Knowledge-Based Systems 122 (2017) 75–89

Contents lists available at ScienceDirect

Knowledge-Based Systems

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

Efficient algorithms for mining colossal patterns in high dimensional databases

Thanh-Long Nguyen[a,b], Bay Vo[c,d,*], Vaclav Snasel[e]

[a] Division of Data Science, Ton Duc Thang University, Ho Chi Minh City, Vietnam
[b] Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh City, Vietnam
[c] Faculty of Information Technology, Ho Chi Minh City University of Technology, Ho Chi Minh City, Vietnam
[d] College of Electronics and Information Engineering, Sejong University, Seoul, South Korea
[e] Department of Computer Science, Faculty of Electrical Engineering and Computer Science, VŠB - Technical University of Ostrava, 17. listopadu 15/2172, 708 33 Ostrava - Poruba, Czech Republic

ARTICLE INFO

Article history:
Received 2 May 2016
Revised 22 January 2017
Accepted 23 January 2017
Available online 25 January 2017

Keywords:
Bottom up
Colossal patterns
Data mining
High dimensional databases

ABSTRACT

Mining association rules plays an important role in decision support systems. To mine strong association rules, it is necessary to mine frequent patterns. There are many algorithms that have been developed to efficiently mine frequent patterns, such as Apriori, Eclat, FP-Growth, PrePost, and FIN. However, these are only efficient with a small number of items in the database. When a database has a large number of items (from thousands to hundreds of thousands) but the number of transactions is small, these algorithms cannot run when the minimum support threshold is also small (because the search space is huge). This thus causes the problem of mining colossal patterns in high dimensional databases. In 2012, Sohrabi and Barforoush proposed the BVBUC algorithm for mining colossal patterns based on a bottom-up scheme. However, this needs more time to check subsets and supersets, because it generates a lot of candidates and consumes more memory to store these. In this paper we propose new, efficient algorithms for mining colossal patterns. Firstly, the CP (Colossal Pattern)-tree is designed. Next, we develop two theorems to rapidly compute patterns of nodes and prune nodes without the loss of information in colossal patterns. Based on the CP-tree and these theorems, an algorithm (named CP-Miner) is proposed to solve the problem of mining colossal patterns. A sorting strategy for efficiently mining colossal patterns is thus developed. This strategy helps to reduce the number of significant candidates and the time needed to check subsets and supersets. The PCP-Miner algorithm, which uses this strategy, is then proposed, and we also conduct experiments to show the efficiency of these algorithms.

1. Introduction

Since the problem of mining association rules was first set out in 1993 [1], many algorithms for mining frequent (closed) patterns have been proposed, such as Apriori-based algorithms [2,8,14], FP (frequent pattern)-tree based algorithms [9,10,15], IT (itemset tidset)-tree based algorithms [19,20,23,24,25], bit vectors based algorithms [8,18,19], N-lists and Nodesets based algorithms [4,5,6,7,11,21]. Moreover, mining frequent patterns in uncertain databases [12,13] has also been examined. These algorithms are based on items (i.e., they are based on 1-items to generate 2-itemsets, based on 2-itemsets to generate 3-itemsets, and so on) for mining frequent (closed) patterns. The main purpose of these is to improve the mining time and/or memory usage for mining frequent (closed) itemsets. However, they are only efficient when the number of items that satisfy the minimum support threshold (minSup, in this paper, and we use this to mean the minSup count) in the database is small. When the number of items that satisfy the minSup is large, this leads to a huge search space and these algorithms are then inefficient, even though they cannot run [26] due to resource limitations. To solve this problem, Zhu et al. [26] proposed a method for mining colossal patterns called the Pattern-Fusion algorithm. To overcome the huge search space, Pattern-Fusion uses an approximation approach to mine K good frequent patterns. This means that Pattern-Fusion may not mine all colossal patterns that satisfy Definition 4 (below). In 2010 Dabbiru and Shashi [3] proposed the CMP (Colossal Pattern Miner) algorithm for mining such patterns. Next, Sohrabi and Barforoush [16] proposed a method for mining colossal patterns in high dimensional databases in 2012. The authors also proposed the BVBUC algorithm, which uses a bottom-up strategy based on transactions. The BVBUC

* Corresponding author.
E-mail addresses: nguyenthanhlong1@tdt.edu.vn (T.-L. Nguyen), bayvodinh@gmail.com, vd.bay@hutech.edu.vn (B. Vo), vaclav.snasel@vsb.cz (V. Snasel).

http://dx.doi.org/10.1016/j.knosys.2017.01.034
0950-7051/

joins 1-transactions together to generate 2-transactions, and so on, until the number of transactions reaches minSup (which means that the patterns in a set of transactions are frequent). Moreover, the authors also proposed a formula to prune branches that cannot expand to reach minSup to reduce the search space.

Although BVBUC is faster than CMP and Pattern-Fusion it has some limitations, as follows:

1. The patterns of a set of transactions are computed many times, and this makes BVBUC inefficient.
2. BVBUC uses the downward closure property to prune items that do not satisfy minSup, but it does not remove transactions (after removing infrequent items, some transactions do not contain any items).
3. BVBUC generates a lot of duplications, and thus more time is needed to check these.
4. BVBUC finds patterns based on a set of transactions (tidset) by computing the intersections among them. In fact, two tidsets only differ by one transaction if they have parent-child relationship. For example: tidsets {1, 2, 3} and {1, 2, 3, 4} only differ in transaction 4. In this case, if we have pattern X of tidset {1, 2, 3}, we can get pattern of tidset {1, 2, 3, 4} by computing the intersection between X and pattern of transaction 4.
5. In high dimensional databases, using bit vectors consumes more memory and time to join them.

Based on some of the limitations of BVBUC, in this work we propose a new method for mining colossal patterns. The main contributions of this paper are as follows:

1. We develop a method for computing the pattern of a set of transactions once.
2. Based on the downward closure property, we remove 1-items for which their supports do not satisfy minSup. After that, we remove transactions that do not contain any item to reduce the number of transactions that need to be traversed.
3. We use patterns at the parent level to compute patterns at the child levels to reduce the number of patterns that need to compute the intersections and reduce the duplications.
4. We develop theorems to prune non-colossal patterns early in the process.
5. We use dynamic bit vectors [18] instead of bit vectors to save memory and computational time.

The rest of this paper is organized as follows. Section 2 presents some works related to frequent (closed) pattern mining, mining colossal patterns and an overview of the BVBUC algorithm. Section 3 presents a theorem to compute the pattern from two sets of transactions and a theorem to quickly prune candidates, and proposes the CP-Miner algorithm for mining colossal patterns based on this. We also present a strategy for early pruning of the items and transactions based on minSup to reduce the search space. Section 4 presents a sorting strategy and pruning technique based on the parent-child relationships in the CP-tree. The PCP-Miner algorithm for efficiently mining colossal patterns is also developed. In Section 5 we compare the proposed algorithms with BVBUC with regard to runtime and number of nodes in the search trees, and discuss the results. Section 6 then gives our conclusions and some future research directions.

