^{negFIN} = l^{dFIN}$ for all datasets. For simplicity, we use the notation l instead of l^{negFIN} and l^{dFIN}. Hence, the time complexity of negFIN and dFIN are $O(ln)$ and $O(l(n+m))$, $(n \leq m)$ respectively.

5.4.2. Runtime comparison

Fig. 11 shows the runtime comparison of negFIN against dFIN. As we can see in this figure, negFIN is not slower than dFIN on all datasets. negFIN runs faster than dFIN on some datasets, especially for low minimum support. The reason is as follows: the time complexity of negFIN and dFIN are $O(ln)$ and $O(l(n+m))$ respectively. As we can see in Fig. 9, both n and m are small values. Hence, the difference between ln and $l(n+m)$ is negligible for small values of l. Again, consider Figs. 10 and 11. As we can see in these figures, for datasets such as chess, pumsb, and accidents, where l has a large value, the difference between the runtimes of negFIN and dFIN is important.

5.4.3. Memory consumption comparison

Fig. 12 shows the memory consumption comparison of negFIN against dFIN. As we can see in this figure, the memory consumption of both algorithms is roughly the same.

6. Conclusion

In this paper, we presented a new data structure, called *NegNodeset*, to store essential information about frequent itemsets. Based on *NegNodeset*, we present an algorithm, called negFIN, to rapidly discover all frequent itemsets in databases. Compared with nFIN, the key advantages of negFIN are as follows: (1) it employs bitwise operators to generate new sets of nodes. (2) It reduces the time complexity of discovering frequent itemsets to $O(ln)$, instead of $O(l(m+n))$, where m and n are the cardinality of two base sets of nodes, $n \leq m$, and l is the number of generated sets of nodes. We implement the negFIN and dFIN algorithms and conduct extensive experiments to compare the performance of negFIN against several state-of-the-art frequent itemset mining algorithms. These experiments show that our algorithm is the fastest algorithm on all datasets with different minimum supports in comparison with previous state-of-the-art algorithms. However, on some datasets with some minimum supports, our algorithm runs with the same speed as dFIN.

7. Future research directions

Future research directions are as follows: employing *NegNodeset* to (1) mine "closed frequent itemsets" (Le & Vo, 2015; Lee, Wang, Weng, Chen, & Wu, 2008; Wang, Han, & Pei, 2003), (2) mine "maximal frequent itemsets" (Burdick, Calimlim, Flannick, Gehrke, & Yiu, 2005; Roberto & Bayardo, 1998), (3) mine "Top-Rank-k frequent itemsets" (Deng, 2014; Huynh-Thi-Le, Le, Vo, & Le, 2015), (4) mine "erasable itemsets" (Le, Vo, & Nguyen, 2014), (5) mine "fuzzy itemsets" (Lan et al., 2015; Lin et al., 2015), (6) mine "frequent disjunctive closed itemsets" (Vimieiro & Moscato, 2014), (7) mine frequent itemsets over data streams (Calders et al., 2014; Chang & Lee, 2003; Li & Deng, 2010; Troiano & Scibelli, 2014), (8) mine frequent itemsets on Hadoop (Kovács & Illés, 2013; Xun, Zhang, Qin, & Zhao, 2017), and (9) mine frequent itemsets under other distributed/parallel systems (Sohrabi & Barforoush, 2013).

Acknowledgments

The authors offer their gratitude to Dr. Zhi-Hong Deng at Peking University, Beijing, School of Electronics Engineering and Computer Science for providing the implementation codes of the FIN algorithm (Deng & Lv, 2014). We employ the FIN implementation codes to implement the dFIN algorithm (Deng, 2016) as well as our algorithm, negFIN.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

Agrawal, R., Imieliński, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. *SIGMOD Record, 22*(2), 207–216. doi: hTtps://doi.org/10.1145/170036.170072 .

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules in large databases. *Paper presented at the proceedings of the 20th international conference on very large data bases.*

Aliberti, G., Colantonio, A., Di Pietro, R., & Mariani, R. (2015). EXPEDITE: EXPress closED ITemset enumeration. *Expert Systems Applications, 42*(8), 3933–3944. doi: hTtps://doi.org/10.1016/j.eswa.2014.12.031 .

Burdick, D., Calimlim, M., Flannick, J., Gehrke, J., & Yiu, T. (2005). MAFIA: A maximal frequent itemset algorithm. *IEEE Transactions Knowledge Data Engineering, 17*(11), 1490–1504. doi: hTtps://doi.org/10.1109/TKDE.2005.183 .

Calders, T., Dexters, N., Gillis, J. J. M., & Goethals, B. (2014). Mining frequent itemsets in a stream. *Information Systems, 39*(Supplement C), 233–255. doi: hTtps://doi.org/10.1016/j.is.2012.01.005 .

