Expert Systems with Applications 42 (2015) 5424–5432

Contents lists available at ScienceDirect

Expert Systems with Applications

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

PrePost$^+$: An efficient N-lists-based algorithm for mining frequent itemsets via Children–Parent Equivalence pruning

Zhi-Hong Deng [*], Sheng-Long Lv

Key Laboratory of Machine Perception (Ministry of Education), School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China

ARTICLE INFO

Article history:
Available online 20 March 2015

Keywords:
Data mining
Frequent itemset mining
N-lists
Pruning
Algorithm

ABSTRACT

N-list is a novel data structure proposed in recent years. It has been proven to be very efficient for mining frequent itemsets. In this paper, we present PrePost$^+$, a high-performance algorithm for mining frequent itemsets. It employs N-list to represent itemsets and directly discovers frequent itemsets using a set-enumeration search tree. Especially, it employs an efficient pruning strategy named Children–Parent Equivalence pruning to greatly reduce the search space. We have conducted extensive experiments to evaluate PrePost$^+$ against three state-of-the-art algorithms, which are PrePost, FIN, and FP-growth*, on six various real datasets. The experimental results show that PrePost$^+$ is always the fastest one on all datasets. Moreover, PrePost$^+$ also demonstrates good performance in terms of memory consumption since it use only a litter more memory than FP-growth* and less memory than PrePost and FIN.

© 2015 Elsevier Ltd. All rights reserved.

1. Introduction

Frequent itemset mining, first proposed by Agrawal, Imielinski, and Swami (1993), has become a popular data mining technique and plays an fundamental role in many important data mining tasks such as mining associations, correlations, episodes, and etc. Since the first proposal of this new data mining task, there have been hundreds of follow-up research publications (Han, Cheng, Xin, & Yan, 2007). Although lots of algorithms have been proposed, how to design efficient mining methods is still one of several key research problems yet to be solved (Baralis, Cerquitelli, & Chiusano, 2009; Deng, Wang, & Jiang, 2012).

In recent years, we present an algorithm called PrePost (Deng et al., 2012) for mining frequent itemsets. The high efficiency of PrePost is achieved by: (1) employing a novel data structure named N-list to represent itemsets; and (2) adopting single path property of N-list to directly discovery frequent itemsets without generating candidate itemsets in some cases. The experiments in Deng et al. (2012) show that PrePost runs fast than some state-of-the-art mining algorithms including FP-growth and FP-growth*.

Although PrePost adopts single path property of N-list to prune the search space, it still incurs the problem of too many candidates because it employs Apriori-like approach for mining frequent itemsets. In this paper, we propose a new algorithm called PrePost$^+$, which can effectively avoid the above problem. PrePost$^+$ employs N-list to represent itemsets and directly discovers frequent itemsets in an itemset & N-list search tree. For avoiding repetitive search, it also adopts Children–Parent Equivalence pruning to greatly reduce the search space. For evaluating the performance of PrePost$^+$, we conduct a comprehensive performance study to compare it against three state-of-the-art algorithms: PrePost, FIN, and FP-growth*. The experimental results show that PrePost$^+$ is efficient and runs much faster than all three compared algorithms while it only consumes a bit more memory than FP-growth*, the best one in terms of memory consumption.

The rest of this paper is organized as follows. In Section 2, we introduce the background and related work for frequent itemset mining. In Section 3, basic principles are presented. We describe PrePost$^+$ in Section 4. Experiment results are shown in Section 5 and conclusions are given in Section 6.

2. Related work

Frequent itemset mining was initially proposed for market basket analysis in dealing with the problem of mining association rule. Formally, it can be described as follows. Assume $I = \{i_1, i_2, \ldots, i_m\}$ is the universal item set and $DB = \{T_1, T_2, \ldots, T_n\}$ is a transaction database, where each T_k $(1 \leqslant k \leqslant n)$ is a transaction which is a set of items such that $T_k \subseteq I$. P is called an itemset if P is a set of items. Let P be an itemset. We say transaction T contains P if and only if $P \subseteq T$. The support of itemset P is the number of transactions in DB that contain P. Let ξ be the predefined minimum support threshold and $|DB|$ be the number of transactions in DB. An itemset P is frequent if P's support is not less than $\xi \times |DB|$. Given

* Corresponding author.
E-mail addresses: zhdeng@cis.pku.edu.cn (Z.-H. Deng), davidfracs@gmail.com (S.-L. Lv).

http://dx.doi.org/10.1016/j.eswa.2015.03.004
0957-4174/© 2015 Elsevier Ltd. All rights reserved.

transaction database *DB* and threshold ξ, the task of mining frequent itemsets is to discover the set of all itemsets with support not less than $\xi \times |DB|$.

Most of the previously proposed algorithms for mining frequent itemsets can be clustered into two groups: the Apriori-like method and the FP-growth method. The Apriori-like method employs anti-monotone property (Agrawal & Srikant, 1994) to find frequent itemsets. It generates candidate itemsets of length $(k+1)$ in the $(k+1)$th pass by using frequent itemsets of length k generated in the previous pass, and counts the supports of these candidate itemsets in the database. A lot of studies, such as Agrawal and Srikant (1994), Savasere, Omiecinski, and Navathe (1995), Shenoy et al. (2000), Zaki (2000), Zaki (2003), adopt the Apriori-like method. The previous studies indicate that the Apriori-like method have two notable shortcomings. The first one is that it repeatedly scans the database. The second one is that it generates a large set of candidates. These two shortcomings make Apriori-like method inefficient. To avoid scanning the database iteratively, a number of vertical mining algorithms have been proposed (Burdick, Calimlim, Flannick, Gehrke, & Yiu, 2005; Shenoy et al., 2000; Zaki, 2000; Zaki, 2003). In these algorithms, each item or itemset is associated with a vertical representation (such as Tid-set or diffsets (Zaki, 2003)). The advantage of the vertical representation is that the counting of supports of itemsets can be obtained via set intersection, which avoids scanning the whole database. Vertical mining algorithms have proven to be very effective and usually outperform horizontal mining methods (Zaki, 2003). To avoid generating too many candidates, the FP-growth method mines frequent itemsets without generating candidates. The FP-growth method adopts a highly condensed data structure called FP-tree to store databases and employs a divide-and-conquer strategy to mine frequent itemsets directly, which make it much more efficient than the Apriori-like method. Some algorithms (Grahne & Zhu, 2005; Han, Pei, & Yin, 2000; Liu, Lu, Lou, Xu, & Yu, 2004; Pei et al., 2001) are based on the FP-growth method. However, the FP-growth method becomes inefficient when datasets are sparse because FP-trees become very complex and larger.

