Expert Systems with Applications 41 (2014) 4505–4512

Contents lists available at ScienceDirect

Expert Systems with Applications

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

Fast mining frequent itemsets using Nodesets

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

Keywords:
Data mining
Frequent itemset mining
Nodesets
Algorithm
Performance

ABSTRACT

Node-list and N-list, two novel data structure proposed in recent years, have been proven to be very efficient for mining frequent itemsets. The main problem of these structures is that they both need to encode each node of a PPC-tree with pre-order and post-order code. This causes that they are memory-consuming and inconvenient to mine frequent itemsets. In this paper, we propose Nodeset, a more efficient data structure, for mining frequent itemsets. Nodesets require only the pre-order (or post-order code) of each node, which makes it saves half of memory compared with N-lists and Node-lists. Based on Nodesets, we present an efficient algorithm called FIN to mining frequent itemsets. For evaluating the performance of FIN, we have conduct experiments to compare it with PrePost and FP-growth*, two state-of-the-art algorithms, on a variety of real and synthetic datasets. The experimental results show that FIN is high performance on both running time and memory usage.

1. Introduction

Frequent itemset mining, first proposed by Agrawal, Imielinski, and Swami (1993), has become a fundamental task in the field of data mining because it has been widely used in many important data mining tasks such as mining associations, correlations, episodes, and etc. Since the first proposal of frequent itemset mining, hundreds of algorithms have been proposed on various kinds of extensions and applications, ranging from scalable data mining methodologies, to handling a wide diversity of data types, various extended mining tasks, and a variety of new applications (Han, Cheng, Xin, & Yan, 2007).

In recent years, we present two data structures called Node-list (Deng & Wang, 2010) and N-list (Deng, Wang, & Jiang, 2012) for facilitating the mining process of frequent itemsets. Both structures use nodes with pre-order and post-order to represent an itemset. Based on Node-list and N-list, two algorithms called PPV (Deng & Wang, 2010) and PrePost (Deng et al., 2012) are proposed, respectively for mining frequent itemsets. The high efficiency of PPV and PrePost is achieved by the compressed characteristic of Node-lists and N-lists. However, they are memory-consuming because Node-lists and N-lists need to encode a node with pre-order and post-order. In addition, the nodes' code model of Node-list and N-list is not suitable to join Node-lists or N-lists of two short itemsets to generate the Node-list or N-list of a long itemset. This may affect the efficiency of corresponding algorithms. Therefore, how to design an efficient data structure without pre-order and post-order is an interesting topic.

To this end, we present a novel data structure called Nodeset, for mining frequent itemsets. Different from Node-lists and N-lists, Nodesets require only the pre-order (or post-order code) of each node without the requirement of both pre-order and post-order. Based on Nodesets, we propose FIN, an efficient mining algorithm, to discover frequent itemsets. FIN directly discovers frequent itemsets in a search tree called set-enumeration tree (Rymon, 1992). For avoiding repetitive search, it also adopts a pruning strategy named promotion, which is similar to Children-Parent Equivalence pruning (Burdick, Calimlim, Flannick, Gehrke, & Yiu, 2005), to greatly reduce the search space. For evaluating the performance of FIN, we conduct a comprehensive performance study to compare it against PrePost and FP-growth*. The experimental results show that FIN is efficient on both running time and 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. Section 3 introduces the Nodeset structure and its basic properties. We describe FIN in Section 4. Experiment results are shown in Section 5 and conclusions are given in Section 6.

2. Related work

Formally, the task of frequent itemset mining can be described as follows. Let $I = \{i_1, i_2, \ldots, i_m\}$ be the universal item set and $DB = \{T_1, T_2, \ldots, T_n\}$ be a transaction database, where each T_k $(1 \leqslant k \leqslant n)$ is a transaction which is a set of items such that $T_k \subseteq I$.

* Corresponding author. Tel.: +86 1062755592.
E-mail address: zhdeng@cis.pku.edu.cn (Z.-H. Deng).

http://dx.doi.org/10.1016/j.eswa.2014.01.025
0957-4174/

P is called an itemset if P is a set of items. Let P be an itemset. A transaction T is said to contain 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 and $|DB|$ be the number of transactions in DB. An itemset P is frequent if its support is no less than $\xi \times |DB|$. Given a transaction database DB and a threshold ξ, the task of mining frequent itemsets is to find the set of all itemsets whose supports are 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 (Deng et al., 2012). The Apriori-like method is based on anti-monotone property (Agrawal & Srikant, 1994), called Apriori, which states that if any length k itemset is not frequent, its length $(k+1)$ super-itemset also cannot be frequent. The Apriori-like methods employ candidate-set-generation-and-test strategy to discover frequent itemsets. That is, it generates candidate length $(k+1)$ itemsets in the $(k+1)$th pass using frequent length k itemsets generated in the previous pass, and counts the supports of these candidate itemsets in the database. A lot of studies, such as (Agrawal & Srikant, 1994; Savasere, Omiecinski, & Navathe, 1995; Shenoy et al., 2000; Zaki, 2000; Zaki & Gouda, 2003), adopt the Apriori-like method. The Apriori-like method achieves good performance by reducing the size of candidates. However, previous studies reveal that it is highly expensive for Apriori-like method to repeatedly scan the database and check a large set of candidates by itemset matching (Han et al., 2007).