2. Related works

The problem of mining frequent patterns was first proposed by Agrawal et al. in 1993 [1], and this is the main problem of association rule mining. In 1994, Agrawal and Srikant developed the downward closure property to prune candidates that do not satisfy minSup [2]. The Apriori algorithm, a level-wise approach, was also proposed. Using the downward closure property, Apriori generates candidate (k+ 1)–itemsets from frequent k-itemsets and also uses this property to prune candidates. In 1997, Zaki et al. developed the Eclat algorithm for mining frequent itemsets using the IT-tree (Itemset Tidset-tree) [23]. Eclat applies an in-depth first search scheme and vertical data format to mine frequent itemsets. The advantages of this approach are that it only scans the database once and rapidly compute the supports of patterns based on the intersections of tidsets. Because tidsets consume more memory in dense databases, in 2003 Gouda and Zaki proposed using diffsets instead to reduce memory usage and time [24]. The IT-tree was also used in CHARM [25] to mine frequent closed patterns. Early in the process CHARM uses subset checking to omit patterns that cannot be closed, and checks whether a candidate is closed or not using hash table. Diffsets are also used in another study [25]. In 2000, Han et al. proposed the FP-tree structure and used it for mining frequent patterns [10]. In this, the FP-tree compresses the database in a prefix tree and then uses projections in this to mine frequent patterns. The authors also used an FP-tree to mine frequent closed patterns [22]. Grahne and Zhu used an FP-array to reduce the number of traverses and projections in an FP-tree, and applied an FP-array to mine frequent (closed) patterns [9]. Bit vector based algorithms were then also developed [8,17,18]. In 2007, Dong and Han proposed the BitTableFI algorithm for mining frequent patterns [8]. This is based on Apriori, but uses bit vectors to store tidsets of patterns and computes the supports of these by computing the intersections of bit vectors. The support of a pattern is the number of bits 1 in a bit vector. By using bit vectors, BitTableFI only scans the database once (Apriori scans the database k times, where k is the longest pattern). In 2008, Song et al. improved BitTableFI by using the subsume concept to quickly determine the support of subsumed patterns, and thus the supports do not need to be computed for these patterns [17]. However, BitTableFI and its improved algorithm (Index-BitTableFI) use fixed bit vectors, which means that the number of bits in each bit vector does not change and is the number of transactions in the database. When the number of transactions in the database is large, bit vectors consume more memory to store and time to compute their intersections. In 2012, Vo et al. proposed the dynamic bit vector (DBV) concept, and used it to mine frequent closed patterns [18]. A dynamic bit vector is also a bit vector, but it removes zero bits at the beginning and end. The authors also proposed an algorithm for computing the intersection between two DBVs, and the way to determine whether a pattern is not frequent in the process of computing the intersections of DBVs. In 2012, Deng and Lv proposed a method for mining frequent patterns using the N-list [4]. N-list has a structure like an FP-tree (adding Pre and Post for each node, and not storing a header table as FP-tree does). Unlike FP-Growth, an N-list based algorithm (PrePost) uses a vertical data format to mine frequent patterns. An improved PrePost algorithm using the subsume concept to reduce the search space has also been proposed [21], while N-lists have been used to mine frequent closed patterns [11]. Deng and Lv then proposed the Nodesets structure. Nodesets only use Pre (or Post) to compress the database, and the FIN algorithm uses this structure to mine frequent patterns. DiffNodesets, a different Nodesets strategy that can reduce both memory usage and computing time, was proposed in 2016 [7]. Yun et al. also proposed an LP-tree structure for mining frequent itemsets [15].

The above algorithms use item-extension to mine frequent (closed) patterns, but are only suitable when the number of frequent 1-items is small. When the number of frequent 1-items is large the search space is huge, making these algorithms inefficient. The concept of colossal patterns was thus developed to solve the problem of high dimensional databases (which cause the number of frequent 1-items to be large when minSup is small) [26]. The

concept of core-fusion was then developed, and the authors proposed the Pattern-Fusion algorithm for mining colossal patterns. In 2012, Sohrabi and Barforoush proposed the BVBUC algorithm [16], which uses bit vectors to present the pattern in each transaction, and applies vertical bottom-up traversing in transactions to mine colossal patterns.

3. CP-Miner algorithm

Definition 1 (Support of an itemset)**.** Given a transaction dataset D, the support of an itemset X, denoted by sup(X), is the number of transactions containing X.

Definition 2 (Frequent pattern)**.** Given a transaction dataset D, a pattern X is frequent if sup(X) $\geq$ minSup.

Definition 3 ((Core Pattern) [26])**.** Given a pattern Y, a pattern $X \subseteq Y$ is said to be a τ-core pattern of Y if sup(Y)/sup(X) $\geq \tau$, $0 < \tau \leq 1$ (τ is called the core ratio).

For a pattern Y, let C_Y be the set of all its core patterns, i.e., $C_Y = \{X \mid X \subseteq Y$ and X is a τ-core pattern of $Y\}$.

Definition 4 (Colossal Pattern)**.** FI is a set of all frequent patterns in a transaction database D. An itemset X is called a colossal pattern in FI if and only if there does not exist an itemset Y such that $X \subset Y$ and X is a τ-core pattern of Y.

The problem of mining colossal patterns is to find colossal patterns that satisfy the core ratio τ.

In this section, we present a new algorithm for mining colossal patterns. First, the downward closure property is used to remove items for which their supports do not satisfy minSup. After that, rows that do not contains any item will be removed to reduce the number of transactions (reducing the number of transactions will reduce the search space). Next, we encode the pattern in each transaction using DBV instead of a bit vector, as in BVBUC. To reduce checking duplications, patterns are sorted in descending order according to their length (the number of items in each pattern).

Before giving the definitions, theorems and data structure, we present an example to show the disadvantages of the BVBUC (with minSup $=3$ according to $\tau = 0.375$, in this case we consider both null items to present the core pattern as in [16]).

BVBUC uses a fixed 6 bits to present the bit vectors of transactions, as in Table 2.

The process for traversing the tree for mining colossal patterns according to BVBUC is shown in Fig. 1.

The results from Fig. 1 show that to obtain three colossal patterns (including patterns (1), (6) and (9)), we must traverse at least 31 nodes (some zero bit vectors are not shown in the tree). From the 10 patterns in Level 3, the following issues are raised:

(i) There are a lot of duplications. For example, nodes (2) and (3) have the same patterns; (4), and (5) have the same patterns; node (7) has the same pattern as nodes (8) and (10).

(ii) Superset checking: BVBUC traverses the tree according to the tids order, so it cannot check colossal patterns based on previous patterns. Therefore, some patterns in the results may not be colossal. To remove redundant patterns, when a new pattern Y is inserted into the result set then BVBUC must check it against each pattern X in the result set, and if Y is a superset of X then it is removed from the results. For example: Consider Fig. 1, pattern 000101 (node (2)) will be inserted into the results first. After that, 011101 (node 6) will be inserted. Because 011101 is a superset of 000101, 000101 is removed before 011101 is inserted.

Table 1
An example database.

Tid	Items
1	H, J
2	B, C, D, E, F
3	A, D, E, F, L
4	G, K
5	A, D, E, J
6	K, L
7	A, B, C, D, F, G
8	A, B, C, D, F, H

Table 2
Bit vectors of transactions in Table 1.

Tid	Bit vector
1	000000
2	011111
3	100111
4	000000
5	100110
6	000000
7	111101
8	111101

(iii) Subset checking: When a pattern in the tree is inserted into the result set, BVBUC must check whether it is a subset of any pattern in the results, if not, it is inserted. For example: Assume that nodes (1) and (2) are inserted into the results. Consider node (3): Because (3) is a subset of (2), (3) is not inserted into the results. A similar process occurs with nodes (4) and (5). Consider node (6): Because (6) is not a subset of any node in the result, it is inserted into the results.

Definition 5 (The pattern contained in a set of transactions)**.** For a set of transactions $\{t_{i1}, t_{i2}, \ldots, t_{ik}\}$, the pattern contained in $\{t_{i1}, t_{i2}, \ldots, t_{ik}\}$ is $t_{i1} \cap t_{i2} \cap \ldots \cap t_{ik}$.

For example: Consider Table 1, the pattern contained in transactions {2, 3, 5} is {B, C, D, E, F} ∩ {A, D, E, F, L} ∩ {A, D, E, J}={A, D, E} and that contained in transactions {2, 3, 7} is {B, C, D, E, F} ∩ {A, D, E, F, L} ∩ {A, B, C, D, F, G}={A, D}.

