International Journal of Computational Intelligence Systems, Vol.3, No. 6 (December, 2010), 733-744

A New Fast Vertical Method for Mining Frequent Patterns

Zhihong Deng
Key Laboratory of Machine Perception (Ministry of Education), School of Electronics Engineering and Computer Science, Peking University
Beijing, 100871, China
E-mail: zhdeng@cis.pku.edu.cn

Zhonghui Wang
Key Laboratory of Machine Perception (Ministry of Education), School of Electronics Engineering and Computer Science, Peking University
Beijing, 100871, China
E-mail: wangzh@cis.pku.edu.cn

Received: 21-12-2009
Accepted: 08-11-2010

Abstract

Vertical mining methods are very effective for mining frequent patterns and usually outperform horizontal mining methods. However, the vertical methods become ineffective since the intersection time starts to be costly when the cardinality of tidset (tid-list or diffset) is very large or there are a very large number of transactions.

In this paper, we propose a novel vertical algorithm called PPV for fast frequent pattern discovery. PPV works based on a data structure called Node-lists, which is obtained from a coding prefix-tree called PPC-tree. The efficiency of PPV is achieved with three techniques. First, the Node-list is much more compact compared with previous proposed vertical structure (such as tid-lists or diffsets) since transactions with common prefixes share the same nodes of the PPC-tree. Second, the counting of support is transformed into the intersection of Node-lists and the complexity of intersecting two Node-lists can be reduced to $O(m+n)$ by an efficient strategy, where m and n are the cardinalities of the two Node-lists respectively. Third, the ancestor-descendant relationship of two nodes, which is the basic step of intersecting Node-lists, can be very efficiently verified by Pre-Post codes of nodes.

We experimentally compare our algorithm with FP-growth, and two prominent vertical algorithms (Eclat and dEclat) on a number of databases. The experimental results show that PPV is an efficient algorithm that outperforms FP-growth, Eclat, and dEclat.

Keywords: data mining; frequent pattern mining; data structure; algorithm

1. Introduction

Data mining (or knowledge discovery in databases, KDD) has attracted tremendous amount of attention in the database research community due to its wide applicability in many areas. Since mining frequent patterns was first introduced in [1], it has emerged as a fundamental problem in data mining and plays an essential role in many important data mining tasks such as associations, correlations, sequential patterns, particle periodicity, classification, etc [2].

Most of the previous proposed frequent pattern mining algorithms can be divided into two groups: the Apriori-like method and the FP-growth method. Apriori-like approach generates candidate patterns of length (k+1) in the (k+1)th pass using frequent patterns of length k generated in the previous pass, and counts the supports of these candidate patterns in the database. The idea of Apriori-like approach depends on an anti-monotone Apriori property [3]: all nonempty subset of a frequent pattern must also be frequent. A lot of studies, such as [3-8], adopt the Apriori-like approach. The FP-growth is a recently proposed method that has proved to be very efficient in mining frequent patterns. FP-growth achieves impressive efficiency by adopting a highly condensed data structure called frequent pattern tree to store databases and employing a partitioning-based, divide-and-conquer method to mine frequent patterns. Some studies, such as [2, 9, 10], adopt the FP-growth approach.

The Apriori-like approach achieves good performance by reducing the size of candidates. However, previous studies reveal that it is highly expensive for Apriori-like approach to repeatedly scan the database and check a large set of candidates by pattern matching [2]. In order to deal with these problems, a number of vertical mining algorithms have been proposed [6-8]. Unlike the traditional horizontal transactional database format used in most Apriori-like algorithms, each item in a vertical database is associated with its corresponding tid-list—the set of all transaction *id*s where it appears. The advantage of vertical database format is that the counting of supports of frequent patterns can be obtained via tid-list intersection, which avoids scanning a whole database. tid-list is much simpler than complex hash or trees used in horizontal algorithms and is also more efficient than them in counting supports of frequent patterns. Vertical mining methods have been shown to be very effective and usually outperform horizontal mining methods [8]. Despite the advantages of the vertical database format, the vertical methods become to be ineffective since the intersection time starts to be costly when tidset cardinality (such as for very frequent items) is very large or there are a very large number of transactions [8].

The FP-growth approach wins an advantage over the Apriori-like approach by reducing search space and generating frequent patterns without candidate generation. However, FP-growth only achieves significant speedups at low support thresholds because the process of constructing and using the frequent pattern trees is complex [11]. In addition, since FP-growth generates frequent patterns by recursively mining conditional frequent pattern trees, it tends to need a large number of memories to store these temporal pattern trees. Though [10] proposed a disk-based method called database projection to address the above problem, obviously, it will damage the performance because of frequently accessing disks.

One advantage of FP-growth approach is the frequent pattern tree, which is a highly condensed data structure for storing the database. On the other hand, the biggest advantage of vertical mining algorithm is that each item is represented by transaction *id*s (TID). A question is that can we integrate the advantages of the two approaches and form a new efficient mining algorithm, which may overcome the shortcomings of the two approaches. This is the motivation of this study.

After some careful examination, we believe that such method is realizable. In this paper, we propose PPV algorithm, which is a new vertical mining algorithm based on some compact codes. PPV adopts a prefix tree structure called PPC-tree to store the database. Each node in a PPC-tree is assigned with a Pre-Post code via traversing the PPC-tree with Pre and Post order. Based on the PPC-tree with Pre-Post code, each frequent item can be represented by a Node-list, which is the list of PP-codes that consists of *pre-order* code, *post-order* code, and *count* of nodes registering the frequent item. Like other vertical algorithms, PPV gets Node-lists of the candidate patterns of length (k+1) by intersecting Node-lists of frequent patterns of length k and thus discovers the frequent patterns of length (k+1). The efficiency of PPV is achieved with three techniques. First, The Node-list is much more compact compared with previous proposed vertical structure (such as tidset or diffsets) since transactions with common prefix share the same nodes of the PPC-tree. Second, the support counting is transformed into intersection of Node-lists and the complexity of intersecting two Node-lists can be reduced to $O(m+n)$ by an efficient strategy, where m and n are the cardinalities of the two Node-lists respectively. Third, the ancestor-descendant relationship of two nodes, which is the basic step of intersecting Node-lists, can be very efficiently verified by Pre-Post codes of nodes.

These above techniques together form the core of our algorithm - PPV. A performance study has been conducted to compare the performance of PPV with FP-growth, Eclat, and dEclat, where Eclat and dEclat are the most efficient among all vertical algorithms. The experimental results show that PPV is efficient. PPV outperforms FP-growth, Eclat, and dEclat.

The remainder of the paper is organized as follows. A detailed problem description is given in Section 2. Node-list, its definition and construction method, and some important properties are described in Section 3. The PPV algorithm proposed for generating frequent patterns is developed in Section 4. Experimental results are presented in Section 5. Section 6 summarizes our study and points out some future research issues.