Different from the Apriori-like method, the FP-growth method mines frequent itemsets without candidate generation and has proven very efficient. The FP-growth method achieves impressive efficiency by adopting a highly condensed data structure called FP-tree (frequent itemset tree) to store databases and employing a partitioning-based, divide-and-conquer approach to mine frequent itemsets. Some studies, such as (Grahne & Zhu, 2005; Han, Pei, & Yin, 2000; Liu, Lu, Lou, Xu, & Yu, 2004; Pei et al., 2001), adopt the FP-growth method. The FP-growth method wins an advantage over the Apriori-like method by reducing search space and generating frequent itemsets without candidate generation. However, the FP-growth method only achieves significant speedups at low minimum supports because the process of constructing and using the FP-trees is complex (Woon, Ng, & Lim, 2004). In addition, recurrently building conditional itemset bases and trees makes the FP-growth method inefficient when datasets are sparse (Deng et al., 2012).

In recent years, we propose N-list (Deng et al., 2012) and Node-list (Deng & Wang, 2010), two novel data structures, to represent itemsets. Both of the two structures are based on a tree structure called PPC-tree, which store the sufficient information about frequent itemsets. A N-list or Node-list is a sorted set of nodes in the PPC-tree. Usually, the nodes in a N-list or Node-list are sorted by the ascending order of the pre-order of nodes. Their main difference is that Node-lists are built by descendant nodes while N-lists are built by ancestor nodes. N-lists or Node-lists have two important properties. The first one is that the N-list or Node-list of a length $(k+1)$ itemset can be constructed by joining the N-lists or Node-lists of its subset with length of k. The other one is that the support of an itemset is the sum of counts registering in the nodes of its N-list or Node-list. The high efficiency of PrePost and PPV, which based on N-lists and Node-lists, respectively, is achieved by these two properties. Extensive experiments show that PrePost and PPV are about an order of magnitude faster than state-of-the-art Apriori-like algorithm, such as dEclat and eclat_goethals, and are not slower than the algorithms based on FP-growth (Deng & Wang, 2010; Deng et al., 2012). Compared with Node-lists, N-lists have two advantages. The first one is that the length of the N-list of an itemset is much smaller than the length of its Node-list. The other one is that N-lists have property called single path property. The first advantage makes the time for joining two N-lists is much shorten than that for joining two Node-lists. This causes the efficiency of PrePost is higher than that of PPV. In addition, the single path property of N-lists is employed by PrePost to directly mine frequent itemsets without generating candidates in some cases while PPV must generate and test all candidates as the Apriori-like algorithms do. Therefore, PrePost is more efficient than PPV (Deng et al., 2012). Since 2012, NC_set (Deng & Xu, 2012), a structure similar to N-list and Node-list, has been proposed to mine erasable itemsets (Deng, Fang, Wang, & Xu, 2009) and the experimental results show that NC_set is very efficient (Deng & Xu, 2012; Le & Vo, 2014; Le, Vo, & Coenen, 2013).

Although N-list and Node-list are efficient structures for mining frequent itemsets, they 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. In this paper, we propose Nodeset, a novel structure, where a node is encoded only by pre-order or post-order code. Nodesets deal with the problem inherent in N-list and Node-list and is proven to be more efficient by extensive experiments. To the best of our knowledge, No such structure has been proposed in previous work.

3. Basic principles

In this section, we introduce relevant concepts and properties about Nodesets. We adopt some notations used in (Deng & Wang, 2010; Deng et al., 2012). For more details, please refer to (Deng & Wang, 2010; Deng et al., 2012). Note that, for simplicity, we denote an itemset of length k as a k-itemset in this paper.

3.1. POC-tree definition

Because Nodesets are based on a POC-tree, we first introduce the definition of POC-tree in brief. Here, POC-tree is the abbreviation of Pre-Order Coding tree.

Definition 1. POC-tree is a tree structure:

(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*. *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.

According to Definition 1, the structure of POC-tree is almost the same as the structure of PPC-tree (Deng & Wang, 2010; Deng et al., 2012). The only difference of these two kinds of tree lie in that each node of POC-tree is encoded by its pre-order while each node of PPC-tree is encoded by both its pre-order and its post-order. In a POC-tree, the *pre-order* of a node is determined by a pre-order traversal of the tree. In other word, the *pre-order* records the time when node N is accessed during the pre-order traversal. A POC-tree is only used to generate the Nodesets of frequent 2-itemsets. Later, we will find that after building these Nodesets, the POC-tree is useless and can be deleted. In fact, we can also use the post-order to encode each node and build a similar tree to generate the Nodesets. That is, the post-order is equivalent to the pre-order for our method. For simplicity, we use the pre-order in this paper.

Based on Definition 1, we have the following POC-tree construction algorithm.

Algorithm 1 (POC-tree Construction)

Input: A transaction database *DB* and a minimum support ξ.

Output: A POC-tree and F_1 (the set of frequent 1-itemsets).

1. [Frequent 1-itemsets Generation]
   According to ξ, scan *DB* once to find F_1, the set of frequent 1-itemsets (frequent items), and their supports. Sort F_1 in support descending order as L_1, which is the list of ordered frequent items. Note that, if the supports of some frequent items are equal, the orders can be assigned arbitrarily.
2. [POC-tree Construction]
   The following procedure of construction POC-tree is the same as that of constructing a FP-tree (Han, Pei, & Yin, 2000).
   Create the root of a POC-tree, *Tr*, and label it as "null".
   For each transaction *Trans* in DB do the following.
   Select the frequent items in *Trans* and sort out them according to the order of F_1. Let the sorted frequent-item list in *Trans* be $[p \mid P]$, where p is the first element and P is the remaining list. Call insert tree $([p \mid P], Tr)$.
   The function insert tree($[p \mid P]$, *Tr*) is performed as follows.
   If *Tr* has a child *N* such that *N.item-name* = *p.item-name*, then increase *N*'s count by 1;
   else create a new node *N*, with its count initialized to 1, and add it to *Tr*'s *children-list*. If *P* is nonempty, call insert tree(*P*, *N*) recursively.
3. [Pre-code Generation]
   Scan the POC-tree to generate the *pre-order* of each node by the pre-order traversal.