Theorem 1. *Given P_1 is the pattern contained in $T_1 = \{t_{i1}, t_{i2}, \ldots, t_{ik-1}, t_{ik}\}$ and P_2 is the pattern contained in $T_2 = \{t_{i1}, t_{i2}, \ldots, t_{ik-1}, t_{ik+1}\}$, then the pattern contained in $T = \{t_{i1}, t_{i2}, \ldots, t_{ik-1}, t_{ik}, t_{ik+1}\}$ is $P = P_1 \cap P_2$.*

Proof. According to Definition 5, we have $P_1 = t_{i1} \cap t_{i2} \cap \ldots \cap t_{ik-1} \cap t_{ik}$, $P_2 = t_{i1} \cap t_{i2} \cap \ldots \cap t_{ik-1} \cap t_{ik+1}$, and $P = t_{i1} \cap t_{i2} \cap \ldots \cap t_{ik} \cap t_{ik+1} \Rightarrow P_1 \cap P_2 = t_{i1} \cap t_{i2} \cap \ldots \cap t_{ik-1} \cap t_{ik} \cap t_{i1} \cap t_{i2} \cap \ldots \cap t_{ik-1} \cap t_{ik+1} = (t_{i1} \cap t_{i1}) \cap (t_{i2} \cap t_{i2}) \cap \ldots \cap (t_{ik-1} \cap t_{ik-1}) \cap t_{ik} \cap t_{ik+1} = t_{i1} \cap t_{i2} \cap \ldots \cap t_{ik} \cap t_{ik+1} = P$.

Based on Theorem 1, the pattern of a k-transaction can be generated from the patterns of (k−1)-transactions.

Theorem 2. *If a pattern X of a k-transaction T is a colossal pattern, then all patterns $X' \neq X$ created from T are not colossal patterns.*

Proof. Assume that X is a pattern of T and X' is a pattern of any $T \cup T'$ ($T' \subset D \setminus T$), this implies that $X' \subset X$. There are two cases:

(i) If sup(X)/sup(X') $< \tau$ then according to Definition 3, X' is not a τ-core pattern and thefore X' cannot be a colossal pattern, according to Definition 4.

(ii) If sup(X)/sup(X') $\geq \tau$. Because of $X' \subset X$, according to Definition 4, X' cannot be a colossal pattern.

All nodes that contain less than minSup transactions are infrequent, and node X that contains minSup transactions contains

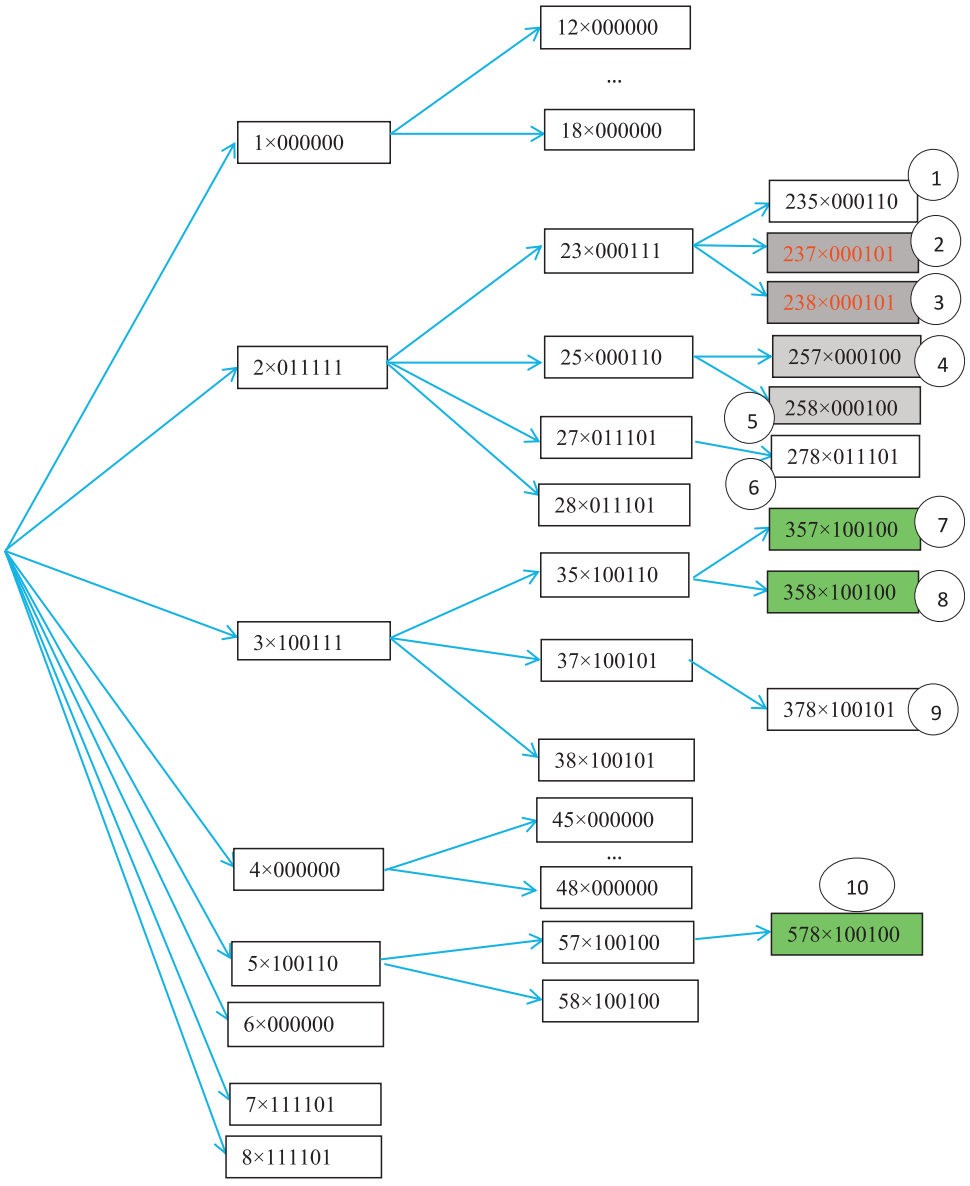

Fig. 1. Search tree using BVBUC.

the longest pattern if we compare it with all its child nodes (i.e., nodes contain more than minSup transactions). Therefore, based on Theorem 2, the proposed algorithm would not traverse nodes that have more than minSup transactions.

For example: Consider Fig. 1 with minSup $=2$ ($\tau = 2/8 = 0.25$), all nodes that contain one transaction are infrequent patterns (Level 1). Consider nodes (1), (2), and (3) in Level 3, these nodes are created from node $X = 000111$ in Level 2 (support $=2$):

- With node (1), $Y = 000110$ cannot be a colossal pattern because $Y \subset X$, according to Definition 4.
- With node (2), $Y = 000101$ cannot be colossal pattern because $Y \subset X$, according to Definition 4.
- With node (3), $Y = 000101$ cannot be colossal pattern because $Y \subset X$, according to Definition 4.

3.1. CP-tree

In this section, we present a tree structure, named the CP-tree, to store nodes to quickly mine colossal patterns.

Vertex: each vertex in this tree contains three elements

- Tids: a set of transaction IDs.
- Pattern: the pattern contained in the set of transaction IDs.
- Sup: the support of a pattern.

For example: Consider node (1) in Fig. 1, this node is created from transactions {2,3,4}, its pattern is 000110 and its support is 3.

Arc: Connect from a node X at level k to node Y at level (k+1) if the tids of X is the prefix of the tids of Y.

For example: Node 23×000111 connects to node 235×000110 because tids 23 is the prefix of tids 235.

In fact, tids are used to easily present the arcs of nodes. However, based on Theorem 1, the pattern of a new node is created from those of two previous nodes, so we do not need to store them in the tree and can thus save memory.