2. Problem Definition

The following is a formal description of the problem of mining frequent patterns. Let $I = \{i_1, i_2, \ldots, i_m\}$ be the universal item set. Let $DB = \{T_1, T_2, \ldots, T_n\}$ be a transaction database, where each T_k $(1 \le k \le n)$ is a transaction which is a set of items such that $T_k \subseteq I$. we also call A a pattern if A is a set of items. Let A be a pattern, a transaction T is said to contain A if and only if $A \subseteq T$. Let SP_A be the support of pattern A, which is the number of transactions in DB that contain A. Let ξ be the predefined minimum support threshold and $|DB|$ be the number of transactions in DB. A pattern A is frequent if SP_A is no less than $\xi \times |DB|$.

Given a transaction database DB and a minimum support threshold ξ, the problem of mining frequent patterns is to discover the complete set of patterns that have support no less than $\xi \times |DB|$..

3. Node-list: Definitions and Properties

In this section, we will describe the Node-list structure and some properties. Before the introduction of the Node-list, we first describe the PPC-tree, which is the basic of the Node-list.

3.1. *PPC-tree: Design and Construction*

We define a PPC-tree as follows.

Definition 1 PPC-tree is a tree structure:

(1) It consists of one root labeled as "null", 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*, *childreNode-list*, *pre-order*, and *post-order*. *item-name* registers which frequent item this node represents. *count* registers the number of transactions presented by the portion of the path reaching this node. *childreNode-list* registers all children of the node. *pre-order* is the preorder rank of the node. *post-order* is the postorder rank of the node.

According to Definition 1, PPC-tree seems like a FP-tree [2]. However, there are three important differences between them.

First, FP-tree has a *node-link* field in each node and a *header table* structure to maintain the connection of nodes whose *item-names* are equal in the tree, where PPC-tree does not have such structures. So PPC-tree is a simpler prefix free. Second, each node in the PPC-tree has *pre-order* and *post-order* fields while nodes in the FP-tree have none. The *pre-order* of a node is determined by a preorder traversal of the tree. In a preorder traversal, a node N is visited and assigned the preorder rank before all its children are traversed recursively from left to right. In other word, the *pre-order* records the time when node N is accessed during the preorder traversal. In the same way, the *post-order* of a node is determined by a postorder traversal of the tree. In a postorder traversal, a node N is visited and assigned its postorder rank after all its children have been traversed recursively from left to right.

Third, after a FP-tree is built, it will be used for frequent pattern mining during the total process of FP-growth algorithm, which is a recursive and complex process. However, PPC-tree is only used for generating the Pre-Post code of each node. Later, we will find that after collecting the Pre-Post code of each frequent item at first, the PPC-tree finishes its entire task and could be deleted.

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

Algorithm 1 (PPC-tree Construction)
Input: A transaction database DB and a minimum support threshold ξ.
Output: PPC-tree, F_1 (the set of frequent 1-patterns)
Method: Construct-PPC-tree(DB, ξ) {
//Generate frequent 1-patterns
(1) Scan DB once to find the set of frequent 1-patterns (frequent items) F_1 and their supports. Sort F_1 in support

descending order as I_f, which is the list of ordered frequent items.

//Construct the PPC-tree

(2) Create the root of a PPC-tree, *PPT*, and label it as "*null*". Scan *DB* again. For each transaction *T* in *DB*, arrange its frequent items into the order of I_f and then insert it into the PPC-tree. (This process is the same as that of FP-tree [2].)

//Generate the Pre-Post code of each node

(3) Scan PPC-tree by preorder traversal to generate the *pre-order*. Scan PPC-tree again by postorder traversal to generate the *post-order*. }

For better understanding of the concept and the construction algorithm of PPC-tree, let us examine the following example.

Example 1 Let the transaction database, DB, be the left two columns of Table 1 and $\xi = 40\%$.

Table 1. A transaction database

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

The PPC-tree storing the *DB* is shown in Figure 1. It should be noted that based on Algorithm 1 the PPC-tree is constructed via the last column of Table 1. Obviously, the second column and the last column are equivalent for mining frequent patterns under the given minimum support threshold. In the last 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.

For Pre-Post code generation, we traverse the PPC-tree twice by preorder and postorder. After that, we get the Figure 1. In this figure, the node with (3,7) means that its *pre-order* is 3, *post-order* is 7, and the *item-name* is *b*, *count* is 4.

3.2. *Node-list: Definition and Properties*

In this section, we will give the definition of the Node-list and introduce some important properties of the Node-list, which decide the efficiency and effectiveness of our new proposed algorithm for mining frequent patterns.

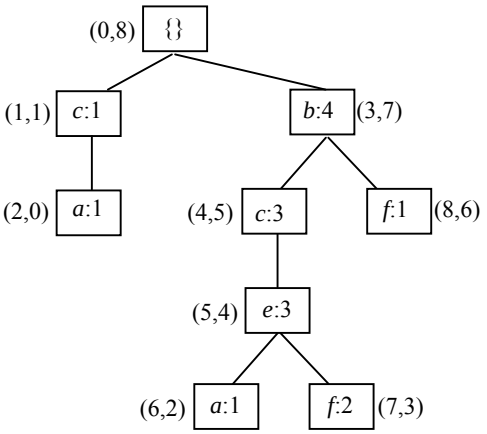

Fig. 1. The PPC-tree in Example 1

We first define the PP-code, which is the consisted element of the Node-list.

Definition 2 (PP-code) For each node *N* in the PPC-tree, we call < (*N.pre-order*, *N.post-order*): *N.count* > as the PP-code of *N*.

In fact, the target of constructing the PPC-tree is to generate the PP-codes of frequent items, since the PP-codes can effectively reflect the structure of the PPC-tree, which is described by the following property [12].

Property 1 Given any two different nodes N_1 and N_2 in a PPC-tree, N_1 is an ancestor of N_2 if and only if N_1.*pre-order* < N_2.*pre-order* and N_1.*post-order* > N_2.*post-order*.

It is determined by the construction of preorder rank and postorder rank. When N_1 is an ancestor of N_2, N_1 must be traversed earlier than N_2 during the preorder traversal and be traversed later than N_2 during the postorder traversal. On the other side, if N_1.pre-order < N_2.pre-order and N_1.post-order > N_2.post-order, N_1 is the node that is traversed earlier than N_2 during the preorder traversal and later during the postorder traversal. Such node must be an ancestor of N_2.

By using this property, it is easy to find the ancestor-descendant relationship of any two nodes just based on the preorder rank and postorder rank. Property 1 also shows that nodes and their PP-codes are 1-1 mapping. That is, a node uniquely determines a PP-code and a PP-code also uniquely determines a node. In fact, a node and its PP-code are equivalent. Therefore, we have the following definition.

Definition 3 (the ancestor-descendant relationship of PP-codes) Given two PP-codes X_1 and X_2, X_1 is the ancestor of X_2 if and only if the node represented by X_1 is the ancestor of the node represented by X_2.