Note that Algorithm 1 is the same as the construction algorithm of PPC (Deng et al., 2012) except the third part of Pre-code Generation. For better understanding the concept and the construction algorithm of POC-tree, let's examine the following example.

Example 1 (*4*). Let the transaction database, *DB*, be represented by the information from the left two columns of Table 1 and $\xi = 0.4$. The frequent 1-itemsets set $F_1 = \{a, b, c, e, f\}$.

Fig. 1 shows the POC-tree which is constructed from the database shown in Example 1 after executing Algorithm 1. The number outside of a node is the pre-order of the node. In fact, the pre-order of a node is its identification. Note that the POC-tree is constructed using the right most column of Table 1 in Algorithm 1. Obviously, the second column and the last column are equivalent for mining frequent itemsets under the given minimum support. In the rightmost columns of Table 1, all infrequent items are eliminated and frequent items are listed in support-descending order. This ensures that the *DB* can be efficiently represented by a compressed tree structure.

3.2. Nodesets: definitions and properties

In this section, we will present some relevant concepts and the definition of Nodesets and then introduce some important properties of Nodesets, which facilitate the task of mining frequent itemsets. We first define N-info, which is the basic component of Nodeset.

Definition 2 (*N-info*). For a node *N* in a POC-tree, we call the pair of its pre-order and the count registering in it, (*pre-order*, *count*), the N-info of *N*.

Table 1
A transaction database.

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

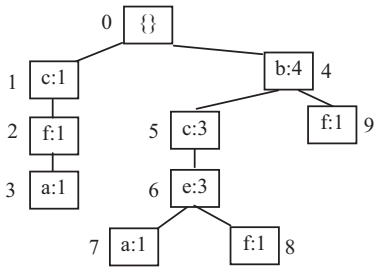

Fig. 1. The POC-tree after running Algorithm 1 on the database shown in Example 1.

Definition 3 (*Nodesets of items*). Given a POC-tree, the Nodeset of frequent item *i* is a sequence of all the N-infos of nodes registering *i* in the POC-tree.

Fig. 2 shows the Nodesets of all frequent items in Example 1.

Property 1. *Given a frequent item i, assume its Nodeset is* $\{(pre_1 : c_1), (pre_2, c_2), \ldots, (pre_m, c_m)\}$. *The support of i is equal to* $c_1 + c_2 + \ldots + c_m$.

Rationale. According to the definition of N-infos, each N-info corresponds to a node in the POC-tree, whose *count* registers the number of transactions including item *i*. Therefore, the sum of *counts* of nodes registering item *i* is equal to *i*'s *support*. □

For example, the Nodeset of *f* is $\{(2,1), (8,1), (9,1)\}$. According to Property 1, the support of *f* is 3 (1 + 1 + 1).

For the sake of discussion, we use the sorted set of frequent items in Algorithm 1, L_1, to define $\succ$ relation of two frequent items as follows.

Definition 4. ($\succ$ **relation)** For any two items i_1 and i_2 ($i_1, i_2 \in L_1$). $i_1 \succ i_2$ if and only if i_1 is ahead of i_2 in L_1.

For simplicity, an itemset *P* is denoted as $i_1 i_2 \ldots i_k$, where $i_1 \succ i_2 \succ \ldots \succ i_k$, in this paper.

We define the Nodeset of a 2-itemset as follows.

Definition 5 (*Nodesets of 2-itemsets*). Given any two items i_1 and i_2 ($i_1, i_2 \in L_1 \wedge i_1 \succ i_2$) and i_2's Nodeset is $\{(pre_1, c_1), (pre_2, c_2), \ldots, (pre_n, c_n)\}$, respectively. The Nodeset of 2-itemset $i_1 i_2$, denote as Nodeset($i_1 i_2$), is a subset of i_2's Nodeset, which is defined as follows:

$$
\begin{aligned}
b &\longrightarrow \{(4,4)\} \\
c &\longrightarrow \{(1,1), (5,3)\} \\
e &\longrightarrow \{(6,3)\} \\
f &\longrightarrow \{(2,1), (8,1), (9,1)\} \\
a &\longrightarrow \{(3,1), (7,1)\}
\end{aligned}
$$

Fig. 2. The Nodesets of frequent items in Example 1.

Nodeset(i_1i_2) = {(pre_k, c_k)| ∃ a node, N, registering i_1, N is an ancestor of the node corresponding to (pre_k, c_k)}

As shown in Fig. 2, the Nodeset of f is {(2,1), (8,1), (9,1)}. Let's see how to generate the Nodeset of bf. According to Definition 5, we first check whether (2,1) should be insert into the Nodeset of bf. Fig. 1 shows that only node 4 registers b. From Fig. 1, we find node 4 is not an ancestor of node 2, which is the corresponding node of (2,1). Therefore, we do not insert (2,1) into the Nodeset of bf. Similarly, we find node 4 is an ancestor of node 8 and 9, which are the corresponding node of (8,1) and (9,1), respectively. Thus, (8,1) and (9,1) are inserted into the Nodeset of bf. Up to now, all elements in the Nodeset of f are checked. Therefore, the Nodeset of bf is {(8,1), (9,1)}. By the same way, we have that the Nodeset of cf is {(2,1), (8,1)}.