In recent years, we propose a new framework of itemset representation. In this framework, each itemset is represented by a list of nodes in the coding prefix tree of a dataset. Itemset representation based on node list has three advantages. First, a coding prefix tree stores sufficient information about frequent itemsets in a dataset while the former is much compressed than the later. Therefore, the length of the node list of an itemset is much shorter than that of its Tid-set or diffsets. Second, the sum of counts stored in the node list of an itemset is equal to its support. Therefore, the support of an itemset can be easily computed by just scanning its node list. Third, the Node list of an itemset of length $(k+1)$ can be obtained by intersecting node lists of two itemsets of length k. The above three advantages make itemset representation based on node list very suitable for frequent itemset mining. Under this framework, we further propose three kinds of itemset representations, which are Node-list (Deng & Wang, 2010), N-list (Deng et al., 2012), and Nodeset (Deng & Lv, 2014). All of the itemset representations are based on a prefix tree called PPC-tree, which store the sufficient information about frequent itemsets. Compared with Node-list, N-list has a useful characteristic named single path property, which can be employed to mine frequent itemsets directly without generating candidates in some cases. Therefore, algorithms (Deng et al., 2012; Vo, Coenen, Le, & Hong, 2013) based on N-list, such as PrePost (Deng et al., 2012), are more efficient that PPV (Deng & Wang, 2010). Both N-list and Node-list need to encode a node of a PPC-tree with pre-order and post-order code, which is memory-consuming and inconvenient to mine frequent itemsets. Whereafter, we propose Nodeset (Deng & Lv, 2014), a novel structure, where a node is encoded only by preorder or post-order code. Based on Nodeset, a mining algorithm named FIN is proposed. The experimental results in Deng and Lv (2014) show that FIN consumes less memory than PrePost while they have almost the same efficiency. Besides used in frequent itemset mining, itemset representation based on node list also proves to be very suitable for erasable itemset mining (Deng & Xu, 2012; Le, Vo, & Coenen, 2013; Le et al., 2014) and top-rank-k frequent pattern mining (Deng, 2014; Huynh-Thi-Le, Le, Vo, & Le, 2015).

Compared with the previous works, which mine frequent itemsets based on the new framework of itemset representation, our main contributions are: (1) we introduce Children–Parent Equivalence pruning technique into PrePost to promote its performance, (2) The experimental results show that PrePost$^+$ is effective and performs better than PrePost on all datasets in terms of runtime and memory consumption.

3. Basic principles

This section introduces relevant concepts and properties about N-list, which is the basic of PrePost$^+$. We adopt the notations used in Deng et al. (2012). For more details, please refer to Deng et al. (2012).

Definition 1. Given a transaction database, *DB* and threshold ξ, the succinct version of *DB* is a database generated by deleting infrequent items of each transaction in *DB* and sorting remainder items according to the descending order of support.

The succinct version of *DB* contains all necessary information related to frequent itemsets. Therefore, we can directly mine frequent itemsets from the succinct version of *DB* instead of *DB*. From here on, *DB* is considered as its succinct version when mentioned in the remainder of this paper.

Definition 2. Given a succinct *DB*, PPC-tree is a tree structure defined as follows:

(1) It consists of one root labeled as "null", and a set of item prefix subtrees as the children of the root.
(2) Each node in the item prefix subtree consists of five fields: *item-name*, *count*, *children-list*, *pre-order*, and *post-order*. *item-name* registers which item this node represents. *count* registers the number of transactions presented by the portion of the path reaching this node. *children-list* registers all children of the node. *pre-order* is the pre-order rank of the node. *post-order* is the post-order rank of the node. For a node, its *pre-order* is the sequence number of the node when scanning the tree by pre-order traversal and its *post-order* is the sequence number of the node when scanning the tree by post-order traversal.

Let's examine the following example.

Example 1. Let the transaction database, *DB*, be represented by the information from the left two columns of Table 1 and ξ = 0.4. The

Table 1
A transaction database.

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

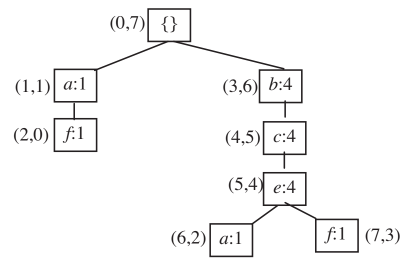

Fig. 1. The PPC-tree resulting from Example 1.

frequent 1-itemsets set $F_1 = \{a, b, c, e, f\}$. Note that, the last column is the succinct version.

Fig. 1 shows the PPC-tree resulting from Example 1. The node with (3, 6) means that its *pre-order* is 3, *post-order* is 6, the *item-name* is *b*, and *count* is 4.

Definition 3 (*PP-code*). For each node *N* in a PPC-tree, we call $\langle (N.pre-order, N.post-order) : count\rangle$ the PP-code of *N*.

Definition 4 (*the N-list of frequent item*). Given a PPC-tree, the N-list of a frequent item is a sequence of all the PP-codes of nodes registering the item in the PPC-tree. The PP-codes are arranged in an ascending order of their pre-order values.

Fig. 2 shows the N-lists of all frequent items in Example 1.

Definition 5 ($\succ$ *relation*). For any two frequent items i_1 and i_2, we denote $i_1 \succ i_2$ (or $i_2 \prec i_1$) if and only if i_1 is ahead of i_2 in *DB*.

For simplicity of description, any itemset *P* is denoted by $i_1i_2\ldots i_k$, where $i_1 \succ i_2 \succ \ldots \succ i_k$ in the remainder of this paper.

Definition 6 (*the N-list of 2-itemset*). Given any two different frequent items i_1 and i_2, whose N-lists are $\{\langle(x_{11}, y_{11}): z_{11}\rangle, \langle(x_{12}, y_{12}): z_{12}\rangle, \ldots, \langle(x_{1m}, y_{1m}): z_{1m}\rangle\}$ and $\{\langle(x_{21}, y_{21}): z_{21}\rangle, \langle(x_{22}, y_{22}): z_{22}\rangle, \ldots, \langle(x_{2n}, y_{2n}): z_{2n}\rangle\}$ respectively. The N-list of 2-itemset i_1i_2 is a sequence of PP-codes sorted by *pre-order* ascending order and is generated by intersecting the N-lists of i_1 and i_2, which follows the rule below:

First, for any $\langle(x_{1p}, y_{1p}): z_{1p}\rangle \in$ the N-list of i_1 ($1 \leqslant p \leqslant m$) and $\langle(x_{2q}, y_{2q}): z_{2q}\rangle \in$ the N-list of i_2 ($1 \leqslant q \leqslant n$), if $\langle(x_{1p}, y_{1p}): z_{1p}\rangle$ is an ancestor of $\langle(x_{2q}, y_{2q}): z_{2q}\rangle$, then $\langle(x_{1p}, y_{1p}): z_{2q}\rangle$ is added to the N-list of i_1i_2. After that, we get an initial N-list of i_1i_2.

Second, check the initial N-list of i_1i_2 again. Merge the nodes with the form of $\langle(x_{1b}, y_{1b}): z_{1b}\rangle\ \langle(x_{1b}, y_{1b}): z_{2b}\rangle \ldots \langle(x_{1b}, y_{1b}): z_{rb}\rangle$ to get a new node $\langle(x_{1b}, y_{1b}): (z_{1b}+_{2b}\ldots+z_{rb})\rangle$.

Similar to Definition 6, we present the definition of the N-list of a *k*-itemset ($k \geqslant 3$) as follows.

Definition 7 (*the N-list of k-itemset*). Let $P = i_xi\ i_1\ i_2\ldots i_{(k-2)}$ be a itemset ($k \geqslant 3$ and each item in *P* is a frequent item), the N-list of $P_1 = i_x\ i_1\ i_2\ldots i_{(k-2)}$ be $\{\langle(x_{11}, y_{11}): z_{11}\rangle, \langle(x_{12}, y_{12}): z_{12}\rangle, \ldots, \langle(x_{1m}, y_{1m}): z_{1m}\rangle\}$, and the N-list of $P_2 = i_y\ i_1\ i_2\ldots i_{(k-2)}$ be $\{\langle(x_{21}, y_{21}): z_{21}\rangle, \langle(x_{22}, y_{22}): z\ _{22}\rangle, \ldots, \langle(x_{2n}, y_{2n}): z_{2n}\rangle\}$ .The N-list of *P* is a sequence of PP-codes according to *pre-order* ascending order and generated by intersecting the N-lists of P_1 and P_2, which follows the rule below:

$b \longrightarrow$ <(3,6): 4>

$c \longrightarrow$ <(4,5): 4>

$e \longrightarrow$ <(5,4): 4>

$a \longrightarrow$ <(1,1): 1>- - -<(6,2): 1>

$f \longrightarrow$ <(2,0): 1>- - -<(7,3): 1>

Fig. 2. The N-lists of frequent items in Example 1.