Let X_1 be $<(x_1, y_1): z_1>$ and X_2 be $<(x_2, y_2): z_2>$, Definition 3 is equal to that X_1 is the ancestor of X_2 if and only if $x_1 < x_2$ and $y_1 > y_2$. We also call Y the descendant of X if X is the ancestor of Y.

Definition 4 (the Node-list of a frequent item) Given a PPC-tree, the Node-list of a frequent item is a sequence of all the PP-codes of nodes registering the item from the PPC-tree. The PP-codes are arranged by the accessed order during the preorder traversal.

Each PP-code in the Node-list is denoted by $<(x, y):z>$, where x is its *pre-order*, y is its *post-order* and z is its *count*. And the Node-list of a frequent item is denoted by $\{<(x_1, y_1): z_1>, <(x_2, y_2): z_2>, \ldots, <(x_l, y_l): z_l>\}$. For example, the Node-list of b includes one node. Its *pre-order* is 3, its *post-order* is 7, and its *count* is 4. Figure 2 shows the Node-lists of all frequent items in Example 1.

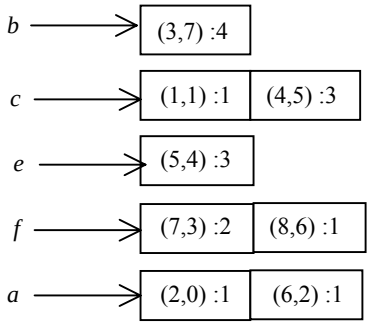

Fig.2. The Node-lists of frequent items in Example 1

Property 2 Given any two different nodes N_1 and N_2, which represent the same item (N_1.*item-name* = N_2.*item-name*), if N_1.*pre-order* < N_2.*pre-order*, then N_1.*post-order* < N_2.*post-order*.

When N_1.*pre-order* < N_2.*pre-order*, it means that N_1 is traveled earlier than N_2 during the preorder traversal. Since N_1 can not be the ancestor of N_2 because they both register the same item, so N_1 must be on the left branch of PPC-tree compared with N_2. During the postorder traversal, the left branch will also be traversed earlier than N_2, so N_1.*post-order* < N_2.*post-order*.

Given a Node-list of any item i, which is denoted by $\{<(x_1, y_1): z_1>, <(x_2, y_2): z_2>, \ldots, <(x_l, y_l): z_l>\}$, since we arrange the PP-code in the accessed order of preorder traversal, we have that $x_1 < x_2 <\ldots< x_l$. By property 2, we also have $y_1 < y_2 <\ldots< y_l$.

For example, in Figure 2 the Node-list of item c is $\{<(1,1):1>, <(4,5):3>\}$ and the Node-list of item f is $\{<(7,3):2>, <(8,6):1>\}$. They both show this property.

Property 3 Given a Node-list of any item i, which is denoted by $\{<(x_1, y_1): z_1>, <(x_2, y_2): z_2>, \ldots, <(x_m, y_m): z_m>\}$, the *support* of item i is $z_1 + z_2 + \ldots + z_m$.

It is determined by the definition of PP-code. Since each PP-code corresponds to a node in PPC-tree, whose *count* registers the number of transactions including item i, the sum of *counts* of nodes registering item i is i's *support*.

For better understanding of the concept of *the Node-list of pattern*, we first give the definition of *the Node-list of a 2-pattern*, which only contains two different items.

We denote L as the set of frequent items, which are sorted in support descending order. Based on L, we define $\succ$ relation of two items as follows.

Definition 5 ($\succ$ relation) For any two frequent items i_1 and i_2. $i_1 \succ i_2$ if and only if i_1 is ahead of i_2 in L.

For the sake of description, any pattern P in this paper is denoted by $i_1 i_2 \ldots i_k$, where $i_1 \succ i_2 \succ \ldots \succ i_k$.

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

For any $<(x_{1p}, y_{1p}): z_{1p}> \in$ the Node-list of i_1 ($1 \le p \le m$) and $<(x_{2q}, y_{2q}): z_{2q}> \in$ the Node-list of i_2 ($1 \le q \le n$), if $<(x_{1p}, y_{1p}): z_{1p}>$ is the ancestor of $<(x_{2q}, y_{2q}): z_{2q}>$, then $<(x_{2q}, y_{2q}): z_{2q}> \in$ the Node-list of $i_1 i_2$.

For example, in Figure 2, the Node-list of b is $\{<(3,7): 4>\}$ and the Node-list of c is $\{<(1,1): 1>, <(4,5): 3>\}$. According to Definition 6, only PP-code $<(4,5): 3>$ satisfies the combining rule. So the Node-list of bc is $\{<(4,5): 3>\}$ as shown in Figure 3.

Based on Definition 6, let us generalize it to the concept of the Node-list of a k-pattern ($k \ge 3$).

Definition 7 (the Node-list of a *k*-pattern) Let $P = i_1 i_2 \ldots i_{(k-2)} i_x i_y$ be a pattern ($k \ge 3$), and the Node-list of $P_1 = i_1 i_2 \ldots i_{(k-2)} i_x$ is $\{<(x_{P11}, y_{P11}): z_{P11}>, <(x_{P12}, y_{P12}): z_{P12}>, \ldots, <(x_{P1m}, y_{P1m}): z_{P1m}>\}$, the Node-list of $P_2 = i_1 i_2 \ldots i_{(k-2)} i_y$ is $\{<(x_{P21}, y_{P21}): z_{P21}>, <(x_{P22}, y_{P22}): z$

$_{P22}$>, …, <(x_{P2n}, y_{P2n}): z_{P2n} >} .The Node-list of P is a sequence of PP-codes according to *pre-order* ascending order and generated by intersecting the Node-lists of P_1 and P_2, which follows the rule below:

For any <(x_{P1r}, y_{P1r}): z_{P1r}> ∈ the Node-list of P_1 ($1 \le r \le m$) and <(x_{P2s}, y_{P2s}): z_{P2s}> ∈ the Node-list of P_2 ($1 \le s \le n$), if <(x_{P1r}, y_{P1r}): z_{P1r}> is the ancestor of <(x_{P2s}, y_{P2s}): z_{P2s}>, then <(x_{P2s}, y_{P2s}): z_{P2s}> ∈ the Node-list of P.

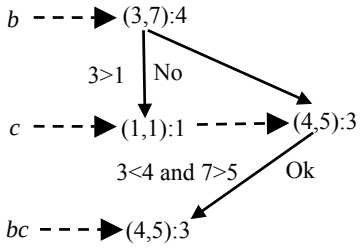

Fig.3. The Node-lists of *bc* in Example 1

Based on Definition 4, 6, and 7, we have property 4 as follows.

Property 4 Let <(x,y):z> be a PP-code in the Node-list of k-pattern i_1 i_2…i_k. The *item-name* of the node represented by <(x,y):z> is i_k.