3.2. Algorithm

Fig. 2 presents the CP-Miner algorithm for mining colossal patterns that satisfy the core ratio τ in database D. First of all, CP-Miner computes the support of 1-items and removes 1-items for which their support does not satisfy minSup. After that, it deletes

```
Input: Transaction database D and core ratio τ.
Output: CP contains colossal patterns.
Method:
1.  minSup = ⌈τ × |D|⌉;
2.  D' = D after filtering items that do not satisfy minSup and removing zero transactions;
3.  [∅] = {{t1,1}, {t2,1}, ..., {tm,1}} from D';
4.  CP = ∅;
5.  CP-Miner([∅], CP, minSup);

Procedure CP-Miner([T], CP, minSup)
6.  If T.sup = minSup - 1 then
7.      For all n in [T] do
8.          If Checking-Colossal(n.pattern) then
9.              Insert n.pattern and n.sup to CP;
10. Else
11.     For all ni in [T] do
12.         If (ni.sup + |[T]| - i ≥ minSup) then
13.             [T1] = ∅;
14.             For all nj in [T] with j > i do
15.                 X.pattern = ni.pattern ∩ nj.pattern;
16.                 X.sup = ni.sup +1;
17.                 Add X into [T1];
18.             CP-Miner([T1], CP, minSup);

Checking-Colossal(pattern)
19. for each n in CP do
20.     if pattern ⊆ n then
21.         return false;
22. return true;
```

Fig. 2. CP-Miner algorithm for mining colossal patterns.

transactions that do not contain any item (after deleting items that do not satisfy minSup, line 2), the resulting database is *D'*. From *D'*, CP-Miner creates [∅] (the support of node ∅ is 0) containing 1-items along with their supports (line 3). *CP* is initialized by null (line 4), and line 4 computes minSup based on the core ratio τ. CP-Miner is then called in line 5 to mine all colossal patterns.

Consider the CP-Miner procedure: Line 6 checks if the support of the child nodes of node *T* is equal to minSup. If it is, the algorithm will add each pattern in these nodes into *CP* if it is colossal (lines 7–9). Lines 11 to 14 traverse each pair (n_i, n_j), where n_i, n_j are child nodes of *T* to create child nodes of n_i, and the pattern of a new node *X* is the intersection between n_i.pattern and n_j.pattern (line 15), the support of *X* is the support of $n_i + 1$ (and is the level of *X* in the CP-tree, line 16), and then add *X* to *CP* (line 17). Finally, CP-Miner is called on recursively to create the child nodes of [*T1*] (line 18).

Note that *T* is a node in the tree and [*T*] contains all nodes of the next level with *T*.tids is the prefix. Each node in [*T*] will create its child nodes as in the CP-Miner algorithm (Fig. 2). This is the reason why CP-Miner calls recursively to create children nodes. For example: If node $T = 2 \times 01{,}1111$ (Fig. 1), then $[T] = \{23 \times 000{,}111, 25 \times 000{,}110, 27 \times 01{,}1101, 28 \times 01{,}1101\}$.

3.3. Complexity analysis

For BVBUC, the complexity does not change and it depends on the number of transactions (n) in the database and minSup. If

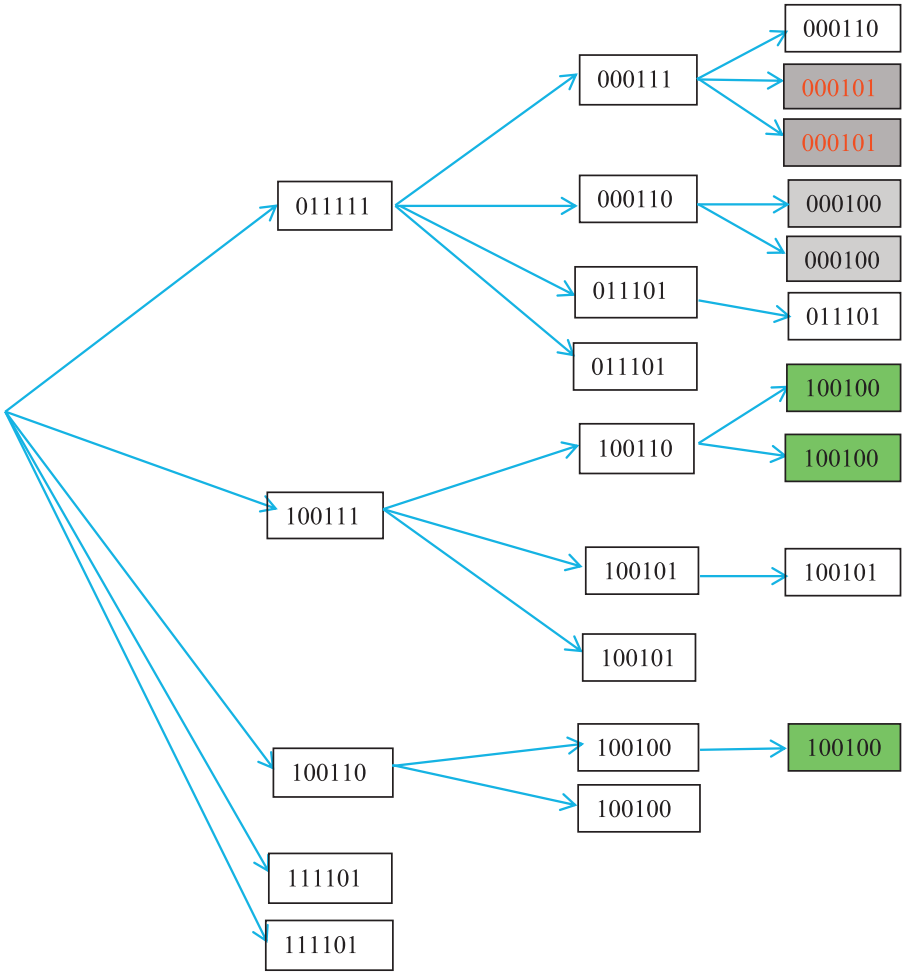

Fig. 3. Search tree using CP-Miner (the supports of the patterns are the levels of the nodes in the tree).

Table 3
Results after removing items for which their supports do not satisfy minSup.

Tid	Items
1	
2	B, C, D, E, F
3	A, D, E, F
4	
5	A, D, E
6	
7	A, B, C, D, F
8	A, B, C, D, F

pruning is not carried out with minSup, the complexity of BVBUC is $\sum_{i=1}^{minSup} C_n^i$.

Let m be the number of transactions in D', the complexity of CP-Miner is $\sum_{i=1}^{minSup} C_m^i$. m is often smaller than n, so the complexity of CP-Miner is smaller than BVBUC (In the worst case $m=n$, they have the same complexity). In fact, both BVBUC and CP-Miner use minSup to prune some nodes in the search tree.

3.4. Illustration

First, because sup(G)=sup(H)=sup(J)=sup(K)=sup(L) =2, these items are removed. Table 3 shows the results after removing items for which their supports do not satisfy minSup.

From Table 3, transactions that do not contain any item will be removed, and the results are shown in Table 4.

Table 4
Results after removing transactions and re-arranging transaction IDs.

Tid	Items
1	B, C, D, E, F
2	A, D, E, F
3	A, D, E
4	A, B, C, D, F
5	A, B, C, D, F

Table 5
Bit vectors of transactions in Table 4.

Tid	Bit vector
1	011111
2	100111
3	100110
4	111101
5	111101

After this, transactions are encoded as shown in Table 5 (in fact, we use DBV to reduce memory usage).

As we can see from Table 5, CP-Miner only processes five transactions instead of the eight that BVBUC does, and thus the search space is significantly reduced.

Fig. 3 illustrates the process of mining colossal patterns using CP-Miner. The number of nodes in the CP-tree is smaller than with BVBUC (at least 31 nodes in Fig. 1 compared to 24 nodes in Fig. 3);

Input: Transaction database D and core ratio τ.

Output: CP contains colossal patterns.