$be \longrightarrow$ <(3,6): 4>

$ce \longrightarrow$ <(4,5): 4>

$bce \longrightarrow$ <(3,6): 4>

Fig. 3. The N-lists of itemsets.

First, for any $\langle(x_{1p}, y_{1p}): z_{1p}\rangle \in$ the N-list of P_1 ($1 \leqslant p \leqslant m$) and $\langle(x_{2q}, y_{2q}): z_{2q}\rangle \in$ the N-list of P_2 ($1 \leqslant q \leqslant n$), if $\langle(x_{1p}, y_{1p}): z_{1p}\rangle$ is an ancestor of $\langle(x_{2q}, y_{2q}): z_{2q}\rangle$, then $\langle(x_{1p}, y_{1p}): z_{2q}\rangle$ is added to the N-list of $i_xi\ i_1\ i_2\ldots i_{(k-2)}$. After that, we get an initial N-list of *P*.

Second, we check the initial N-list of *P* again and merge the nodes with the form of $\langle(x_{1b}, y_{1b}): z_{1b}\rangle\ \langle(x_{1b}, y_{1b}): z_{2b}\rangle \ldots \langle(x_{1b}, y_{1b}): z_{rb}\rangle$ to get a new node $\langle(x_{1b}, y_{1b}): (z_{1b}+_{2b}\ldots+z_{rb})\rangle$.

Fig. 3 shows the N-lists of *be*, *ce*, *bce*.

For N-lists, we have the following property [Deng et al. 2012].

Property 1. Given an N-list of any *k*-itemset $P = i_1\ i_2\ldots i_k$, which is denoted by $\{\langle(x_1, y_1): z_1\rangle, \langle(x_2, y_2): z_2\rangle, \ldots, \langle(x_m, y_m): z_m\rangle\}$, the *support* of itemset *P* is $z_1+z_2+\ldots+z_m$.

4. PrePost$^+$: The proposed method

The framework of PrePost$^+$ is the same as that of PrePost, which consists of: (1) Construct PPC-tree and identify all frequent 1-itemsets; (2) based on PPC-tree, construct the N-list of each frequent 1-itemset; (3) scan PPC-tree to find all frequent 2-itemsets; (4) mine all frequent k(>2)-itemsets. The main difference between PrePost$^+$ and PrePost is that PrePost$^+$ adopts superset equivalence as pruning strategy while PrePost$^+$ adopts single path property of N-list as pruning strategy.

For facilitating the mining process, PrePost$^+$ adopts a set-enumeration tree (Burdick et al., 2005) to represent the search space. Given a set of items $I = \{i_1, i_2, \ldots, i_m\}$ where $i_1 \prec i_2 \prec \ldots \prec i_m$, a set-enumeration tree can be constructed as follows. Firstly, the root of the tree is created. Secondly, the *m* child nodes of the root registering and representing *m* 1-itemsets are created, respectively. Thirdly, for a node representing itemset $\{i_{js}\ i_{-1}\ldots i_{j1}\}$ and registering i_{js}, the $(m - j_s)$ child nodes of the node representing itemsets $\{i_{js+1}i_{js}i_{-1}\ldots i_{j1}\}$, $\{i_{js+2}i_{js}i_{-1}\ldots i_{j1}\},\ldots,$ $\{i_mii_{js-1}\ldots i_{j1}\}$ and registering i_{js+1}, $i_{js+2},\ldots,$ i_m respectively are created. Finally, the set-enumeration tree is built by executing the third step repeatedly until all leaf nodes are created. Let's take Example 1 into account. The set-enumeration tree for finding frequent itemsets is depicted in Fig. 4. For example, the node in the bottom left of Fig. 4 represents itemset {*bceaf*} and registers item *b*.

Property 2. Given itemset *P* and item *i* ($\notin P$), if the support of *P* is equal to the support of $P \cup \{i\}$, for any itemset *A* ($A \cap P = \emptyset \wedge i \notin A$), the support of $A \cup P$ is equal to the support of $A \cup P \cup \{i\}$.

Rationale. The fact that the support of *P* is equal to the support of $P \cup \{i\}$ means that any transaction containing *P* also contains *i*. Given a transaction *T*, if *T* containing $A \cup P$, it must contain *A*. According to the anterior conclusion, we know that *T* also contains *i*. That is, the support of $A \cup P$ is equal to the support of $A \cup P \cup \{i\}$.

By employing Property 2, PrePost$^+$ can greatly reduce the search space. Let's examine Fig. 4. When generating the node representing *ce*, we find the support of *ce* is equal to the support of *e*, which is 4. Let *A* be any superset of *e* without containing *c*. Note that, *A* can be rewritten as $(A\text{-}\{e\}) \cup \{e\}$. Therefore, the support of *A* is equal to the

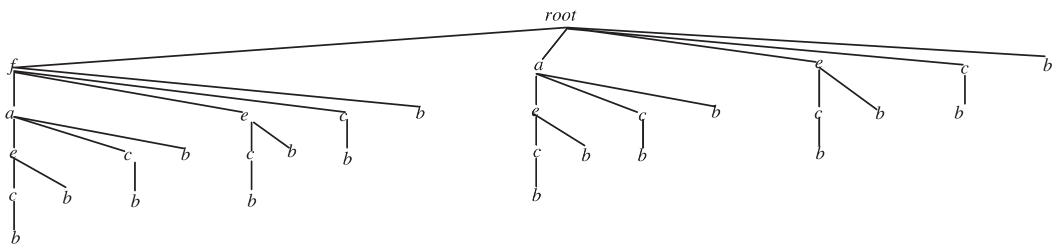

Fig. 4. A set-enumeration tree example.

support of $A \cup \{c\}$ according to Property 2. We denote the set of all frequent itemsets under the node representing e as FIS_e. In addition, $FIS_{e/c}$ is defined as the set of frequent itemsets which do not contain c in FIS_e and $FIS_{e,c}$ is defined as the set of frequent itemsets which contain c in FIS_e. Clearly, the union of $FIS_{e,c}$ and $FIS_{e/c}$ is FIS_e. To obtain $FIS_{e,c}$, we should search the subtree rooted at the node representing ce. To obtain $FIS_{e/c}$, we should search these subtrees rooted at the child nodes of the node representing e except the child node representing ce. According to Property 2, $FIS_{e,c}$ is equal to $\{A \cup \{c\}| \ A \in FIS_{e/c}\}$. Therefore, we can obtain FIS_e without searching the subtree rooted at the node representing ce.

Algorithm 1 shows the pseudo-code of PrePost$^+$. Step (1) – (4) of PrePost$^+$ is the same as corresponding procedures of PrePost, which generate the frequent 1-itemsets, frequent 2-itemsets, and their N-list. Function ***NL_intersection***() are used to generate N-lists of $(k+1)$-itemsets by intersecting N-lists of k-itemsets. Step (6) – (9) call Procedure ***Building_Pattern_Tree*** () to search the set-enumeration tree (from level 2) and discovery all frequent k-itemsets ($k \geqslant 3$) extended from each frequent 2-itemset.