Like the Nodesets of items, the Nodesets of 2-itemsets have a similar property as follows.

Property 2. *Let P (=p_1p_2,p_1,$p_2 \in L_1$ and $p_1 \succ p_2$) be a 2-itemset and assume its Nodeset is {(pre_1: c_1),(pre_2,c_2),...,(pre_m,c_m)}. The support of P is equal to $c_1 + c_2 + \ldots + c_m$.*

Rationale. Let T be a transaction that contains P. According to the construction algorithm of POC-tree (Algorithm 1), T must register p_1 to a node, denoted as N_1, and p_2 to a node, denoted as N_2. In addition, N_1 must be an ancestor of N_2. According to Definition 5, the N-info of N_2 must be one element in the Nodeset of P. For an element, (pre_j,c_j), let its corresponding node is N_j. According to Algorithm 1, c_j is the number of transactions that contain P and register p_2 on N_j. Therefore, the support of P is the sum of c_1,c_2,..., and c_m. □

For example, the Nodeset of bf is {(8,1), (9,1)}. According to Property 2, the support of f is 2 (1 + 1).

Based on Definition 5, we define the Nodesets of k-itemsets ($k \geqslant 3$) as follows.

Definition 6 (*Nodesets of k-itemsets*). Let $P = p_1p_2p_3\ldots p_k$ be an itemset ($p_j \in L_1$ and $p_1 \succ p_2 \succ p_3 \succ \ldots \succ p_k$). Assume the Nodeset of $P_1 = p_1p_3\ldots p_k$ is Nodeset_{P1} and the Nodeset of $P_2 = p_2p_3\ldots p_k$ is Nodeset_{P2}. The Nodeset of P, denoted as Nodeset_P, is defined as intersection of Nodeset_{P1} and Nodeset_{P2}. That is, $\text{Nodeset}_P = \text{Nodeset}_{P1} \cap \text{Nodeset}_{P2}$.

For example, we know that the Nodesets of bf and cf are {(8,1),(9,1)} and {(2,1),(8,1)}, respectively. According to Definition 6, the Nodeset of bcf is {(8,1),(9,1)} ∩ {(2,1),(8,1)} = {(8,1)}.

For the Nodesets of k-itemsets, similar property likes Property 1 and 2 still holds.

Property 3. *Given k-itemset P, assume its Nodeset is {(pre_1: c_1),(pre_2,c_2),...,(pre_m,c_m)}, the support of P is equal to $c_1 + c_2 + \ldots + c_m$.*

Rationale. This Property can be proved by the similar way used in the proof of Property 2. Let P be denoted as $p_1p_2p_3\ldots p_k$. For any transaction T containing P. it must register p_1,p_2,p_3,..., and p_k to a series of node, N_1, N_2,N_3,..., and N_k, respectively according to Algorithm 1. In addition, N_s must be an ancestor of N_t if s is less than t. By repeatedly employing Definition 6 and 5, we know the N-info of N_k must be one element in the Nodeset of P. For an element, (pre_j,c_j), let its corresponding node is N_j. According to Algorithm 1, c_j is the number of transactions that contain P and register p_k on N_j. Therefore, the support of P is the sum of c_1, c_2, ..., and c_m. □

Note that, for the sake of high efficiency of intersection operation, the elements (N-infos) in a nodeset are sorted by the pre-order ascending order in this paper. As is known to all, the complexity of intersection of two ordered sets is linear.

4. Fin: the proposed method

The framework of FIN consists of: (1) Construct the POC-tree and identify all frequent 1-itemsets; (2) scan the POC-tree to find all frequent 2-itemsets and their Nodesets; (3) mine all frequent k(>2)-itemsets. For enhance the efficiency of mining frequent itemsets, FIN adopts promotion, which is based on superset equivalence property, as pruning strategy.

For facilitating the mining process, FIN employs a set-enumeration tree (Rymon, 1992) 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 $\{i_{js}i_{js-1}\ldots i_{j1}\}$ and registering i_{js}, the $(m - j_s)$ child nodes of the node representing itemsets $\{i_{js+1}i_{js}i_{js-1}\ldots i_{j1}\},\{i_{js+2}i_{js}i_{js-1}\ldots i_{j1}\},\ldots,\{i_mi_{js}i_{js-1}\ldots i_{j1}\}$ and registering i_{js+1}, i_{js+2}, ..., 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 is represented in Fig. 3. For example, the node in the bottom left of Fig. 4 represents itemset {$bceaf$} and registers item b.

Property 4 (*superset equivalence*). *Given item i and itemset P ($i \notin P$), if the support of P is equal to the support of $P \cup \{i\}$, the support of $A \cup P$, where $A \cap P = \varnothing \wedge i \notin A$, is equal to the support of $A \cup P \cup \{i\}$.*

Rationale. That the support of P is equal to the support of $P \cup \{i\}$ indicates that any transaction containing P also contains i. Given a transaction T, if T contains $A \cup P$, it must contain P. Therefore, we know that T also contains i. That is, the support of $A \cup P$ is equal to the support of $A \cup P \cup \{i\}$. □

FIN employs Property 4 to narrow the search space greatly. Let's take Fig. 3 as an example. If we find the support of af is equal to the support of eaf, the subtree, whose root is the node registering e and representing eaf in the left of Fig. 3, will be pruned in the mining process.