Ceglar, A., & Roddick, J. F. (2006). Association mining. *ACM Computing Surveys, 38*(2), 5. doi: hTtps://doi.org/10.1145/1132956.1132958 .

Chang, J. H., & Lee, W. S. (2003). Finding recent frequent itemsets adaptively over online data streams. *Paper presented at the proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining, Washington, D.C..*

Cheng, H., Yan, X., Han, J., & Yu, P. S. (2008). Direct discriminative pattern mining for effective classification. *Paper presented at the proceedings of the 2008 IEEE 24th international conference on data engineering.*

Deng, Z.-H. (2014). Fast mining top-rank-k frequent patterns by using node-lists. *Expert Systems Applications, 41*(4, Part 2), 1763–1768. doi: hTtps://doi.org/10.1016/j.eswa.2013.08.075 .

Deng, Z.-H. (2016). DiffNodesets: An efficient structure for fast mining frequent itemsets. *Applied Soft Computing, 41*(Supplement C), 214–223. doi: hTtps://doi.org/10.1016/j.asoc.2016.01.010 .

Deng, Z.-H. An efficient structure for fast mining high utility itemsets, Applied Intelligence, published online on 08 February 2018, 1-17, doi: https://doi.org/10.1007/s10489-017-1130-x.

Deng, Z.-H., Gao, N., & Xu, X.-R. (2011). Mop: An efficient algorithm for mining frequent pattern with subtree traversing. *Fundamental Information, 111*(4), 373–390.

Deng, Z.-H., & Lv, S.-L. (2014). Fast mining frequent itemsets using nodesets. *Expert Systems Applications, 41*(10), 4505–4512. doi: hTtps://doi.org/10.1016/j.eswa.2014.01.025 .

Deng, Z.-H., & Lv, S.-L. (2015). PrePost+: An efficient N-lists-based algorithm for mining frequent itemsets via children–parent equivalence pruning. *Expert Systems Applications, 42*(13), 5424–5432. doi: hTtps://doi.org/10.1016/j.eswa.2015.03.004.

Deng, Z., & Wang, Z. (2010). A new fast vertical method for mining frequent patterns. *International Journal Computational Intelligence Systems, 3*(6), 733–744. doi: hTtps://doi.org/10.1080/18756891.2010.9727736 .

Deng, Z., Wang, Z., & Jiang, J. (2012). A new algorithm for fast mining frequent itemsets using N-lists. *Science China Information Sciences, 55*(9), 2008–2030. doi: hTtps://doi.org/10.1007/s11432-012-4638-z.

Goethals, B., & Zaki, M. J. (2004). Advances in frequent itemset mining implementations: Report on FIMI'03. *SIGKDD Exploration Newsletter, 6*(1), 109–117. doi: hTtps://doi.org/10.1145/1007730.1007744 .

Grahne, G., & Zhu, J. (2005). Fast algorithms for frequent itemset mining using FP-trees. *IEEE Transactions on Knowledge and Data Engineering, 17*(10), 1347–1362. doi: hTtps://doi.org/10.1109/TKDE.2005.166 .

Han, J., Pei, J., & Yin, Y. (2000). Mining frequent patterns without candidate generation. *SIGMOD Record, 29*(2), 1–12. doi: hTtps://doi.org/10.1145/335191.335372.

Huynh-Thi-Le, Q., Le, T., Vo, B., & Le, B. (2015). An efficient and effective algorithm for mining top-rank-k frequent patterns. *Expert Systems with Applications, 42*(1), 156–164. doi: hTtps://doi.org/10.1016/j.eswa.2014.07.045 .

Jian, P., Jiawei, H., Hongjun, L., Shojiro, N., Shiwei, T., & Dongqing, Y. (2001). H-mine: Hyper-structure mining of frequent patterns in large databases. *Paper presented at the proceedings 2001 IEEE international conference on data mining.*

Kovács, F., & Illés, J. (2013). Frequent itemset mining on hadoop. *Paper presented at the 2013 IEEE 9th international conference on computational cybernetics (ICCC).*

Lan, G.-C., Hong, T.-P., Lin, Y.-H., & Wang, S.-L. (2015). Fuzzy utility mining with upper-bound measure. *Applied Soft Computing, 30*(Supplement C), 767–777. doi: hTtps://doi.org/10.1016/j.asoc.2015.01.055.

Le, T., & Vo, B. (2015). An N-list-based algorithm for mining frequent closed patterns. *Expert Systems Applications, 42*(19), 6648–6657. doi: hTtps://doi.org/10.1016/j.eswa.2015.04.048 .

Le, T., Vo, B., & Nguyen, G. (2014). A survey of erasable itemset mining algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 4*(5), 356–379. doi: hTtps://doi.org/10.1002/widm.1137 .