NL_intersection() is a linear algorithm, thus it is very efficient. Step (1) – (9) adopts 2-way comparison to find all elements in NL_1, each of which is an ancestor of some element in NL_2. All such elements constitute the intermediate result. Step 10 – (9) merge elements with same (*pre-code*, *post-code*) pair in the intermediate result to obtain the final result. Let m and n are the cardinalities of NL_1 and NL_2 respectively. The computational complexity of 2-way comparison is $O(m+n)$. According to 2-way comparison, the cardinality of the intermediate result is less than the cardinality of NL_2. Therefore, the whole computational complexity of ***NL_intersection***() is $O(m+n)$.

Procedure ***Building_Pattern_Tree*** () employ Property 2 to construct a compressed frequent itemset tree without generating the subtree rooted at a child node of a node if the support of the itemset represented by the child node is equal to the support of the itemset represented by the node. Step (4) – (16) check each item in *Cad_items*, which is used to extend *Nd*. In Step (6), $P_1[1]$ means the first item in P_1. P in Step (7) is an itemset with i as the first item and *Nd.itemset* as the remainder. Step (8) generates the N-list of P. As shown by Step (9) and (10), if the support of P is equal to the support of *Nd.itemset*, only i is inserted into *Nd.equivalent_items* without generating the node representing P. This technique is called Children–Parent Equivalence pruning, which has been used to discover maximal frequent itemset in Burdick et al. (2005). Step (11) – (16) find the items that are used to construct the child nodes of Nd_i and store these items in *Next_Cad_ items* for future extension. Step (17) – (24) find all frequent itemsets contained in *Nd*. Step (26) – (27) continue to extend the child nodes of *Nd* by recursively calling ***Building_Pattern_Tree*** ().

Algorithm 1: PrePost$^+$ Algorithm

```
Input: A transaction database $DB$ and a minimum support $\xi$.
Output: $F$, the set of all frequent itemsets.
(1) Scan $DB$ to obtain $F_1$, the set of all frequent 1-itemset, and
    build the PPC-tree;
(2) Scan the PPC-tree to generate the N-lists of frequent 1-
    itemset;
(3) Scan the PPC-tree to obtain $F_2$, the set of all frequent 2-
    itemset
(4) For each $i_s i_t$ ($\in F_2$), generate its N-list by calling
    NL_intersection($i_s$.N-list, $i_t$.N-list);
(5) $F \leftarrow F_1$;
(6) For each frequent itemset, denoted as $i_x i_y$, in $F_2$ do
(7)     Create the root of a tree, $FPT_{xy}$, and label it with $i_x i_y$;
(8)     Building_Pattern_Tree($FPT_{xy}$, $\{i \mid i \in F_1, i \succ i_x\}$, $\emptyset$);
(9) Return $F$;
Procedure Building_Pattern_Tree (Nd, Cad_items, Parent_fit)
(1) $Nd.equivalent\_items \leftarrow \emptyset$;
(2) $Nd.childnodes \leftarrow \emptyset$;
(3) $Next\_Cad\_items \leftarrow \emptyset$;
(4) For each $i \in Cad\_items$ do
(5)     $P_1 \leftarrow Nd.itemset$;
(6)     $P_2 \leftarrow \{i\} \cup (P_1 - P_1[1])$
(7)     $P \leftarrow \{i\} \cup P_1$;
(8)     $P.N\text{-}list \leftarrow$ NL_intersection($P_1$, $P_2$);
(9)     If $P.support = P_1 support$ then
(10)        $Nd.equivalent\_items \leftarrow Nd.equivalent\_items \cup \{i\}$;
(11)    Else if $P.support \geqslant |DB| \times \xi$, then
(12)        Create node $Nd_i$;
(13)        $Nd_i.label \leftarrow i$;
(14)        $Nd_i.itemset \leftarrow P$;
(15)        $Nd.childnodes \leftarrow Nd.childnodes \cup \{Nd_i\}$;
(16)        $Next\_Cad\_items \leftarrow Next\_Cad\_items \cup \{i\}$;
(17) If $Nd.equivalent\_items \neq \emptyset$ then
(18)    $SubSets \leftarrow$ the set of all subsets of $Nd.equivalent\_items$;
(19)    $Cand\_itemsets \leftarrow \{A \mid A = Nd.label \cup A', A' \in SS\}$;
(20)    If $Parent\_fit = \emptyset$, then
(21)        $Nd\_fit \leftarrow Cand\_itemsets$;
(22)    Else
(23)        $Nd\_fit \leftarrow \{P' \mid P' = P_1 \cup P_2, (P_1 \neq \emptyset \wedge P_1 \in Cand\_itemsets)$
    and $(P_2 \neq \emptyset \wedge P_2 \in Parent\_fit\}$;
(24)    $F \leftarrow F \cup Nd\_fit$;
(25) If $Nd.childnodes \neq \emptyset$ then
(26)    For each child node, $Nd_i$, do
(27)        Building_Pattern_Tree($Nd_i$, $\{j \mid j \in Next\_Cad\_items, j$
    $\succ i\}$, $Nd\_fit$);
(28) Else Return.
Function NL_intersection($NL_1$, $NL_2$)
Input: $NL_1 = \{\langle(x_{11}, y_{11}):z_{11}\rangle, \langle(x_{12}, y_{12}):z_{12}\rangle, \ldots, \langle(x_{1m},$
```

(continued on next page)

Algorithm 1 (*continued*)

```
    $y_{1m}$):$z_{1m}$⟩} and $NL_2 = \{\langle(x_{21}, y_{21}):z_{21}\rangle, \langle(x_{22}, y_{22}):z_{22}\rangle, \ldots, \langle(x_{2n}, y_{2n}):z_{2n}\rangle\}$, which are the N-lists of $P_1 = i_u\, i_1\, i_2 \ldots i_{(k-2)}$ and $P_2 = i_v\, i_1\, i_2 \ldots i_{(k-2)}$ ($i_u \succ i_v$) respectively.
Output: $NL_3$, the N-list of $P_3 = i_u i_v\, i_1 i_2 \ldots i_{(k-2)}$.
(1) $i \leftarrow 1$;
(2) $j \leftarrow 1$;
(3) while ($i \leqslant m$ && $j \leqslant n$)
(4)   if ($x_{1i} \langle x_{2j}$) then
(5)     if ($y_{1i} \rangle y_{2j}$) then
(6)       Insert $\langle(x_{1i}, y_{1i})$: $z_{2j}\rangle$ into $NL_3$;
(7)     $j$ + +;
(8)     else $i$ + +;
(9)   else $j$ + +;
// merge elements with same (pre-code, post-code) pair in $NL_3$.
(10) $ptr_1 \leftarrow NL_3$.first_element; //the first element of $NL_3$;
(11) $ptr_2 \leftarrow ptr_1$.next_element; //the next element of $ptr_1$;
(12) while $ptr_1$ is not the last element of $NL_3$ do
(13)   if $ptr_1$.pre-code = $ptr_2$.pre-code and $ptr_1$.post-code = $ptr_2$.post-code
(14)     $ptr_1$. count $\leftarrow ptr_1$. count + $ptr_2$. count;
(15)     delete $ptr_2$ from $NL_3$;
(16)     $ptr_2 \leftarrow ptr_1$.next_element;
(17)   else
(18)     $ptr_1 \leftarrow ptr_2$;
(19)     $ptr_2 \leftarrow ptr_1$.next_element;
(20) return $NL_3$;
```

5. Experimental evaluation

In this section, we report our experiment results on the performance of the proposed algorithm and the baseline algorithms in terms of running time and memory consumption. Note that all these algorithms discover the same frequent itemsets, which confirms the result generated by any algorithm in the experiment is correct and complete.