Algorithm 2 shows the pseudo-code of FIN. Line (1) initializes F, which is used to store frequent itemsets, by setting it to be null. Line (2) constructs the POC-tree and finds F_1, the set of all frequent

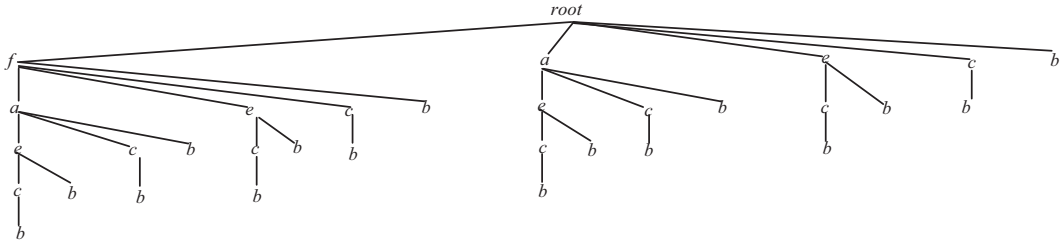

Fig. 3. An example of set-enumeration tree.

1-itemset, by calling Algorithm 1. Line (3) initializes F_2, which is used to store frequent 2-itemsets, by setting it to be null. Line (4)–(15) insert all candidate frequent 2-itemsets in F_2, by scanning the POC-tree with the pre-order traversal. Line (16)–(22) delete all infrequent 2-itemsets from F_2, as Line (18) do, and initialize the Nodesets of all frequent 2-itemsets by setting them to be null, as Line (20) do. Line (23)–(31) generate the Nodesets of all frequent 2-itemsets by scanning the POC-tree with the pre-order traversal. Line (33)–(36) generate all frequent k-itemsets ($k \geqslant 3$) by calling Procedure ***Constructing_Pattern_Tree*** () to generate all frequent k-itemsets ($k \geqslant 3$) extended from each frequent 2-itemset.

Algorithm 2: FIN Algorithm

Input: A transaction database *DB* and a minimum support ξ.
Output: *F*, the set of all frequent itemsets.
(1) $F \leftarrow \varnothing$;
(2) Call Algorithm 1 to construct the POC-tree and find F_1, the set of all frequent 1-itemset;
(3) $F_2 \leftarrow \varnothing$;
(4) Scan the POC-tree by the pre-order traversal **do**
(5) $N \leftarrow$ currently visiting Node;
(6) $i_y \leftarrow$ the item registered in N;
(7) **For** each ancestor of N, N_a, **do**
(8) $i_x \leftarrow$ the item registered in N_a;
(9) **If** $i_x i_y \in F_2$, **then**
(10) $i_x i_y.support \leftarrow i_x i_y.support + N.account$;
(11) **Else**
(12) $i_x i_y.support \leftarrow N.account$;
(13) $F_2 \leftarrow F_2 \cup \{i_x i_y\}$;
(14) **Endif**
(15) **Endfor**
(16) **For** each itemset, P, in F_2 **do**
(17) **If** $P.support < \xi \times |DB|$, **then**
(18) $F_2 \leftarrow F_2 - \{P\}$;
(19) **Else**
(20) $P.Nodeset \leftarrow \varnothing$;
(21) **Endif**
(22) **Endfor**
(23) Scan the POC-tree by the pre-order traversal **do**
(24) $Nd \leftarrow$ currently visiting Node;
(25) $i_y \leftarrow$ the item registered in Nd;
(26) **For** each ancestor of Nd, Nd_a, **do**
(27) $i_x \leftarrow$ the item registered in Nd_a;
(28) **If** $i_x i_y \in F_2$, **then**
(29) $i_x i_y.Nodeset \leftarrow i_x i_y.Nodeset \cup Nd.N\_info$;
(30) **Endif**
(31) **Endfor**
(32) $F \leftarrow F \cup F_1$;
(33) **For** each frequent *itemset*, $i_s i_t$, in F_2 **do**
(34) Create the root of a tree, R_{st}, and label it by $i_s i_t$;
(35) ***Constructing_Pattern_Tree***$(R_{st}, \{i \mid i \in F_1, i \succ i_s\}, \varnothing)$;
(36) **Endfor**
(37) **Return** *F*;

Procedure ***Building_Pattern_Tree*** () employ Property 4 to pruning the search space. *Nd, Cad_set, ex_frequent_itemsets* are three input parameters. *Nd* stands for the current node in the set-enumeration tree. *Cad_set* are available items that are used to extend Node *Nd*. In fact, *Cad_set* are used to generate child nodes of *Nd*. *FIS_parent* are the frequent itemsets generated on the parent of *Nd*. Line (4)–(19) check each item in *Cad_set* to find the promoted items and the items that will be used to construct the child nodes of *Nd*. Line (9) and (10) inserts the promoted items into *Nd.equivalent_items*. An item, *i*, is called promoted if the support of $\{i\} \cup Nd.itemset$ is equal to the support of *Nd.itemset*. Because all information about the frequent itemsets relevant to the promoted items is stored in *Nd*, we don't not need to use the promoted items to further generate the child nodes (actually, subtrees) for discovering frequent itemsets. This pruning technique is called promotion. In fact, identifying the promoted items is the main pruning strategy of FIN. Line (11)–(16) find all items with which the extension of *Nd.itemset* are frequent. These items are stored in *Next_Cad_ set* for the next procedure, which generates the child nodes of *Nd*. Line (21)–(27) discover all frequent itemsets on *Nd*. In fact, this is done by Line (24) or Line (26) according to *FIS_parent*. If *FIS_parent* is null, *PSet* is the set of all frequent itemsets on *Nd*. Otherwise, the itemsets, which are generated by *PSet* and *FIS_parent* as Line (26) does, are all frequent itemsets on on *Nd*. *FIT_Nd* stores these frequent itemsets for the future procedure of constructing the child nodes of *Nd*. Line (30)–(34) continue to extend the child nodes of *Nd* by recursively calling ***Building_Pattern_Tree*** ().