Method:

1. minSup = $\lceil \tau \times |D| \rceil$;
2. D' = D after pruning items that do not satisfy minSup and removing transactions;
3. $[\varnothing]$ = {{t_1,1}, {t_2,1}, ..., {t_m,1}} from D';
4. $CP = \varnothing$;
5. PCP-Miner($[\varnothing]$, CP, minSup);

Procedure PCP-Miner($[T]$, CP, minSup)

1. Sort-Pattern($[T]$);
2. If T.sup = minSup-1 then
3. For all n in $[T]$ do
4. If Checking-Colossal(n.pattern) then
5. Insert n.pattern and n.sup to CP;
6. Else
7. For all n_i in $[T]$ do
8. If (n_i.sup + |$[T]$| - i ≥ minSup) then
9. $[T1] = \varnothing$;
10. For all n_j in $[T]$ with j > i do
11. X.pattern = n_i.pattern ∩ n_j.pattern;
12. X.sup = n_i.sup +1;
13. If (|X.pattern| = | n_j.pattern|) then
14. Remove n_j from $[T]$;
15. Add X into $[T1]$;
16. PCP-Miner($[T1]$, CP, minSup);

Fig. 4. PCP-Miner algorithm for mining colossal patterns.

however, the number of patterns that need subset- and superset-checking is the same, because these two algorithms only check them at the minSup level (Level 3, both of them have 10 nodes, as shown in Figs. 1 and 4).

4. Pruning techniques

Definition 6 (Subsuming of a pattern)**.** Given two patterns P_1 and P_2, if $P_1 \subseteq P_2$ then P_1 is subsumed by P_2.

Theorem 3 (Check duplication based on subsuming)**.** *Given P_1 is the pattern of T_1 = {t_{i1}, t_{i2}, ..., t_{ik-1}, t_{ik}}, P_2 is the pattern of T_2 = {t_{i1}, t_{i2}, ..., t_{ik-1}, t_{ik+1}}, if $P_1 \subseteq P_2$ then the pattern that is created from T_1 and T_2 is a duplication.*

Proof. It is easy to see that the pattern that is created from T_1 and T_2 is $P = P_1 \cap P_2 = P_1$. This means that P is a duplication of P_1.

Based on Theorem 3, we can prune a node containing a subsumed pattern without losing information.

Theorem 4. *If pattern X of a node is subsumed by any colossal pattern then all patterns created from this node cannot be colossal patterns.*

Proof. According to Theorem 3, all patterns generated from X are the subsets of X. On the other hand, because X is a subset of a colossal pattern Y, this means that any X' generated from X is a subset of Y. This implies that X' cannot be a colossal pattern.

According to Theorem 4, we can prune nodes containing X from the search tree to save memory and time without losing information.

4.1. PCP-Miner algorithm

Based on Theorems 3 and 4, we propose an efficient algorithm, named PCP-Miner, for mining colossal patterns. Firstly, we sort transactions in D' in descending order according to their length. After that, we traverse the tree as with CP-Miner. Consider each node n_i with each node n_j following it, and because of sorting the n_i.pattern cannot be a subset of n_j.pattern, and thus there are two cases:

(i) If n_i.pattern ⊇ n_j.pattern then, according to Theorem 3, insert {n_j.pattern, n_i.sup+ 1} as a subnode of n_i and delete n_j.
(ii) Elseif insert {n_j.pattern ∩ n_j.pattern, n_i.sup+ 1} as a subnode of n_i.

Besides, when a pattern is inserted into CP, because we sort patterns based on their length, a pattern on the left hand side can-

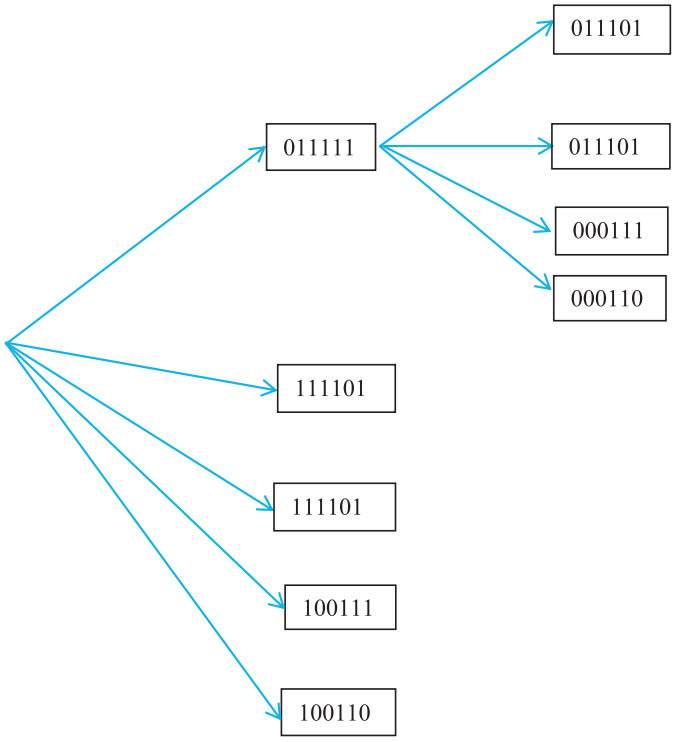

Fig. 5. CP-tree after extending to Level 2 of node {011111, 1}.

not be subset of all patterns on the right hand side. Therefore, only condition (i) needs checking and we may delete some nodes in the search tree to save memory.

Table 6
Bit vectors of transactions after sorting.

Tid	Bit vector
1	011111
2	111101
3	111101
4	100111
5	100110

Based on Theorems 3 and 4, an efficient algorithm named PCP-Miner is proposed in Fig. 4.

4.2. Complexity analysis

In worst case, i.e., all nodes in CP-tree cannot be pruned by Theorems 3 and 4, the complexity of PCP-Miner is the same that of CP-Miner. However, in case $t_1 \supseteq t_2 \supseteq ... \supseteq t_m$, the result is only t_1 and therefore all its following nodes are removed by Theorem 3. In this case, the complexity of PCP-Miner is only minSup.

4.3. Illustration

Consider the example database in Table 5, after transactions are sorted according to their length, we have the results as shown in Table 6.

The process for mining colossal patterns based on PCP-Miner is as follows.

First, Level 1 of CP-tree contains five bit vectors from Table 6, as shown in Fig. 5.

Consider the process of extending node n_i={011101, 2} (From Fig. 5) to find colossal patterns:

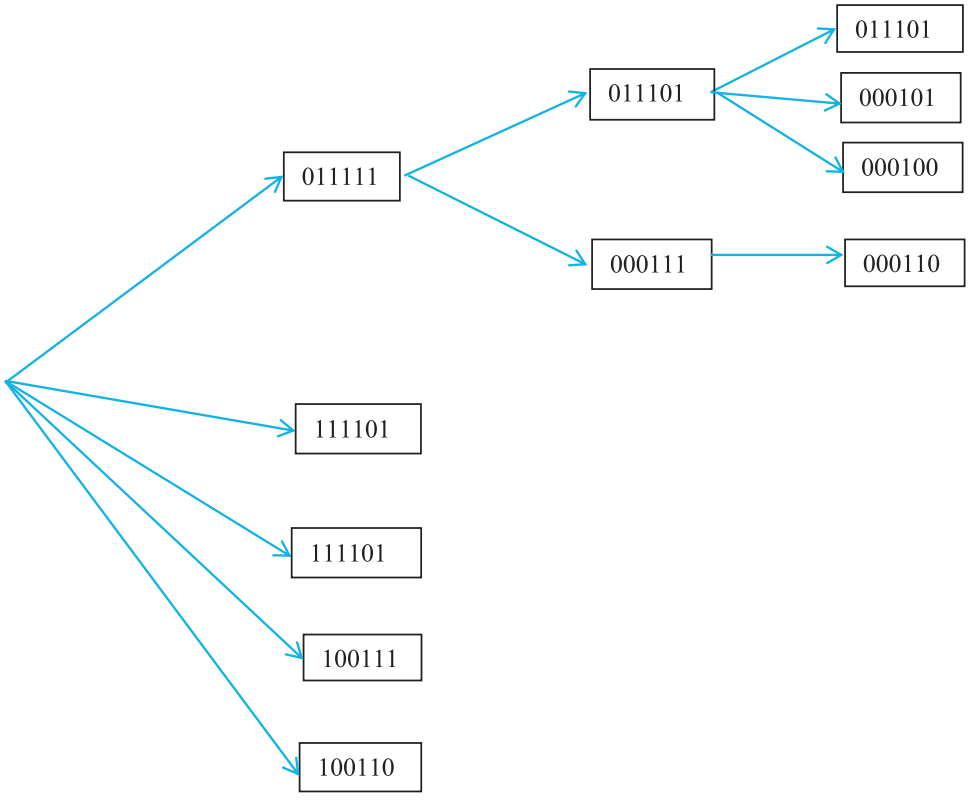

Fig. 6. CP-tree after extending to Level 3 of node {011111, 1}.