5.1. Experiment Setup

In the experiments, we used six real datasets, which were often used in previous study of frequent itemset mining, for testing the performance of algorithms. These datasets were downloaded from FIMI repository (http://fimi.ua.ac.be). The pumsb dataset contains census data. The mushroom dataset contains characteristics of various kinds of mushrooms. The connect and chess datasets are originated from different game steps. The accidents dataset contains traffic accident data. The kosarak dataset contains click-stream data of an on-line news portal. Table 2 shows the characteristics of these datasets, where shows the average transaction length (denoted by #Avg.Length), the number of items (denoted by #Items) and the number of transactions (denoted by #Trans) in each dataset.

Table 2
The summary of the used datasets.

Dataset	Avg. Length	#Items	#Trans
Pumsb	74	7117	49,046
Mushroom	23	119	8124
Connect	43	129	67,557
Chess	37	75	3196
Accidents	33.8	468	340,183
Kosarak	8.1	41,270	990,002

We have compared the algorithm PrePost$^+$ with PrePost (Deng et al., 2012), FIN (Deng & Lv, 2014), and FP-growth* (Grahne & Zhu 2005). FP-growth* is a state-of-the-art FP-growth algorithm and is the winner of the FIMI03. Note that the previous studies (Deng et al. 2012; Grahne & Zhu 2005) show that PrePost and FP-growth* perform much better than dEclat, the state-of-the-art vertical mining algorithm. Therefore, no vertical mining algorithms are chosen as baseline algorithms.

PrePost$^+$, PrePost and FIN were all implemented in C++. The implementation of FP-growth* was downloaded from http://fimi.ua.ac.be/src/. All the experiments were performed on a computer with 16G memory and 2GHZ Intel processor. The operating system is X64 windows server 2003.

5.2. Visited nodes comparison of PrePost$^+$ and PrePost

Our first experiment is to compare the advantage of PrePost$^+$ versus PrePost in terms of pruning efficiency. By this experiment, we want to show that the pruning strategy adopted by PrePost$^+$, namely Children–Parent Equivalence pruning, is more efficient than the pruning strategy adopted by PrePost, which is based on single path property. For better illustrating the comparison, we use the number of nodes that PrePost$^+$ or PrePost has visited in the set-enumeration tree to represent the efficiency of their respective pruning strategies. Such nodes are called visited nodes. Clearly, the smaller the size of the visited nodes is, the better the pruning strategy is. Table 3 shows the details, where # Visited nodes means the number of visited nodes and *ratio* stands for (# visited nodes of PrePost) / (# visited nodes of PrePost$^+$). In the table, # visited nodes of PrePost is larger than # visited nodes of PrePost$^+$ on dataset pumsb, mushroom, connect, and chess while they are almost equal on dataset accidents and kosarak. Especially, # visited nodes of PrePost is two orders of magnitude bigger than that of PrePost$^+$ on dataset connect. The experimental results shown in Table 3 clearly substantiate the advantage of PrePost$^+$ in terms of pruning efficiency. Note that this advantage is key point that PrePost$^+$ performs better than PrePost on both runtime and memory consumption. In the following subsection, we will present the experimental results on runtime and memory consumption.

5.3. Comparison of runtime

In this subsection, we compare PrePost$^+$ against PrePost, FIN, and FP-growth* in terms of runtime. We have conducted substantial experiments spanning all six real datasets for various values of minimum support. Figs. 5–10 show the experimental results. Note that runtime here means the total execution time, which is the period between input and output.

Fig. 5 shows the time of all algorithms running on dataset pumsb. In the figure, PrePost$^+$ is the fastest one among all algorithms for each minimum support. FP-growth* is slower than PrePost and FIN for high minimum support. However, for low minimum support, FP-growth* becomes more efficient and is faster than PrePost and FIN. PrePost is always faster than FIN. For low minimum support, such as 50%, PrePost$^+$ is about an order of magnitude faster than FIN, outperforms PrePost by a factor of 3 and FP-growth* by a factor of 2.

Fig. 6 shows the runtime of all algorithms on dataset mushroom. In the figure, PrePost$^+$ is the fastest one for each minimum support while FP-growth* performs worst for all minimum supports. When the minimum support decreases, the advantage of PrePost$^+$ over PrePost and FIN becomes greater.

Fig. 7 shows the runtime of all algorithms on dataset connect. In the figure, PrePost$^+$ is once again the fastest one for each minimum support while PrePost becomes the lowest one. For low minimum

Table 3
The number of visited nodes: PrePost and PrePost⁺.

Pumsb		70%	65%	60%	55%	50%
	# visited nodes of PrePost	2,127,945	5,853,302	13,694,037	36,582,807	131,594,548
	# visited nodes of PrePost⁺	658,564	1,380,589	2,854,005	7,652,462	22,402,411
	Ratio	3.2	4.2	4.8	4.8	5.9
Mushroom		25%	20%	15%	10%	5%
	# visited nodes of PrePost	1,178	3,944	6,388	27,608	122,186
	# visited nodes of PrePost⁺	1,033	2,340	4,016	11,242	33,837
	Ratio	1.1	1.7	1.6	2.5	3.6
Connect		60%	55%	50%	45%	40%
	# visited nodes of PrePost	7,641,816	14,869,458	28,194,046	52,663,732	98,108,725
	# visited nodes of PrePost⁺	68,349	94,916	130,111	175,508	239,390
	Ratio	111.8	156.7	216.7	300.1	409.8
Chess		35%	30%	25%	20%	15%
	# visited nodes of PrePost	7,123,444	16,272,925	39,074,515	102,046,108	299,865,190
	# visited nodes of PrePost⁺	2,820,009	5,738,032	11,952,813	26,077,293	60,522,437
	Ratio	2.5	2.8	3.3	3.9	5.0
Accidents		10%	8%	6%	4%	2%
	# visited nodes of PrePost	10,071,850	19,527,551	42,905,928	117,023,470	527,992,103
	# visited nodes of PrePost⁺	9,959,665	19,124,698	41,276,662	108,908,139	459,916,837
	Ratio	1.0	1.0	1.0	1.1	1.1
Kosarak		1%	0.8%	0.6%	0.4%	0.2%
	# visited nodes of PrePost	278	425	758	1,681	24,158
	# visited nodes of PrePost⁺	278	425	758	1,681	24,158
	Ratio	1.0	1.0	1.0	1.0	1.0

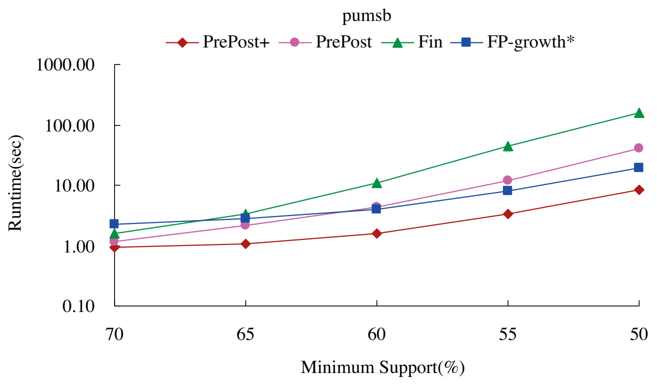

Fig. 5. Runtime on dataset pumsb.