Procedure *Constructing_Pattern_Tree* (*Nd, Cad_set, FIS_parent*)
(1) $Nd.equivalent\_items \leftarrow \varnothing$;
(2) $Nd.childnodes \leftarrow \varnothing$;
(3) $Next\_Cad\_ set \leftarrow \varnothing$;
(4) **For** each $i \in Cad\_set$ **do**
(5) $X \leftarrow Nd.itemset$;
(6) $Y \leftarrow \{i\} \cup (X - X[1])$;
(7) $P \leftarrow \{i\} \cup X$;
(8) $P.Nodeset \leftarrow X.Nodeset \cap Y.Nodeset$;
(9) **If** $P.support = X.support$ **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) $Next\_Cad\_ set \leftarrow Next\_Cad\_ set \cup \{i\}$;
(17) **Endif**
(18) **Endif**
(19) **Endfor**
(20) **If** $Nd.equivalent\_items \neq \varnothing$ **then**
(21) $SS \leftarrow$ the set of all subsets of $Nd.equivalent\_items$;
(22) $PSet \leftarrow \{A \mid A = Nd.label \cup A', A' \in SS\}$;
(23) **If** $FIS\_parent = \varnothing$, **then**
(24) $FIT\_Nd \leftarrow PSet$;
(25) **Else**
(26) $FIT\_Nd \leftarrow \{P' \mid P' = P_1 \cup P_2, (P_1 \neq \varnothing \wedge P_1 \in PSet)$ and $(P_2 \neq \varnothing \wedge P_2 \in FIS\_parent\}$;
(27) **Endif**
(28) $F \leftarrow F \cup FIT\_Nd$;
(29) **Endif**
(30) **If** $Nd.childnodes \neq \varnothing$ **then**
(31) **For** each $Nd_i \in Nd.childnodes$ **do**
(32) ***Constructing_Pattern_Tree***$(Nd_i, \{j \mid j \in Next\_Cad\_ set, j \succ i\}, FIT\_Nd)$;
(33) **Endfor**
(34) **Else *return***;
(35) **Endif**

5. Experimental evaluation

In this section, we report two sets of experiment results in which running time and memory consumption of FIN are com-

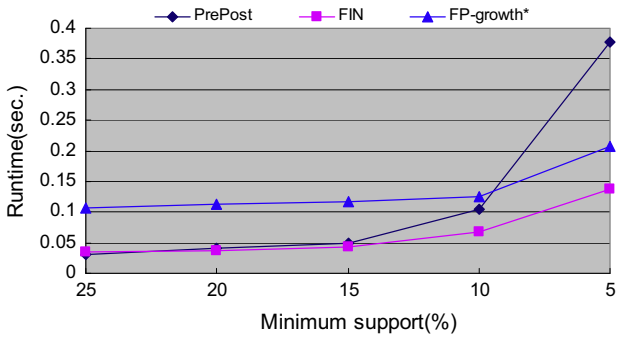

Fig. 4. Running time on Mushroom.

pared with two state-of-the-art algorithms, PrePost and FP-growth*. Note that all these algorithms discover the same frequent itemsets, which confirms the result generated by any algorithms in our experiments is correct and complete.

5.1. Experiment setup

To test the performance of FIN on different environments with various data distributions, we used two real datasets and one synthetic dataset in the experiments. The three datasets are Mushroom, Connect, and T25I10D100 K, which were often used in previous study of frequent itemset mining. The Mushroom and Connect datasets are real datasets and are downloaded from FIMI repository (http://fimi.ua.ac.be). The mushroom dataset contains characteristics of various species of mushrooms while the connect dataset is derived from game steps. The T25I10D100 K dataset is a synthetic dataset and was generated by the IBM generator (http://www.almaden.ibm.com/cs/quest/syndata.html). To generate T25I10D100 K, the average transaction size and average maximal potentially frequent itemset size are set to 25 and 10, respectively, while the number of transactions in the dataset and different items used in the dataset are set to 100 K and 1 K, respectively.

Note that, these two real datasets are very dense. For example, when the minimum support is set to 5%, the number of frequent itemsets discovered from the Mushroom dataset is more than 3 millions. The synthetic datasets generated by the IBM generator mimic the transactions in a retailing environment. Therefore, the synthetic datasets are usual much sparser when compared to the real sets. 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.

We choose PrePost and FP-growth* as the baseline algorithms. PrePost has proven to be the best algorithm among all node-based methods (Deng et al., 2012). FP-growth* is the best algorithm among FP-tree-based methods (Grahne & Zhu, 2005) and is the winner of FIMI 2003. Both FIN and PrePost are implemented in C++. The implementation of FP-growth* in C++ was downloaded from http://fimi.cs.helsinki.fi/src/. All the experiments are performed on a computer with 14G memory and Intel Xeon @2.0GHZ processor. The operating system is Windows Server 2003 Standard x64 Edition.

5.2. Comparison of running time

The runtime comparison of FIN against PrePost and FP-growth* is shown in Figs. 4–6, where the *X* and *Y* axes stand for minimum support and running time, respectively. Note that, running time here means the total execution time, which is the period between input and output. We conduct a thorough sets of experiments

Table 2
The summary of the used datasets.