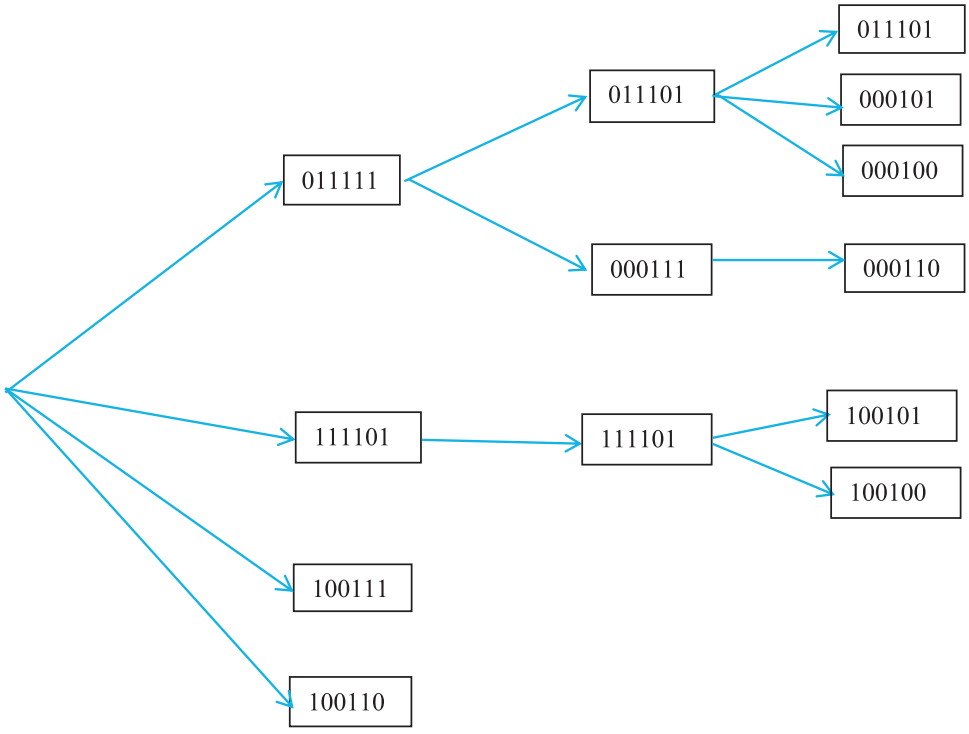

Fig. 7. Final CP-tree of the PCP-Miner algorithm.

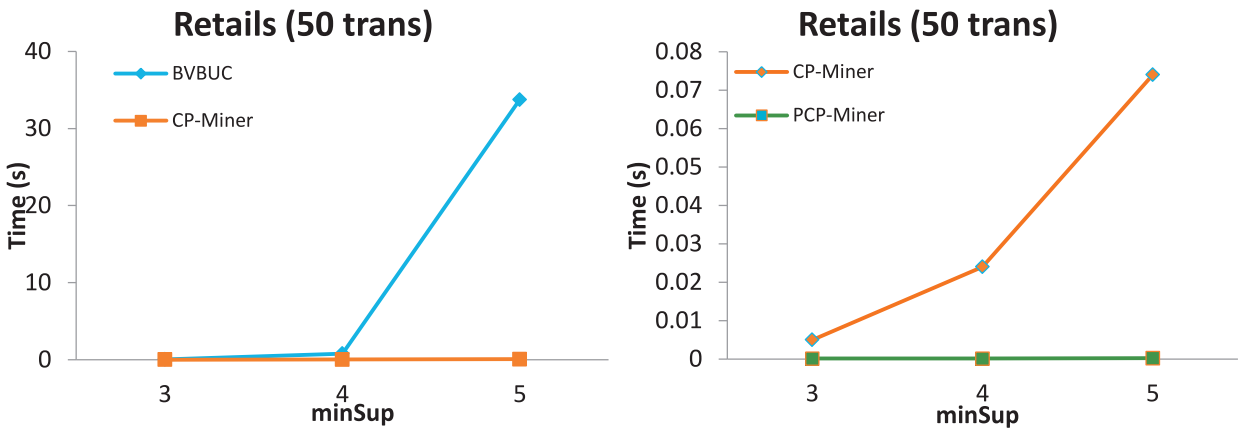

Fig. 8. Runtime in the Retails database when the number of transactions is 50 (the number of items is 275).

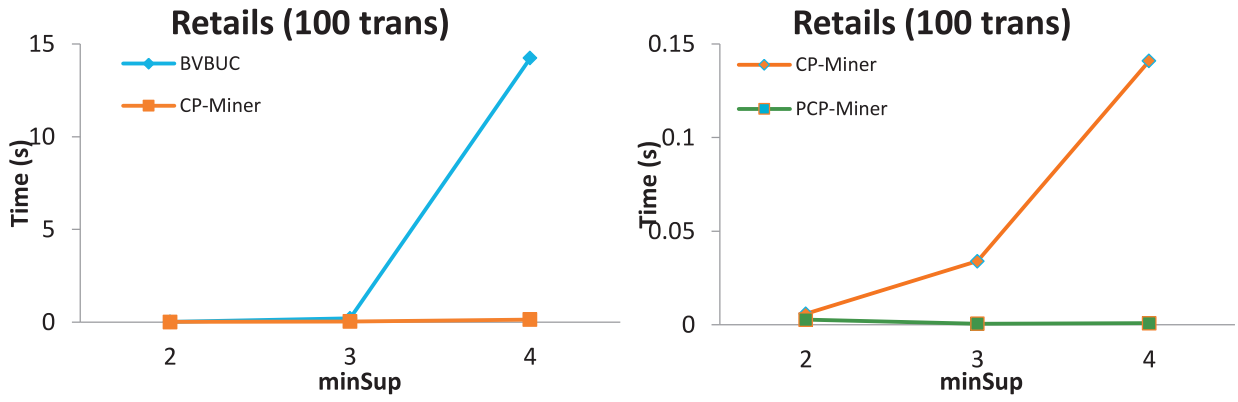

Fig. 9. Runtime in the Retails database when the number of transactions is 100 (the number of items is 532).

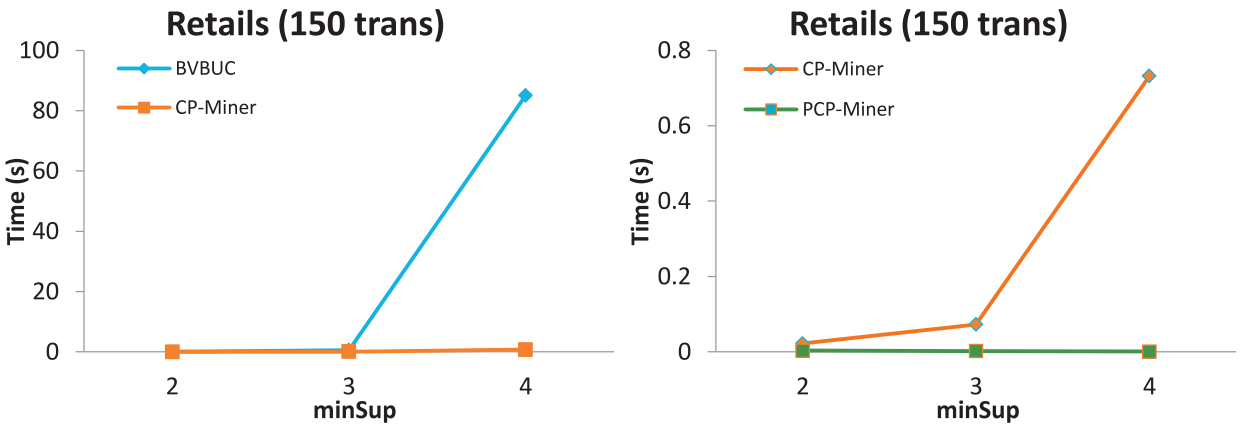

Fig. 10. Runtimes in the Retails database when the number of transactions is 150 (the number of items is 773).