For k = 1. According to Definition 4, we know that each PP-code in the Node-list of any frequent item i represents a node registering i. Therefore, Property 4 is right for k = 1.

For k = 2. According to Definition 6, we know that each PP-code in the Node-list of i_1i_2 is also in the Node-list of i_2. According to Definition 4, we know that each PP-code in the Node-list of i_2 represents a node registering i_2. Therefore, Property 6 is right for k = 2.

For k = 3. The Node-list of k-pattern $i_1i_2i_3$ is generated by the Node-list of i_1i_2 and i_1i_3. According to Definition 7, we know that each PP-code in the Node-list of $i_1i_2i_3$ is also in the Node-list of i_1i_3. However, according to the case k = 2, each PP-code of the Node-list of i_1i_3 represents a node registering i_3. Therefore, Property 6 is right for k = 3.

For k >3, the rationale is the same as k = 3. Therefore, we have Property 4.

Based on the above definitions and properties, we have the following important properties.

Property 5 Given a Node-list of any k-pattern $P = i_1$ i_2…i_k, which is denoted by {<(x_1, y_1): z_1>, <(x_2, y_2): z_2>, …, <(x_m, y_m): z_m >}, the *support* of pattern P is $z_1+z_2+\ldots+z_m$.

For k = 1. According to Property 3, the conclusion is right.

For k = 2. According to Definition 6, for any PP-code PC = <(x_j, y_j): z_j> in the Node-list of i_1i_2, <(x_j, y_j): z_j> must be a PP-code in the Node-list of i_2 and there must be a PP-code PC_1, which is the ancestor of PC, in the Node-list of i_1. By Definition 3, N_1, the node represented by PC_1, is the ancestor of N_2, the node represented by PC, in the original PPC-tree. According to Property 4, the *item-name* of N_2 is i_2 and the *item-name* of N_1 is i_1. That is, the *count* of N_2 registers the number of transactions containing both item i_1 and i_2. By recording all such nodes whose *item-name* is i_2 and one of its ancestors' *item-name* is i_1, we can get the *support* of pattern i_1i_2. Luckily, by Definition 6, the Node-list of 2-pattern keeps all these information in a simple and smart way. So we can directly get the *support* of pattern i_1i_2 by calculating the sum of the *counts* in each PP-code.

For k = 3. According to Definition 7, for any PP-code PC = <(x_j, y_j): z_j> in the Node-list of $i_1i_2i_3$, <(x_j, y_j): z_j> must be a PP-code in the Node-list of i_1i_3 and there must be a PP-code PC_1, which is the ancestor of PC, in the Node-list of i_1i_2. By Definition 3, N_1, the node represented by PC_1, is the ancestor of N_2, the node represented by PC, in the original PPC-tree. According to Property 4, the *item-name* of N_2 is i_3 and the *item-name* of N_1 is i_2. Because PC_1 is in the Node-list of i_1i_2, there must be a node N with i_1 as *item-name*, which is the ancestor N_1, according to the case k = 2. That is, N is the ancestor of N_1 and N_1 is the ancestor of N_2. Therefore, the *count* of N_2 registers the number of transactions containing item i_1, i_2, and i_3. So we can directly get the *support* of pattern $i_1i_2i_3$ by calculating the sum of the *counts* of each PP-code in the Node-list of $i_1i_2i_3$.

For k > 3. The rationale is the same as k = 3. Therefore, we have Property 5.

For example, the Node-list of *bc* is {<(4,5): 3>} as shown in Figure 3, so there is only one node in Figure 1, whose *item-name* is *c* and one of its ancestors' *item-name* is *b*. So the support of *bc* is 3.

Figure 4 shows the procedure that generates the Node-list of *bce* by the intersection of the Node-list of *bc* and the Node-list of *be*.

Property 6 Let $P = i_1$ i_2…i_k be a k-pattern and the Node-list of P is {<(x_1, y_1): z_1>, <(x_2, y_2): z_2>, …, <(x_m, y_m): z_m>}, we have $x_1 < x_2 < \ldots < x_m$ and $y_1 < y_2 < \ldots < y_m$.

In terms of Definition 4, 6 and 7, The Node-list of P is a sequence of PP-codes according to *pre-order* ascending order. Therefore, we have $x_1 < x_2 <\ldots< x_m$. In addition, each $<(x_j, y_j): z_j>$ corresponds to a node with *item-name* $= i_k$ according to Property 4. By property 2, we have $y_1 < y_2 <\ldots< y_m$.

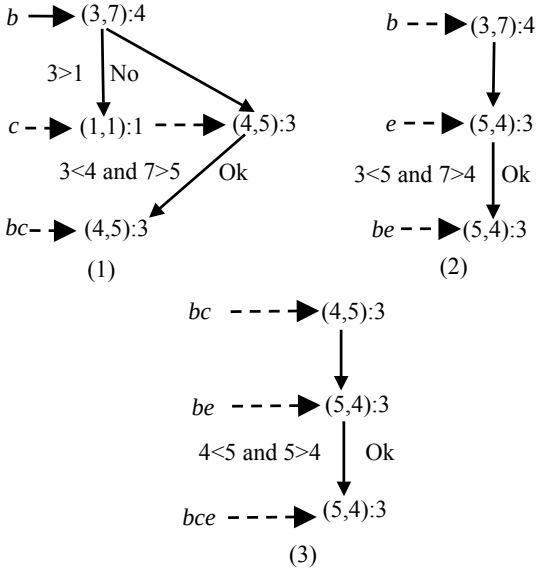

Fig. 4. The Node-lists of *bce* in Example 1

4. Mining Frequent Patterns using Node-list

In this section, we examine how to efficiently mine frequent patterns using Node-lists.

We adopt Apriori-like approach for mining frequent patterns. First, we generate the Node-lists of candidate (k+1)-patterns by intersecting the Node-lists of frequent k-patterns. Second, for any candidate (k+1)-pattern P_c, we obtain the supports of P_c by summing *count* values of all PP-codes in its Node-lists. According to the support of P_c, we can judge whether P_c is frequent or not. By repeating the above procedure, all frequent patterns will be found. The process of our method is the same as Eclat [7]. Eclat adopts tid-lists to mine the frequent patterns, while our method adopts Node-lists to mine the frequent patterns. It is obvious that the efficiency of intersecting two Node-lists is vital to the efficiency of mining frequent patterns. Before giving our intersecting method, let us first examine the following example.