Database	Avg. length	#Items	#Trans
Mushroom	23	119	8124
Connect13	43	130	67,557
T25I10D100 K	25	990	99,822

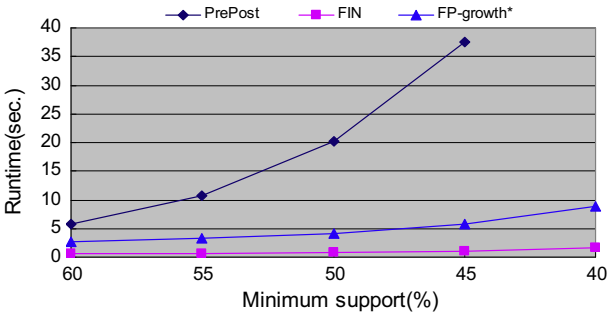

Fig. 5. Running time on Connect.

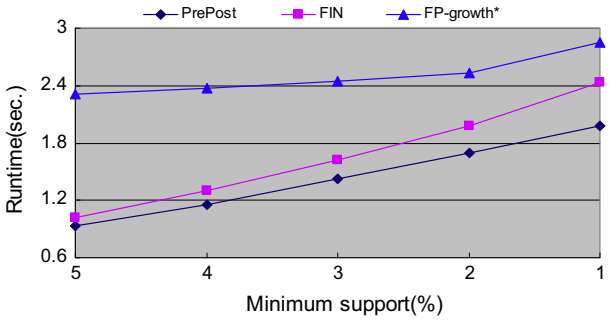

Fig. 6. Running time on T25I10D100 K.

Table 3
Memory usage comparison on Mushroom.

Minimum support (%)	25	20	15	10	5
Memory usage ratio Mem(PrePost)/Mem(FIN)	1	1.37	1.53	3.42	8.98

Table 4
Memory usage comparison on Connect.

Minimum support (%)	60	55	50	45	40
Memory usage ratio Mem(PrePost)/Mem(FIN)	122	185	271	386	N/A

spanning all the real and synthetic datasets mentioned above with various values of minimum support.

Fig. 4 shows the running time of all algorithms on the Mushroom dataset. In the figure, FIN is faster than PrePost and FP-growth* for each minimum support. For high minimum support ($\geqslant$10%), PrePost is faster than FP-growth*. However, FP-growth* become to be faster than PrePost when the minimum support is less than 10%. We observe that when the minimum support become small, the runtime of PrePost increase faster than that of FIN and FP-growth* while FIN has the same growth trend as FP-growth*. This means the scalability of PrePost is worse than FIN and FP-growth*. The results shown in Fig. 4 can be explained as fol-

Table 5
Memory usage comparison on T25I10D100 K.

Minimum support (%)	5	4	3	2	1
Memory usage ratio Mem(PrePost)/Mem(FIN)	1.05	1.05	1.05	1.05	1.05

lows: with dense datasets, the single path property employed by PrePost is not good enough to narrow search space so that PrePost need to generate too much candidates. FP-growth* directly discovery frequent itemsets without generating candidates so that it scales well with low minimum supports (many frequent itemsets). The promotion strategy adopted by FIN works well on the Mushroom dataset. We find the level range of nodes which occurs promotion in FIN is from 2 to 3 when the minimum threshold changes from 25% to 5%. This means that FIN can trim the search space at high level and thus avoid much redundant computation. However, the level range of nodes which occurs promotion in PrePost is from 3 to 5 when the minimum threshold changes from 25% to 5%. This means PrePost trim the search space at low level when compared with FIN. That is, PrePost is not as efficient as FIN in terms of pruning strategy. However, the N-lists are shorter than Nodesets. This means that the time used for generating N-lists and computing supports in PrePost is shorter than that in FIN. Therefore, PrePost and FIN have the same efficiency when the minimum threshold is high. However, when the minimum threshold is low, the advantage of pruning strategy of FIN is much bigger than the advantage of computing N-lists and supports of PrePost. Therefore, FIN is more efficient than PrePost on the whole.

Fig. 5 shows the runtime of all algorithms on the other dense dataset, Connect. Once again, FIN is the fastest algorithm for each minimum support as shown in Fig. 5. FP-growth* consistently outperforms PrePost. PrePost performs worst and can not find all frequent itemsets in reasonable time (100 s) when the minimum threshold is 40%. The results shown in Fig. 5 can be explained by the discussion in the case of Fig. 4. On the connect dataset, we find that the average level of nodes which occurs promotion in PrePost is 9 while the average level of nodes which occurs promotion in FIN is 3. This means that PrePost trim the search space at much lower level on the Connect dataset than on the Mushroom dataset. Therefore, PrePost performs worse on the Connect dataset than on the Mushroom dataset.

Fig. 6 shows the runtime of all algorithms on the sparse dataset, T25I10D100 K, for different minimum supports, respectively. In Fig. 6, we observe that PrePost$^+$ runs fastest for each minimum support. FIN runs faster than FP-growth* consistently. This is quite different from the case observed in Figs. 4 and 5. The results shown in Fig. 4 can be explained as follows. On T25I10D100 K, we find that no promotion occurs in both FIN and PrePost. This means pruning strategy of FIN and FIN and PrePost is disabled. Therefore, computing N-lists or Nodesets is the main time cost. As mentioned above, computing N-lists is more efficient than computing Nodesets. Thus, PrePost outperforms FIN. Sparsity of datasets may be the reason that no promotion occurs. However, how to measure the quantitative relation is very hard and is out of the scope of this paper. The reason that FP-growth* performs worst for all minimum support can be explained as follows. T25I10D100 K is so sparse that the generated FP-trees are much bigger that those on dense datasets. Clearly, the bigger is the tree, the more is time cost.