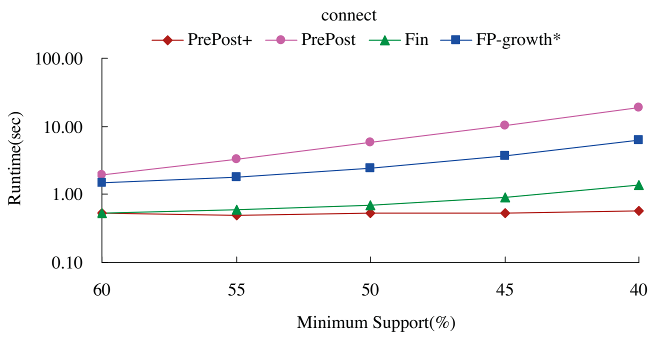

Fig. 7. Runtime on dataset connect.

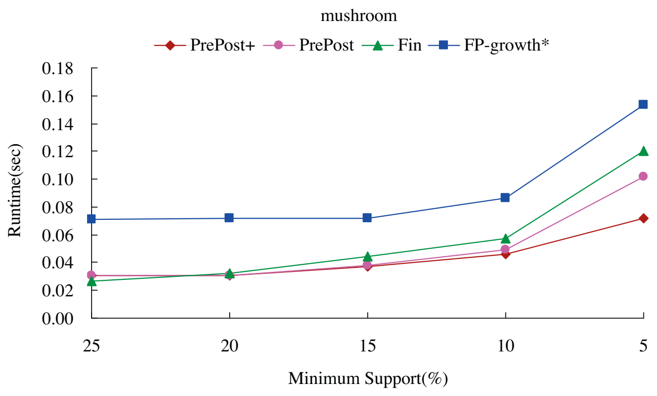

Fig. 6. Runtime on dataset mushroom.

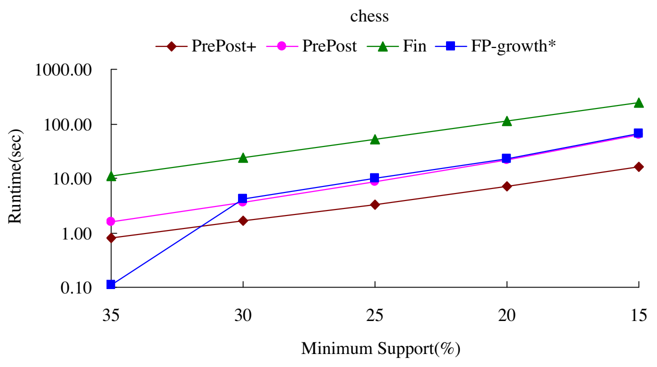

Fig. 8. Runtime on dataset chess.

support, PrePost⁺ runs much faster. For example, when the minimum support is set to 40%, PrePost⁺ is about an order of magnitude faster than PrePost and FP-growth* and outperforms FIN by a factor of 3.

The running results on dataset chess is shown by Fig. 8. PrePost⁺ still runs fastest for all minimum supports except the highest minimum support where FP-growth* runs fastest. FIN performs worst for each minimum supports. For dataset chess, PrePost⁺ is about an order of magnitude faster than FIN and outperforms PrePost and FP-growth* by a factor of 3 on average.

Fig. 9 shows the results on dataset accidents. In the feature, PrePost⁺ and PrePost perform best for each minimum support. Although PrePost⁺ is a little bit better than PrePost, their difference

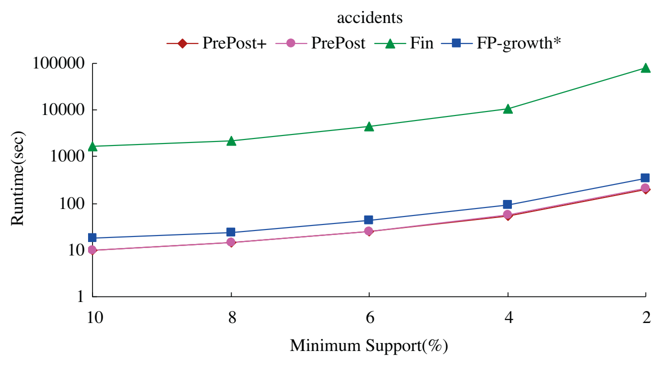

Fig. 9. Runtime on dataset accidents.

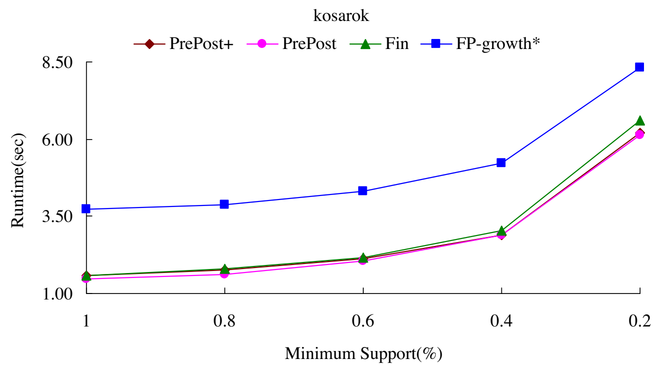

Fig. 10. Runtime on dataset kosarok.

is negligible. FIN runs lowest once again and is about two orders of magnitude lower than PrePost$^+$ and PrePost.

Fig. 10 shows the runtime of all algorithms on dataset kosarok. For dataset kosarok, PrePost$^+$, PrePost, and FIN have almost the same performance, though PrePost$^+$ and PrePost run a little bit faster than FIN. FP-growth* perform worst and is lower than other three algorithms by a factor of 2 on average.

Based on the above analysis, we find that PrePost$^+$ always performs best no matter which dataset is used and what the minimum threshold is. The reason can be explained as follows.

Compared with FP-growth*, PrePost$^+$ avoids the time consuming process of constructing a lot of conditional pattern bases and conditional FP-trees by simply N-list intersection. Our previous study (Deng et al., 2012) shows that N-list intersection is more efficient than the construction of conditional pattern base and conditional FP-tree. In addition, the Children–Parent Equivalence pruning strategy adopted by PrePost$^+$ is as efficient as the pruning strategy adopted by FP-growth*, which is based on single FP-tree path pattern generation property (Han et al., 2000).

Compared with PrePost, the Children–Parent Equivalence pruning strategy adopted by PrePost$^+$ is more efficient than the pruning strategy adopted by PrePost, which is based on single path property, especially on dense datasets. This can be validated by Table 3. Therefore, PrePost$^+$ avoids generating lots of redundant N-lists, which makes PrePost$^+$ more efficient than PrePost.

Compare with FIN, PrePost$^+$ employs a more efficient data structure, namely N-list, to representation itemsets. For validating this, we conduct an experiment comparing the size of N-lists versus Nodesets, which are adopted by FIN. Table 4 shows the average cardinality of the Nodesets and N-lists for frequent itemsets of various lengths on different dataset, for a given minimum support. Note that, in the table, Reduction Ration means the value of the

Table 4
Average Nodesets and N-list Cardinality.