Let $P_1 = i_1\ i_2\ldots i_{(k-2)}i_u$ and $P_2 = i_1\ i_2\ldots i_{(k-2)}i_v$ ($i_u \succ i_v$) be two (k-1)-patterns. The Node-list of P_1 is $\{<(x_{11}, y_{11}): z_{11}>, <(x_{12}, y_{12}): z_{12}>, \ldots, <(x_{1m}, y_{1m}): z_{1m}>\}$. The Node-list of P_2 is $\{<(x_{21}, y_{21}): z_{21}>, <(x_{22}, y_{22}): z_{22}>, \ldots, <(x_{2n}, y_{2n}): z_{2n}>\}$, For generating the Node-list of $P = i_1\ldots i_{(k-2)}\ i_u i_v$, a naïve method is to check each PP-code of the Node-list of P_1 with each PP-codes of the Node-list of P_2 to decide whether they satisfy the ancestor-descendant relationship. It is obvious that the time complexity of the naïve method is $O(mn)$. This time complexity is unsatisfying. After some careful analysis, we find a linear-time-complexity method, which is based on the following Lemma.

Lemma 1 Let $P_1 = i_1\ i_2\ldots i_{(k-2)}i_u$ and $P_2 = i_1\ i_2\ldots i_{(k-2)}i_v$ ($i_u \succ i_v$) be two (k-1)-patterns. The Node-list of P_1 is $\{<(x_{11}, y_{11}): z_{11}>, <(x_{12}, y_{12}): z_{12}>, \ldots, <(x_{1m}, y_{1m}): z_{1m}>\}$. The Node-list of P_2 is $\{<(x_{21}, y_{21}): z_{21}>, <(x_{22}, y_{22}): z_{22}>, \ldots, <(x_{2n}, y_{2n}): z_{2n}>\}$. If $\exists <(x_{1s}, y_{1s}): z_{1s}> \in P_1$ and $<(x_{2t}, y_{2t}): z_{2t}> \in P_2$, $<(x_{1s}, y_{1s}): z_{1s}>$ is the ancestor of $<(x_{2t}, y_{2t}): z_{2t}>$, then any $<(x_{1k}, y_{1k}): z_{1k}> \in P_1$ ($k \neq s$) cannot be the ancestor of $<(x_{2t}, y_{2t}): z_{2t}>$.

Proof. Let $<(x_{1s}, y_{1s}): z_{1s}>$ be the ancestor of $<(x_{2t}, y_{2t}): z_{2t}>$, N_1 be the node represented by $<(x_{1s}, y_{1s}): z_{1s}>$, N_2 be the node represented by $<(x_{2t}, y_{2t}): z_{2t}>$, and N be the node represented by $<(x_{1k}, y_{1k}): z_{1k}>$ ($k \neq s$). If N is the ancestor of N_2, then N_1 and N must have the ancestor-descendant relationship. According to Property 4, the *item-names* of N_1 and N are both i_u. But, by the construction of PPC-tree, nodes with the same *item-name* cannot have the ancestor-descendant relationship. So, we have the conclusion. □

Based on Property 6 and Lemma 1, the generation of the Node-list of $P = i_2\ldots i_{(k-2)}i_u i_v$ can be efficiently implemented by a linear method. The method first selects a PP-code from $\{<(x_{11}, y_{11}): z_{11}>, <(x_{12}, y_{12}): z_{12}>, \ldots, <(x_{1m}, y_{1m}): z_{1m}>\}$ according to the order from left to right. Then, it check the ancestor-descendant relationship of the PP-code and PP-codes in $\{<(x_{21}, y_{21}): z_{21}>, <(x_{22}, y_{22}): z_{22}>, \ldots, <(x_{2n}, y_{2n}): z_{2n}>\}$. In a word, our method makes use of the characteristic that PP-codes in a Node-list are ordinal. Let $<(x_{1i}, y_{1i}): z_{1i}>$ and $<(x_{2j}, y_{2j}): z_{2j}>$ be the current PP-codes to be proceeded, the detailing procedures are as follows:

(1) Check the ancestor-descendant relationship of $<(x_{1i}, y_{1i}): z_{1i}>$ and $<(x_{2j}, y_{2j}): z_{2j}>$.

(2) If $<(x_{1i}, y_{1i}): z_{1i}>$ is the ancestor of $<(x_{2j}, y_{2j}): z_{2j}>$ then insert $<(x_{2j}, y_{2j}): z_{2j}>$ into the Node-list of P. Go to (1) and go on checking the ancestor-

descendant relationship of $<(x_{1i}, y_{1i}): z_{1i}>$ and $<(x_{2(j+1)}, y_{2(j+1)}): z_{2(j+1)}>$.

(3) If $<(x_{1i}, y_{1i}): z_{1i}>$ is not the ancestor of $<(x_{2j}, y_{2j}): z_{2j}>$, there would be two cases: $x_{1i} > x_{2j}$ or $x_{1i} < x_{2j} \wedge y_{1i} < y_{2j}$. x_{1i} can not be equal to x_{2j} because they are the preorder ranks of different nodes. Similarly, y_{1i} can not be equal to y_{2j}.

(3.1) If $x_{1i} > x_{2j}$, go to (1) and go on checking the ancestor-descendant relationship of $<(x_{1i}, y_{1i}): z_{1i}>$ and $<(x_{2(j+1)}, y_{2(j+1)}): z_{2(j+1)}>$.

(3.2) If $x_{1i} < x_{2j} \wedge y_{1i} < y_{2j}$, go to (1) and go on checking the ancestor-descendant relationship of $<(x_{1(i+1)}, y_{1(i+1)}): z_{1(i+1)}>$ and $<(x_{2j}, y_{2j}): z_{2j}>$.