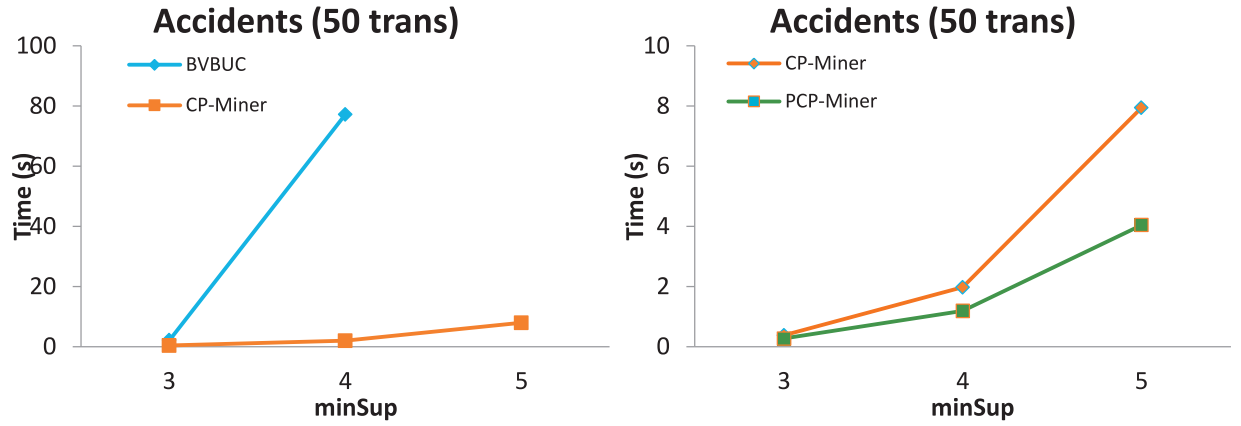

Fig. 11. Runtimes in the Accidents database when the number of transactions is 50 (the number of items is 151).

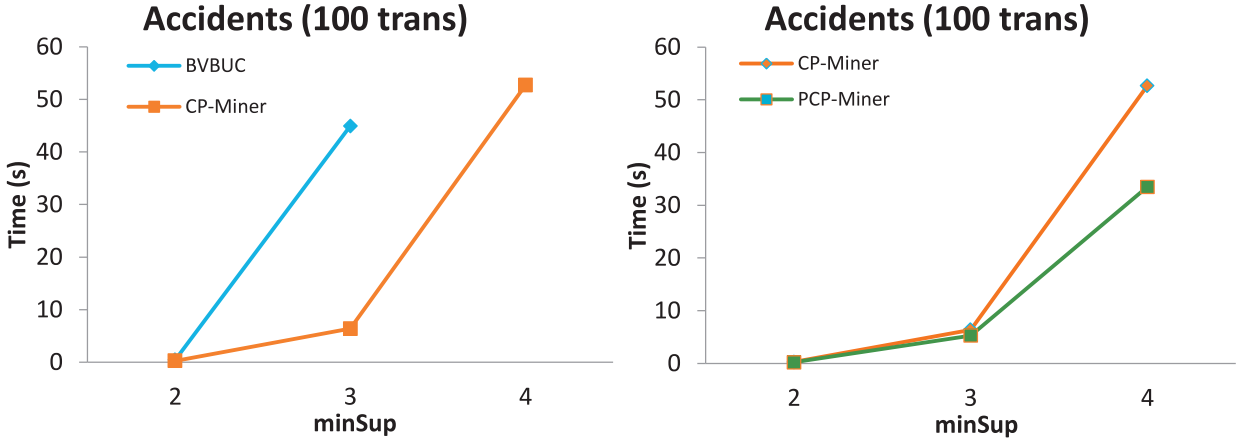

Fig. 12. Runtimes in the Accidents database when the number of transactions is 100 (the number of items is 168).

- Joins with node n_j={011101, 2}: Because 011101 ∩ 011101 = 011101, remove node n_j and add node X = {011101, 3} into Level 3.
- Joins with node n_j={000111, 2}: Because 011101 ∩ 000111 = 000101, add node X = {000101, 3} into Level 3.
- Joins with node n_j={000110, 2}: Because 011101 ∩ 000110 = 000100, add node X = {000100, 3} into Level 3.

Similar with node n_i={000111, 2}: Join with node n_j={000110, 2}, we get node {000110, 3}. Because 000110 is the same n_j.pattern, remove n_j and add node {000110, 3} into Level 3. The tree after considering these two nodes is shown in Fig. 6.

The final CP-tree is shown in Fig. 7. The number of nodes in Fig. 7 is 13, which is very small compared to that seen with CP-Miner and BVBUC (24 and 31, respectively). Besides this, the number of nodes at Level 3 (i.e., the number of nodes that need to be checked with patterns in *CP*) is only six (while BVBUC and CP-Miner is 10). Moreover, adding these patterns into *CP* is also more efficient than with CP-Miner and BVBUC. Consider the process of adding these patterns into *CP*:

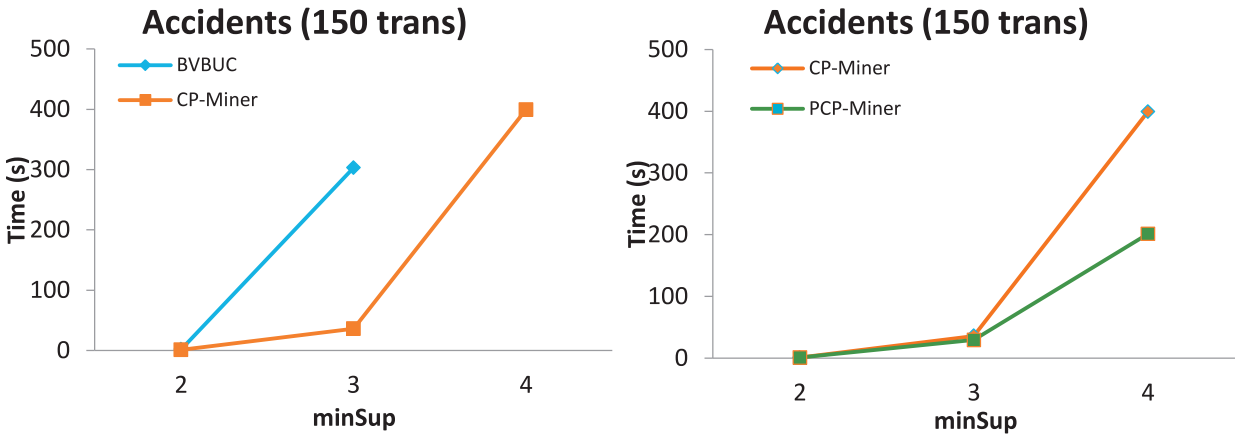

Fig. 13. Runtimes in the Accidents database when the number of transactions is 150 (the number of items is 177).