Dataset	Minimum support (%)	Avg. N-list size	Avg. Nodeset size	Reduction ration
Pumsb	50	3	431	144
Mushroom	5	5	134	27
Connect	40	2	2	1
Chess	15	2	368	184
Accidents	2	3	6785	2261
Kosarak	0.2	728	2091	3

average N-list size divided by the average Nodeset size. We find N-lists are far smaller than Nodesets for most datasets.

5.4. Comparison of memory consumption

Figs. 11–16 show the peak memory consumption of all algorithms on six real datasets. In the figures, FP-growth* uses the lowest amount of memory on all datasets for each minimum threshold. Generally speak, PrePost$^+$ consumes less memory than the other algorithms except FP-growth*. Although PrePost$^+$ consumes a little more memory than FP-growth*, their difference is very small. In most cases, the memory consumed by PrePost$^+$ is less than 1.3 times the memory consumed by FP-growth*. However, PrePost$^+$ run much faster than FP-growth* on all datasets as discussed in Section 5.3.

Note that, the peak memory consumption concerns many factors that relates to the data distribution of the dataset, the pruning strategy employed by the algorithm, and the data structure used. Thus, it is very difficult to obtain analytically conclusion.

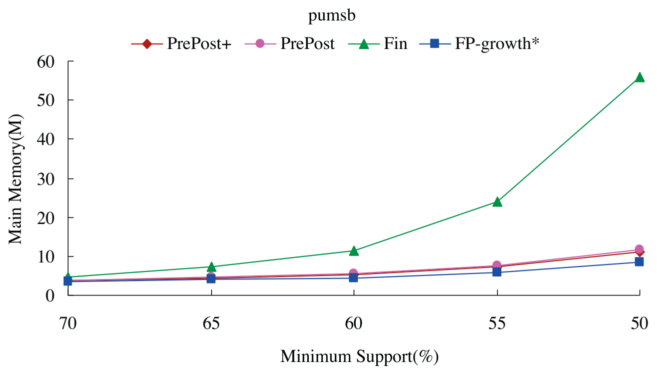

Fig. 11. Memory consumption on dataset pumsb.

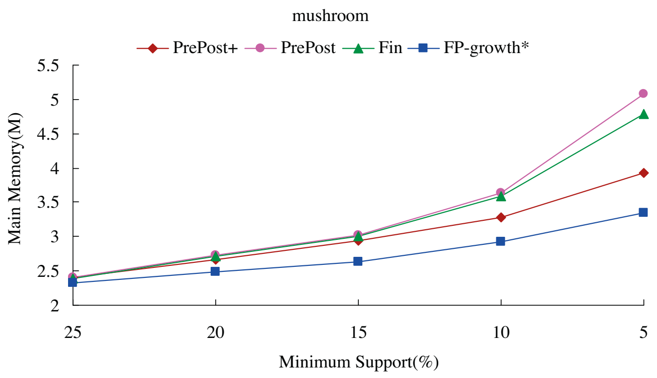

Fig. 12. Memory consumption on dataset mushroom.

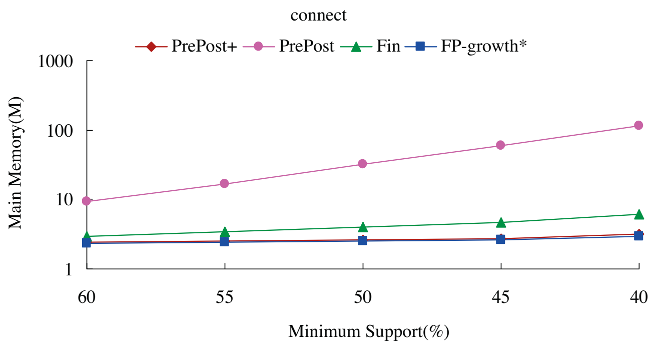

Fig. 13. Memory consumption on dataset connect.

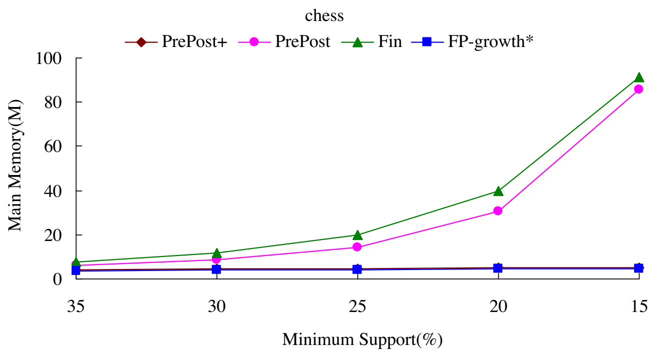

Fig. 14. Memory consumption on dataset chess.

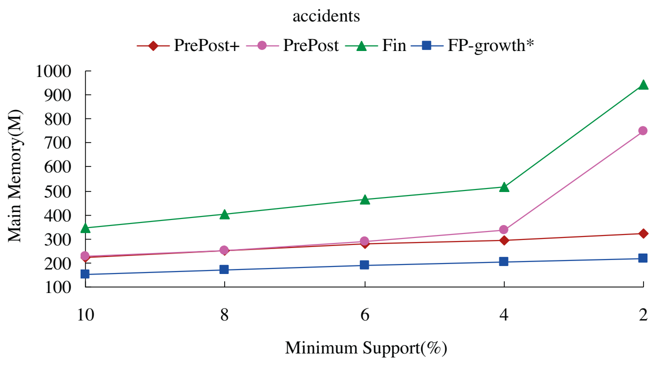

Fig. 15. Memory consumption on dataset accidents.

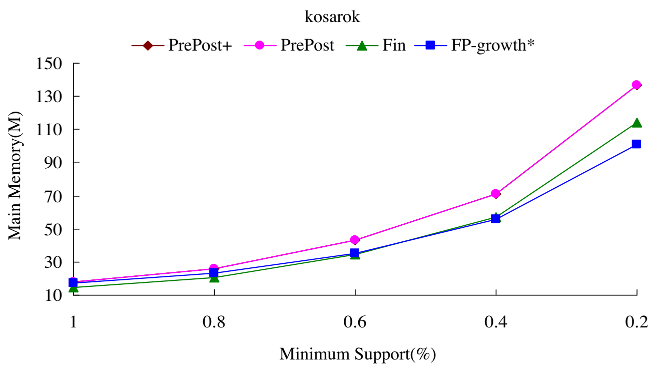

Fig. 16. Memory consumption on dataset kosarok.

Therefore, we focus on qualitative discussion instead of quantitative analysis in this subsection.

The reason that FP-growth* consumes less memory than the other algorithms can be explained as follows. All of PrePost$^+$, PrePost, and FIN must maintain both a PPC-tree and the representations of 2-itemsets, which consist of nodes in the PPC-tree. A PPC-tree is a little bigger than the corresponding FP-tree since the former contains additional information which is the code of each node. In addition, the number of 2-itemsets is usually large. Therefore, the peak memory consumption of PrePost$^+$, PrePost, and FIN is larger than that of FP-growth*.