The rationality of step 3.2 can be explained as following. According to Property 6, we have $y_{2j} < y_{2t}$ for $j < t$. Because of $y_{1i} < y_{2j}$ in step 3.2, we have $y_{1i} < y_{2t}$. That is, $<(x_{1i}, y_{1i}): z_{1i}>$ can't be the ancestor of $<(x_{2t}, y_{2t}): z_{2t}>$. So, we need not check the ancestor-descendant relationship of $<(x_{1i}, y_{1i}): z_{1i}>$ and $<(x_{2t}, y_{2t}): z_{2t}>$, which means $<(x_{1i}, y_{1i}): z_{1i}>$ need not be processed any more. So, $<(x_{1(i+1)}, y_{1(i+1)}): z_{1(i+1)}>$, the next PP-code of $<(x_{1i}, y_{1i}): z_{1i}>$, should be selected as the next proceeded PP-code to check the ancestor-descendant relationship with the PP-codes from the Node-list of P_2. For any $<(x_{2k}, y_{2k}): z_{2k}>$ ($k < j$), there are two cases: (1) there exists $<(x_{1v}, y_{1v}): z_{1v}>$ ($1 \le v \le i$) that is the ancestor of $<(x_{2k}, y_{2k}): z_{2k}>$; (2) $<(x_{2k}, y_{2k}): z_{2k}>$ cannot be the descendant of $<(x_{1v}, y_{1v}): z_{1v}>$ for any v ($1 \le v \le i$). For case (1), $<(x_{2k}, y_{2k}): z_{2k}>$ cannot be the descendant of $<(x_{1(i+1)}, y_{1(i+1)}): z_{1(i+1)}>$ according to Lemma 1. For case (2), let us suppose $x_{1i} < x_{2k}$. We have $y_{1i} < y_{2k}$ because $<(x_{1i}, y_{1i}): z_{1i}>$ is not the ancestor of $<(x_{2k}, y_{2k}): z_{2k}>$. According to the above procedure, the ancestor-descendant relationship of $<(x_{1i}, y_{1i}): z_{1i}>$ and $<(x_{2u}, y_{2u}): z_{2u}>$ has not been checked for any u ($u > k$), which conflicts with the fact that we are checking the ancestor-descendant relationship of $<(x_{1i}, y_{1i}): z_{1i}>$ and $<(x_{2j}, y_{2j}): z_{2j}>$. So we have $x_{1i} > x_{2k}$. According to Property 6, we have $x_{1(i+1)} > x_{1i}$. So we have $x_{1(i+1)} > x_{2k}$, which means that $<(x_{2k}, y_{2k}): z_{2k}>$ cannot be the descendant of $<(x_{1(i+1)}, y_{1(i+1)}): z_{1(i+1)}>$. That is, We need not check the ancestor-descendant relationship of $<(x_{1(i+1)}, y_{1(i+1)}): z_{1(i+1)}>$ and $<(x_{2k}, y_{2k}): z_{2k}>$. Therefore, we should go to 1 to check the ancestor-descendant relationship of $<(x_{1(i+1)}, y_{1(i+1)}): z_{1(i+1)}>$ and $<(x_{2j}, y_{2j}): z_{2j}>$.

It is obvious that the method has an average running time of $O(m+n)$. Based on the idea of this method, we have the following code-intersection algorithm.

Algorithm 2 (code-intersection)

Input: NL_1 and NL_2, which are the Node-lists of two k-patterns.

Output: The Node-list of (k+1)-pattern.

Method: **code-intersection**(NL_1, NL_2)

```
(1)  int i = 0;   //Point to the start of NL1.
(2)  int j = 0;   //Point to the start of NL2.
(3)  while (i < NL1.size() && j < NL2.size()) {
(4)     if ( NL1[i].pre-order < NL2[j].pre-order) {
(5)        if (NL1[i]. pos-order > NL2 [j].pos-order){
(6)           Insert NL2 [j] into NL3;
(7)           j++;
(8)        }
(9)        else  i++;
(10)    }
(11)    else j++; }
(12) return NL3;
```

Based on the above analysis, we have the following algorithm for mining frequent patterns using Node-lists.

Algorithm 3 (PPV)

Input: the threshold ξ, the frequent 1-patterns and their Node-lists

Output: The complete set of frequent patterns.

Method: PPV (ξ, L_1, NL_1)

```
(1)  L1 = {frequent 1-patterns};
(2)  NL1 = {the Node-lists of L1};
(3)  For ( k = 2; Lk-1≠∅; k++) do begin {
(4)     For all p∈Lk-1 and q∈Lk-1, where p.i1 = q.i1,…,
        p.ik-2 = q.ik-2, p.ik-1 ≻ q.ik-1 do begin {
(5)     l = p.i1, p.i2,…,p.ik-1,q.ik-1; // Candidate k-pattern
(6)     If all k-1 subsets l of are in Lk-1 {
(7)        l.Node-list = code-intersection(p.Node-list,
           q.Node-list);
(8)        If (l.count ≥|DB|×ξ) { // Use Property 5 to get
           l.count from l.Node-list
(9)           Lk = Lk ∪{l};
(10)          NLk = NLk ∪ {l.Node-list}; }
(11)    }
(12)    end For; }
(13)    Delete NLk-1;
(14) end For; }
(15) Answer = ∪k Lk;
```

On step 5, we generate a new *k*-pattern from two (*k*-1)-patterns, and use Definition 6 and 7 to generate the Node-list of this candidate *k*-pattern on step 6. According to Property 5, the support of candidate *k*-patterns can be obtained from its Node-list. So we can get all the frequent patterns and their supports only based on the structure of Node-lists, which can be deleted after being used on step 11.

Many studies [7, 8] reveal that the process of mining frequent 2-patterns is high-cost because the number of candidate 2-patterns is usually huge, so it will be time-consuming to find frequent 2-pattersn by joining frequent 1-items. Therefore, we adopt the strategy in [8] to find all frequent 2-patterns without joining frequent 1-items. The processes of the strategy are follows. First, for each transaction, we get all its 2-patterns (subset). Then, we can get the support of each 2-pattern when we deal with all transaction. Finally, it's easy for us to find frequent 2-patterns when the support of each 2-pattern is known. In the study, we get all frequent 2-patterns in the process of constructing PPC-tree. For frequent k-patterns (k>2), we join frequent (k-1)-patterns in L_{k-1} to generate the candidate k-patterns, and then check whether these candidate k-patterns are frequent as showed in Algorithm 3 (PPV).

5. Experiments

In this section, we present a performance comparison of our algorithm with three classical frequent pattern mining algorithm, which are FP-growth, Eclat and dEclat.

5.1. *Experiment Setup*

We use four datasets, which are T25.I20.D100k, T40.I10.D100k, T40.I30.D1000k and Accidents. Accident [13] is a real dataset and includes traffic accidents records of the region of Flanders (Belgium) for the period 1991-2000. The other three datasets are synthetic and generated by IBM generator [14]. T25.I20.D100k and T40.I10.D100k have been used as benchmarks in many studies of frequent patterns mining, such as [2, 8]. For the sake of testing the performance our algorithm in large volume data, we build T40.I30.D1000k, which includes a million transactions. Table 2 shows the parameters of the synthetic and real datasets used in our evaluation, where *T* denote the average transaction length, *D* the number of transactions, *Size* the capability of datasets. As the support threshold goes down, there will be exponentially numerous frequent patterns in all datasets. They contain abundant mixtures of short and long frequent patterns.

All experiments are performed on an IBM xSeries 366 server with 2G memory, running Microsoft Windows 2000. All algorithms are coded in C++. Because different experiment platforms, such as software and hardware, may differ greatly on the runtime for the same algorithms, we do not directly compare our results with those in some published reports running on different experiment platforms. For the sake of impartiality, PPV, FP-growth, Eclat and dEclat are implemented on the same machine and compared in the same experiment environment. Notice that we implement FP-growth, Eclat and dEclat to the best of our knowledge based on the published literature.

Table 2. Database Parameter

Dataset	T	D	Size
T25.I20.D100k	25	100,000	12.5MB
T40.I10.D100k	40	100,000	16.5MB
T40.I30.D1000k	40	1,000,000	157.0MB
Accidents	34	340,183	33.8MB

5.2. *Performance Evaluation*

For evaluating our algorithm, we give a thorough set of experiments covering all the real and synthetic datasets mentioned in Table 2 for different values of minimum support. Figure 5 to Figure 8 show the advantage of PPV over the base methods, which are FP-growth, Eclat, and dEclat.