Lee, A. J. T., Wang, C.-S., Weng, W.-Y., Chen, Y.-A., & Wu, H.-W. (2008). An efficient algorithm for mining closed inter-transaction itemsets. *Data & Knowledge Engineering, 66*(1), 68–91. doi: hTtps://doi.org/10.1016/j.datak.2008.02.001 .

Li, X., & Deng, Z.-H. (2010). Mining frequent patterns from network flows for monitoring network. *Expert Systems Applications, 37*(12), 8850–8860. doi: hTtps://doi.org/10.1016/j.eswa.2010.06.012 .

Lin, J. C.-W., Hong, T.-P., & Lin, T.-C. (2015). A CMFFP-tree algorithm to mine complete multiple fuzzy frequent itemsets. *Applied Soft Computing, 28*(Supplement C), 431–439. doi: hTtps://doi.org/10.1016/j.asoc.2014.11.049 .

Liu, G., Lu, H., Lou, W., Xu, Y., & Yu, J. X. (2004). Efficient mining of frequent patterns using ascending frequency ordered prefix-tree. *Data Mining and Knowledge Discovery, 9*(3), 249–274. doi: hTtps://doi.org/10.1023/B:DAMI.0000040905.52966.1a .

Roberto, J., & Bayardo, J. (1998). Efficiently mining long patterns from databases. *SIGMOD Record, 27*(2), 85–93. doi: hTtps://doi.org/10.1145/276305.276313 .

Rymon, R. (1992). Search through systematic set enumeration. *Paper presented at the proceedings of the third international conference on principles of knowledge representation and reasoning, Cambridge, MA.*

Savasere, A., Omiecinski, E., & Navathe, S. B. (1995). An efficient algorithm for mining association rules in large databases. *Paper presented at the proceedings of the 21th international conference on very large data bases.*

Shenoy, P., Haritsa, J. R., Sudarshan, S., Bhalotia, G., Bawa, M., & Shah, D. (2000). Turbo-charging vertical mining of large databases. *SIGMOD Record, 29*(2), 22–33. doi: hTtps://doi.org/10.1145/335191.335376 .

Sohrabi, M. K., & Barforoush, A. A. (2013). Parallel frequent itemset mining using systolic arrays. *Knowledge-Based Systems, 37*(Supplement C), 462–471. doi: hTtps://doi.org/10.1016/j.knosys.2012.09.005 .

Troiano, L., & Scibelli, G. (2014). Mining frequent itemsets in data streams within a time horizon. *Data Knowledge Engineering, 89*(Supplement C), 21–37. doi: hTtps://doi.org/10.1016/j.datak.2013.10.002 .

Vimieiro, R., & Moscato, P. (2014). Disclosed: An efficient depth-first, top-down algorithm for mining disjunctive closed itemsets in high-dimensional data. *Information Sciences, 280*(Supplement C), 171–187. doi: hTtps://doi.org/10.1016/j.ins.2014.04.044 .

Vo, B., Coenen, F., Le, T., & Hong, T. P. *A hybrid approach for mining frequent itemsets*. Paper presented at the 2013 IEEE international conference on systems, man, and cybernetics.

Vo, B., Le, T., Hong, T.-P., & Le, B. (2015). Fast updated frequent-itemset lattice for transaction deletion. *Data Knowledge Engineering, 96*(Supplement C), 78–89. doi: hTtps://doi.org/10.1016/j.datak.2015.04.006 .

Wang, H., Wang, W., Yang, J., & Yu, P. S. (2002). Clustering by pattern similarity in large data sets. *Paper presented at the proceedings of the 2002 ACM SIGMOD international conference on management of data, Madison, Wisconsin.*

Wang, J., Han, J., & Pei, J. (2003). CLOSET+: Searching for the best strategies for mining frequent closed itemsets. *Paper presented at the proceedings of the ninth ACM SIGKDD international conference on knowledge discovery and data mining, Washington, D.C.*

Woon, Y. K., Ng, W. K., & Lim, E. P. (2004). A support-ordered trie for fast frequent itemset discovery. *IEEE Transactions Knowledge Data Engineering, 16*(7), 875–879. doi: hTtps://doi.org/10.1109/TKDE.2004.1318569 .

Xun, Y., Zhang, J., Qin, X., & Zhao, X. (2017). FiDoop-DP: Data partitioning in frequent itemset mining on hadoop clusters. *IEEE Transactions Parallel Distributed Systems, 28*(1), 101–114. doi: hTtps://doi.org/10.1109/TPDS.2016.2560176 .

Zaki, M. J. (2000). Scalable algorithms for association mining. *IEEE Transactions Knowledge Data Engineering, 12*(3), 372–390. doi: hTtps://doi.org/10.1109/69.846291 .

Zaki, M. J., & Gouda, K. (2003). Fast vertical mining using diffsets. *Paper presented at the proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, Washington, D.C.*