As mentioned in Section 5.2, compared with PrePost, PrePost$^+$ visits fewer nodes in the set-enumeration tree. This means that PrePost$^+$ generates fewer intermediate itemsets. Therefore, PrePost$^+$ consumes less memory than PrePost. The bigger the *ratio* of the number of visited nodes of PrePost and that of PrePost$^+$ is, the greater the advantage of PrePost$^+$ over PrePost is in terms of memory consumption. For example, as shown in Table 3, the *ratio* is more than 100 on dataset connect for each minimum support. The memory consumption of PrePost$^+$ is one to two orders of magnitude smaller than that of PrePost on dataset connect as shown in Fig. 13. On dataset kosarak, the *ratio* is 1 and they consume almost the same memory as shown in Table 3 and Fig. 16. Note that, we can not distinguish the line for PrePost$^+$ and the line for PrePost in Fig. 16 since they consume almost the same memory.

Finally, the advantage of PrePost$^+$ over FIN in terms of memory consumption depends on average Nodesets and N-list Cardinality. The larger the reduction ration, mentioned in Table 4, is, the greater advantage of PrePost$^+$ over PrePost is in terms of memory consumption. On dataset accidents, the reduction ration is more than 2,000 for minimum support 2% as shown in Table 4. The memory consumption of PrePost$^+$ is about one third of that of PrePost for minimum support 2% as shown in Fig. 15. On dataset kosarak, the reduction ration is 3 for minimum support 0.2%. As shown in Fig. 16, PrePost uses less memory than PrePost$^+$.

6. Conclusions

In this paper, we present PrePost$^+$, a novel algorithm for mining frequent itemsets in databases. Besides adopting N-lists to represent itemsets, PrePost$^+$ employs an efficient pruning strategy named Children–Parent Equivalence pruning to greatly reduce the search space. An extensive set of experiments confirm that PrePost$^+$ is a state-of-the-art algorithm in terms of both runtime and memory consumption.

As shown in our experiments, although PrePost$^+$ runs fastest, it consumes more memory than FP-growth*. How to improve the memory consumption of PrePost$^+$ is an interesting research topic.

In recent years, probabilistic frequent itemsets (Aggarwal, Li, & Wang, 2009; Chui, Kao, & Hung, 2007; Tong, Chen, Cheng, & Yu, 2012) and high utility itemsets (Ahmed, Tanbeer, Jeong, & Choi, 2011; Lin, Tu, & Hsueh, 2012; Tseng, Wu, Shie, & Yu, 2010; Yin, Zheng, & Cao, 2012) have received a great deal of attention due to their wide applications. Mining probabilistic frequent itemsets and high utility itemsets has emerged as hot topics in the field of pattern mining. An interesting direction of future work is to extend PrePost$^+$ to develop efficient methods to mine such itemsets. In addition, since the available data is growing exponentially, employing PrePost$^+$ to mine frequent itemsets from big data is also an interesting work.

Acknowledgements

This work is partially supported by Project 61170091 supported by National Natural Science Foundation of China.

References

Aggarwal, C. C., Li, Y. & Wang, J. (2009). Frequent pattern mining with uncertain data. In: SIGKDD, pp. 29–38.

Agrawal, R., & Srikant, R. (1994). Fast algorithm for mining Association rules. In: VLDB, pp. 487–499.

Agrawal, R., Imielinski, T., & Swami, A. (1993). Mining Association Rules Between Sets of Items in Large Databases. In: SIGMOD, pp. 207–216.

Ahmed, C. F., Tanbeer, S. K., Jeong, B.-S., & Choi, H.-J. (2011). A framework for mining interesting high utility patterns with a strong frequency affinity. *Information Sciences, 181*, 4878–4894.

Baralis, E., Cerquitelli, T., & Chiusano, S. (2009). IMine: Index support for item set mining. *IEEE TKDE Journal, 21*(4), 493–506.

Burdick, D., Calimlim, M., Flannick, J., Gehrke, J., & Yiu, T. (2005). Mafia: A maximal frequent itemset algorithm. *IEEE TKDE Journal, 17*(11), 1490–1504.

Chui, C. K., Kao, B., & Hung, E. (2007). Mining frequent itemsets from uncertain data. In: PAKDD, pp. 47–58.

Deng, Z. H. (2014). Fast mining Top-Rank-k frequent patterns by using node-lists. *Expert Systems with Applications, 41*(4–2), 1763–1768.

Deng, Z. H., & Lv, S. L. (2014). Fast mining frequent itemsets using Nodesets. *Expert Systems with Applications, 41*(10), 4505–4512.

Deng, Z. H., & Wang, Z. H. (2010). A new fast vertical method for mining frequent itemsets. *International Journal of Computational Intelligence Systems, 3*(6), 733–744.

Deng, Z. H., Wang, Z. H., & Jiang, J. J. (2012). A new algorithm for fast mining frequent itemsets using n-lists. *Science China Information Sciences, 55*(9), 2008–2030.

Deng, Z. H., & Xu, X. (2012). Fast mining erasable itemsets using NC_sets. *Expert Systems with Applications, 39*(4), 4453–4463.

Grahne, G., & Zhu, J. (2005). Fast algorithms for frequent itemset mining using FP-trees. *IEEE TKDE Journal, 17*(10), 1347–1362.

Han, J., Cheng, H., Xin, D., & Yan, X. (2007). Frequent itemset mining: Current status and future directions. *DMKD Journal, 15*(1), 55–86.

Han, J., Pei, J., & Yin, Y. (2000). Mining frequent itemsets without candidate generation. In: SIGMOD, pp. 1–12.

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.

Le, T., Vo, B., & Coenen, F. (2013). An efficient algorithm for mining erasable itemsets using the difference of NC-sets. In: SMC, pp. 2270-2274.

Le, T., & Vo, B. (2014). MEI: An efficient algorithm for mining erasable itemsets. *Engineering Applications of Artificial Intelligence, 27*, 155–166.

Lin, M.-Y., Tu, T.-F., & Hsueh, S.-C. (2012). High utility pattern mining using the maximal itemset property and lexicographic tree structures. *Information Sciences, 215*, 1–14.

Liu, G., Lu, H., Lou, W., Xu, Y., & Yu, J. X. (2004). Efficient mining of frequent itemsets using ascending frequency ordered prefix-tree. *DMKD Journal, 9*(3), 249–274.

Pei, J., Han, J., Lu, H., Nishio, S., Tang, S., & Yang, D. (2001). H-mine: Hyper-structure mining of frequent itemsets in large databases. In: ICDM, pp. 441–448.

Savasere, A., Omiecinski, E. & Navathe, S. (1995). An efficient algorithm for mining association rules in large databases. In: VLDB, pp. 432–443.

Shenoy, P., Haritsa, J. R., Sundarshan, S., Bhalotia, G., Bawa, M., & Shah, D. (2000). Turbo-charging vertical mining of large databases. In: SIGMOD, pp. 22–33.

Tong, Y., Chen, L., Cheng, Y., & Yu, P. S. (2012). Mining frequent itemsets over uncertain databases. In: VLDB, pp. 1650–1661.

Tseng, V. S., Wu, C.-W., Shie, B.-E., & Yu, P. S. (2010). UP-Growth: an efficient algorithm for high utility itemset mining. In: KDD, pp. 253–262.

Vo, B., Coenen, F., Le, T., & Hong, T.-P. (2013). A hybrid approach for mining frequent itemsets. In: SMC, pp. 4647–4651.

Yin, J., Zheng, Z., & Cao, L. (2012). USpan: An efficient algorithm for mining high utility sequential patterns. In: KDD, pp. 660–668.

Zaki, M. J. (2000). Scalable algorithms for association mining. *IEEE TKDE Journal, 12*(3), 372–390.

Zaki, M. J. & Gouda, K. (2003). Fast vertical mining using diffsets. In: SIGKDD, pp. 326–335.