Let first compare how the algorithms perform on synthetic datasets. Figure 5 shows the run time of the algorithms on T25.I20.D100k as the minimum support decreased from 7% to 3%. We observe that Eclat is the worst and work only for high values of minimum support. The best among the four algorithms is PPV, which can be about twice as fast as FP-growth and dEcalt on average. For T25.I20.D100k, PPV and FP-growth shows very similar scalability. That is, they show similar trend when the minimum support changes. When the value of minimum support drops to 3%, the efficiency of dEcalt declines much faster than that of PPV and FP-growth.

Figure 6 shows the run time of the algorithms on T40.I10.D100k as the minimum support decreased from 4.5% to 2.5%. We observe that FP-growth is the worst. This result coincides with [8]. PPV and dEcalt are the best among the four algorithms. However, the scalability of dEcalt is worse than that of PPV. When the value of minimum support is below 3.5%, the efficiency of dEcalt is worse than that of PPV. It also obvious that the lower the value of minimum support is, the clearer the difference between dEcalt and PPV becomes. In addition, PPV and FP-growth still shows very similar scalability. When minimum support changes, the variety of efficiency of PPV and FP-growth is smoother than that of Ecalt and dEcalt.

Figure 7 shows the run time of the algorithms on T40.I30.D1000k as the minimum support decreased from 8.6% to 7.8%. We observe Eclat and dEcalt show good efficiency when the value of minimum support is above 8.4%. However, when the value of minimum support is below 8.4%, PPV is absolute the best among the four algorithms. Obviously, the lower the value of minimum support is, the more distinct the advantage of PPV is. When the value of minimum support is 8%, the rum time of Eclat is more than 2000 seconds. The rum time of FP-growth also exceeds 1000 seconds when the value of minimum support is 7.8%.

Now, let us compare how the algorithms perform on real datasets. Figure 8 shows the run time of the algorithms on Accidents as the minimum support decreased from 48% to 40%. We observe that efficiencies of PPV and FP-growth are almost the same. They are about an order of magnitude faster than Ecalt and dEcalt.

According to the above discussions, we have the conclusion that PPV is the best among the four algorithms on all synthetic and real dataset with various minimum supports.

The reason that PPV performs better than FP-growth lies in that PPV avoids the time consuming process of constructing a lot of conditional frequent pattern tree in FP-Growth by simply intersecting Node-lists. This advantage is more distinct when dataset are sparse, such as three synthetic datasets used in this study.

The reason that PPV performs better than Eclat and dEclat lies in that PPV adopts compact Node-lists to stand for patterns. Figure 9 and Figure 10 show the average length of Node-lists, tid-lists and diffsets of frequent patterns when frequent patterns are mined from T25.I20.D100k and Accidents. For T25.I20.D100k, the average length of Node-lists of frequent patterns is about an order of magnitude smaller than that of their tid-lists and diffsets. For Accidents, the average length of Node-lists of frequent patterns is about two orders of magnitude smaller than that of their tid-lists and diffsets. As for T40.I10.D100k, T40.I30.D1000k, the cases are the same as T25.I20.D100k. Of course, the operation of intersecting two Node-lists is much more complex than the operation of intersecting two tid-lists or two diffsets. Therefore, the advantage of PPV over Eclat and dEclaton on efficiency is not as much as the advantage Node-lists over tid-lists and diffsets on compression.

In fact, the advantage of PPV is that it effectively combines the idea of data compression of FP-growth and the idea of simple support computing of vertical mining method, which makes its success on different dataset and for various thresholds.