5.3. Comparison of memory consumption

As for memory consumption, we adopt Memory Usage Ratio to measure the performance of FIN and PrePost. Memory Usage Ratio is defined as the ratio of the peak memory consumed by PrePost divided by the peak memory consumed by FIN. Tables 3–5 show the results.

From Tables 3 and 4, we observer that PrePost$^+$ consumes more memory than FIN. Specially, the memory usage of PrePost$^+$ is an order of magnitude bigger than that of FIN. As shown in Table 5, PrePost$^+$ and FIN consume the same memory on the T25I10D100 K dataset. The results can also be explained by the pruning strategy adopted by PrePost$^+$ and FIN. As stated in Section 5.2, FIN prunes more candidates in search space than PrePost$^+$ on the Mushroom and Connect dataset. Therefore, the number of candidates that needs to be stored in FIN is much less than that in PrePost$^+$. In fact, the number of candidates is directly proportional to the pruning efficiency. Thus, FIN consumes less memory than PrePost$^+$ on the Mushroom and Connect dataset. On the T25I10D100 K dataset, no pruning occurs for both algorithms. Therefore, the memory usage depend on the size of N-lists and Nodesets. On the one hand, the length of N-list of an itemset is shorter than the length of its Nodeset. Note that, the length of a N-lists (or Nodeset) is the number of elements contained in it. On the other hand, the size of elements in a N-list is bigger than that of elements in a Nodeset because N-list stores the pre-order and post-orde code of nodes where Nodesets stores only pre-order code (or post-order code). As we know, the size of a N-list (or Nodeset) is equal to the sum of size of all elements, which is directly proportional to the length of the N-list (or Nodeset) and the size of elements in it. Based on the above discussion, PrePost$^+$ and FIN consume almost the same memory on the T25I10D100 K dataset.

6. Conclusions

In this paper, we present a novel structure called Nodeset to facilitate the process of mining frequent itemsets. Based on Nodesets, an efficient algorithm called FIN is proposed to mine frequent itemsets in databases. The advantage of Nodeset lies in that it encodes each node of a POC-tree with only pre-order (or post-order). This causes that Nodesets consume less memory and are easy to be constructed. The extensive experiments show that the Nodeset structure is efficient and FIN run faster than PrePost and FP-growth* on the whole. Especially, FIN consumes much less memory than PrePost on dense datasets.

As future extensions of this work, first we will explore how to employ Nodesets to mine maximal frequent itemsets (19 Bayardo Jr, 1998; Burdick et al., 2005), closed frequent itemsets (Lee, Wang, Weng, Chen, & Wu, 2008; Wang, Han, & Pei, 2003), Top-rank-k frequent Patterns (Deng, 2014). Second, we will further extend Nodesets to make it suitable to mine frequent itemsets from data streams (Chang & Lee, 2003; Li & Deng, 2010). Finally, as big data become more and more popular in practice, the parallel/distributed implementation of Nodesets to mine frequent itemsets from huge dataset is also an interesting work.

Acknowledgements

This work is partially supported by Project 61170091 supported by National Natural Science Foundation of China. We are also grateful to the anonymous reviewers for their helpful comments.

References

Agrawal, R., & Srikant, R. (1998). Fast algorithm for mining association rules. In VLDB'94 (pp. 487–499).
Agrawal, R., Imielinski, T., & Swami, A. (1993). Mining association rules between sets of items in large databases. In SIGMOD'93 (pp. 207–216).

Bayardo Jr, R. J. (1998). Efficiently mining long itemsets from databases. In SIGMOD'98 (pp. 85–93).
Burdick, D., Calimlim, M., Flannick, J., Gehrke, J., & Yiu, T. (2005). Mafia: a maximal frequent itemset algorithm. *IEEE TKDE Journal, 17*(11), 1490–1504.
Chang, J. H., & Lee, W. S. (2003). Finding recent frequent itemsets adaptively over online data streams. In KDD'03 (pp. 487–492).
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., Fang, G. D., Wang, Z. H., & Xu, X. R. (2009). Mining erasable itemsets. In ICMLC'09 (pp. 67–73).
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. R. (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'00 (pp. 1–12).
Le, T., Vo, B., & Coenen, F. (2013). An efficient algorithm for mining erasable itemsets using the difference of NC-sets. In IEEE SMC'13 (pp. 2270–2274).
Le, T., & Vo, B. (2014). MEI: an efficient algorithm for mining erasable itemsets. *Engineering Applications of Artificial Intelligence, 27*, 155–166.
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 and Knowledge Engineering, 66*(1), 68–91.
Li, X., & Deng, Z. H. (2010). Mining frequent itemsets from network flows for monitoring network. *Expert Systems with Applications, 37*(12), 8850–8860.
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'01 (pp. 441–448).
Rymon, R. (1992). Search through systematic set enumeration. In Proc. Int'l Conf. principles of knowledge representation and reasoning (pp. 539–550).
Savasere, A., Omiecinski, E., & Navathe, S. (1995). An efficient algorithm for mining association rules in large databases. In VLDB'95 (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'00 (pp. 22–33).
Wang, J. Y., Han, J., & Pei, J. (2003). CLOSET+: searching for the best strate-gies for mining frequent closed itemsets. In SIGKDD'03 (pp. 236–245).
Woon, Y. K., Ng, W. K., & Lim, E. P. (2004). A support-ordered trie for fast frequent itemset discovery. *IEEE TKDE Journal, 16*(7), 875–879.
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'03 (pp. 326–335).