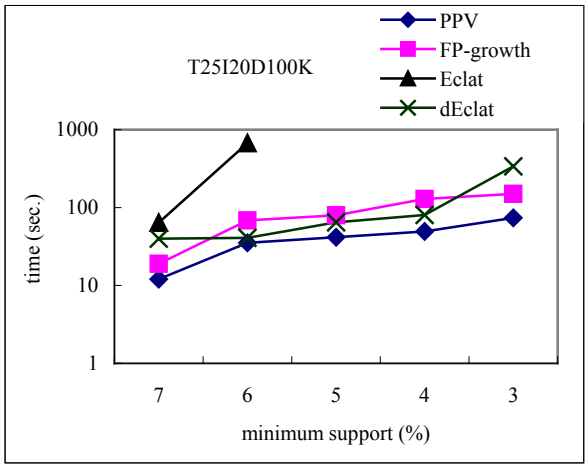

Fig. 5. Comparative Performance on T25I20D100K

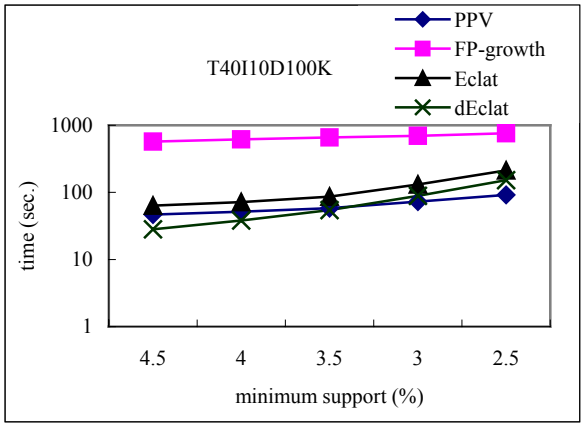

Fig. 6. Comparative Performance on T40I10D100K

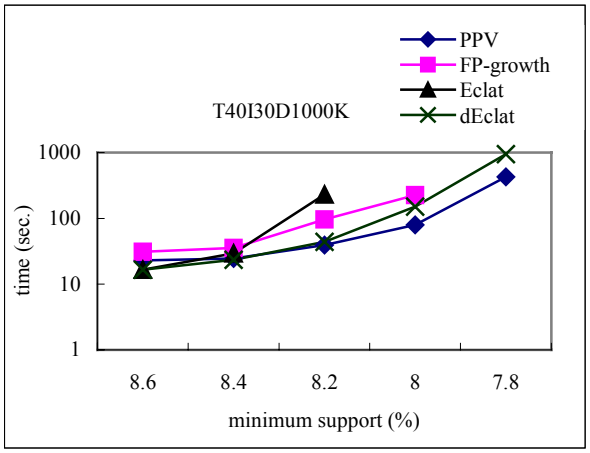

Fig. 7. Comparative Performance on T40I30D1000K

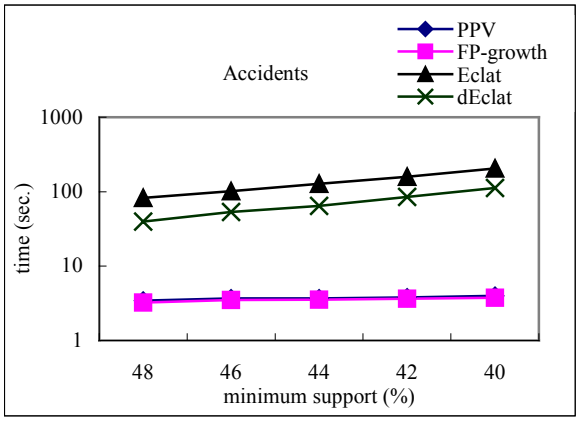

Fig. 8. Comparative Performance on Accidents

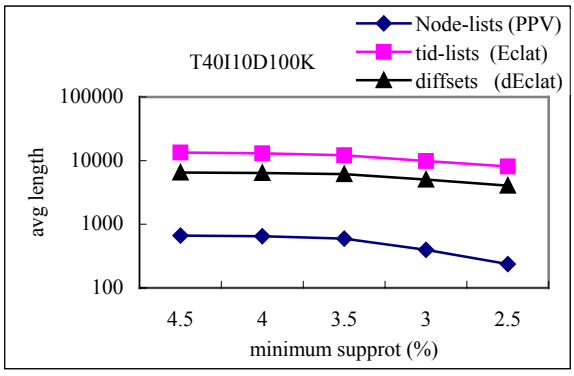

Fig. 9. Average Length of Node-lists, tid-lists, and diffsets Cardinality on T40I10D100K

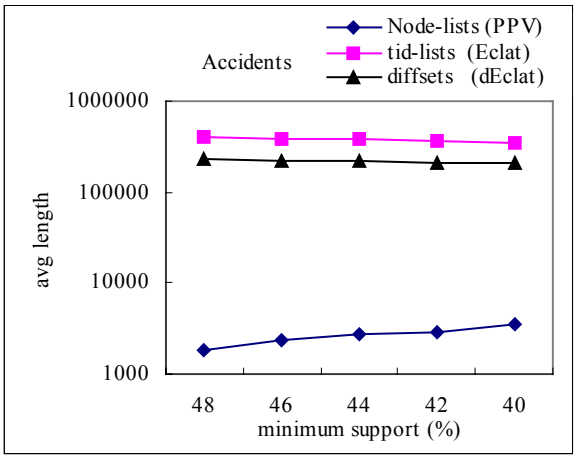

Fig. 10. Average Length of Node-lists, tid-lists, and diffsets Cardinality on Accidents

6. Conclusions

In this paper, we have proposed a compact tree structure, PPC-tree, for storing a transaction database. Based on PPC-tree, we developed an Apriori-like algorithm, PPV, for efficiently mining frequent patterns in large datasets. First, PPV obtains the Node-list of each frequent item. Then, PPV obtains Node-lists of the candidate patterns of length (k+1) by intersecting Node-lists of frequent patterns of length k and thus discovers the frequent patterns of length (k+1). The advantages of PPV are that it transforms the mining of frequent patterns into the intersecting of Node-lists, which makes mining process easier, and adopts an efficient method for intersecting two Node-lists, which has an average time complexity of $O(m+n)$. Our experimental results show that PPV is an efficient algorithm that outperforms FP-growth, Eclat, and dEclat.

Recently, there have been some interesting studies at mining closed frequent patterns [15, 16], top-k frequent patterns [17] and Sequential Patterns [18]. The extension of PPV for mining these special frequent patterns is an interesting topic for future research.

Acknowledgements

This work is supported by the National High Technology Research and Development Program of China (863 Program) under Grant No. 2009AA01Z136

and the National Natural Science Foundation of China under Grant No.90812001. The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

References

1. R. Agrawal, T. Imielinski, and A. Swami, Mining Association Rules Between Sets of Items in Large Databases, In proceedings of 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93), pp. 207-216.
2. J. Han, J. Pei, and Y. Yin, Mining frequent patterns without candidate generation, In proceedings of 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00), pp. 1-12.
3. R. Agrawal and R. Srikant, Fast algorithm for mining Association rules, In Proceedings of 1994 International Conference on Very Large Data Bases (VLDB'94), pp. 487-499.
4. A. Savasere, E. Omiecinski, and S. Navathe, An efficient algorithm for mining association rules in large databases, In Proceedings of 1995 International Conference on Very Large Data Bases (VLDB'95), pp. 432–443.
5. J. S. Park, M. S. Chen, and P. S. Yu, Using a hash-based method with transaction trimming for mining association rules, IEEE Transactions on Knowledge and Data Engineering, 9(5) (1997) 813–824.
6. P. Shenoy, J. R. Haritsa, S. Sundarshan, G. Bhalotia, M. Bawa and D. Shah, Turbo-Charging Vertical Mining of Large Databases, In Proceedings of 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00), pp. 22-33.
7. M. J. Zaki, Scalable algorithms for association mining, IEEE Transactions on Knowledge and Data Engineering, 12(3) (2000) 372-390.
8. M. J. Zaki and K. Gouda, Fast vertical mining using diffsets, In Proceedings of 2003 International Conference on Knowledge Discovery and Data Mining (SIGKDD'03), pp. 326-335.
9. G. Liu, H. Lu, Y. Xu, J. X. Yu, Ascending Frequency Ordered Prefix-tree: Efficient Mining of Frequent Patterns, In Proceedings of 2003 International Conference on Database Systems for Advanced Applications (DASFAA'03), pp. 65-72.
10. J. Han, J. Pei, Y. Yin, and R. Mao, Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach, Data Mining and Knowledge Discovery, 8 (2004) 53–87.
11. Y. K. Woon, W. K. Ng, and E. P. Lim, A Support-Ordered Trie for Fast Frequent Pattern Discovery, IEEE Transactions on Knowledge and Data Engineering, 16(7) (2004) 875–879.
12. T. Grust, Accelerating xpath location steps, In Proceedings of 2002 ACM SIGMOD international conference on Management of data (SIGMOD'02), pp: 109–120.
13. http://fimi.cs.helsinki.fi/data/
14. http://www.almaden. ibm.com/cs/quest/syndata.html.
15. J. Y. Wang, J. Han, and J. Pei, CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Patterns, In Proceedings of 2003 International Conference on Knowledge Discovery and Data Mining (SIGKDD'03), pp. 236-245.
16. M. J. Zaki and C. J. Hsiao, Efficient Algorithm for Mining Closed Patterns and Their Lattice Structure, IEEE Transactions on Knowledge and Data Engineering, 17(4) (2005) 462-478.
17. J. Y. Wang, J. Han, Y. Lu, and P. Tzvetkov, TFP: An Efficient Algorithm for Mining Top-k Frequent Closed Patterns, IEEE Transactions on Knowledge and Data Engineering, 17(5) (2005) 652-664.
18. S. Qiao, T. Li, and J. Peng, Parallel Sequential Pattern Mining of Massive Trajectory Data, International Journal of Computational Intelligence Systems, 3(3) (2010) 343 - 356.