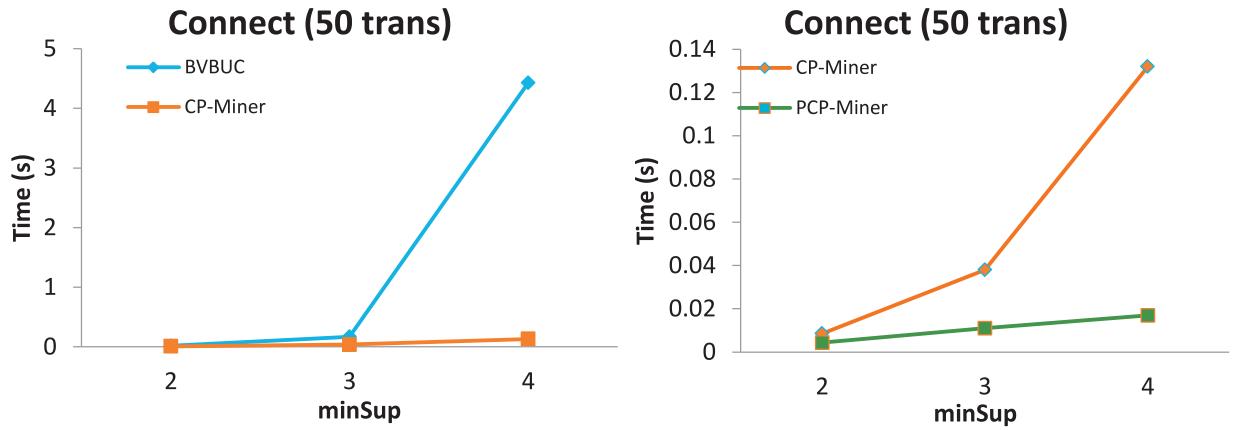

Fig. 14. Runtimes in the Connect database when the number of transactions is 50 (the number of items is 65).

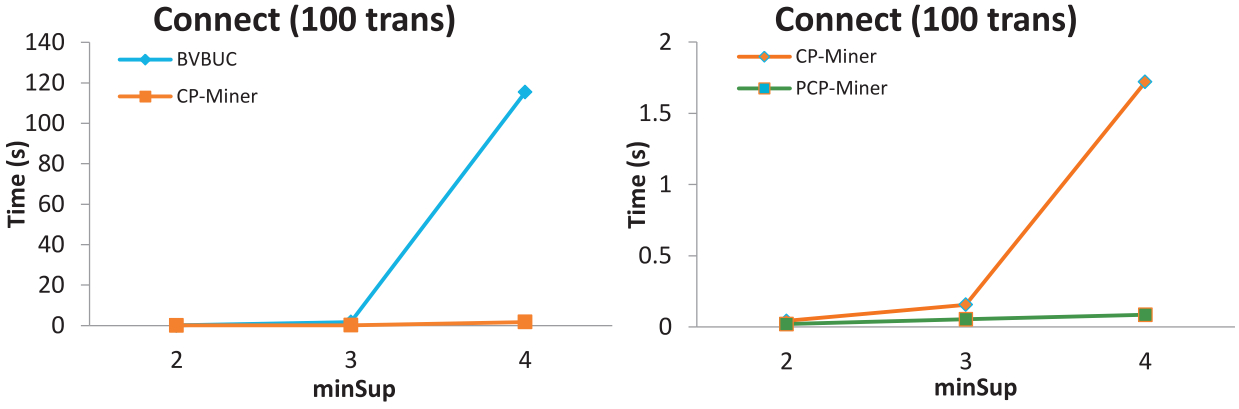

Fig. 15. Runtimes in the Connect database when the number of transactions is 100 (the number of items is 69).

- Consider pattern 011101: Add into *CP*.
- Consider pattern 000101: Because 000101 is a subset of 011101 (in *CP*), 000101 is not added into *CP*.
- Consider pattern 000100: Because 000100 is a subset of 011101 (in *CP*), 000100 is not added into *CP*.
- Consider pattern 000110: Add 000110 into *CP*.
- Consider pattern 100101: Add 100101 into *CP*.
- Consider pattern 100100: Because 100100 is a subset of 100101 (in *CP*), 100100 is not added into *CP*.

5. Experiments

5.1. Experimental environment and databases

All experiments presented in this section were performed on a PC with an Intel Core i5 3.2 GHz, and 4 GB of Ram, running on Windows 7 operating system, with the Visual C# 2010.

We made experiments on five databases, as shown in Table 7, which were downloaded from http://fimi.cs.helsinki.fi/data/.

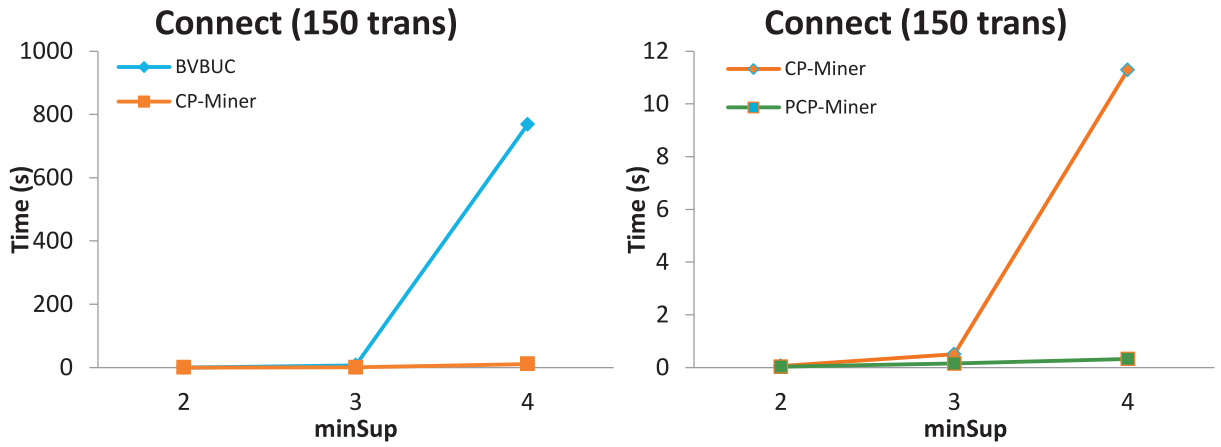

Fig. 16. Runtimes in the Connect database when the number of transactions is 150 (the number of items is 75).

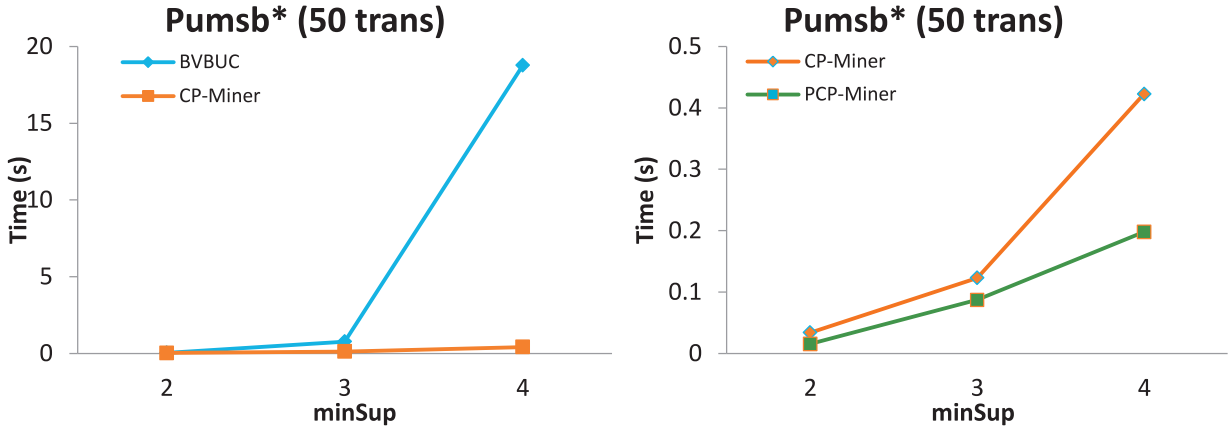

Fig. 17. Runtimes in the Pumsb* database when the number of transactions is 50 (the number of items is 344).

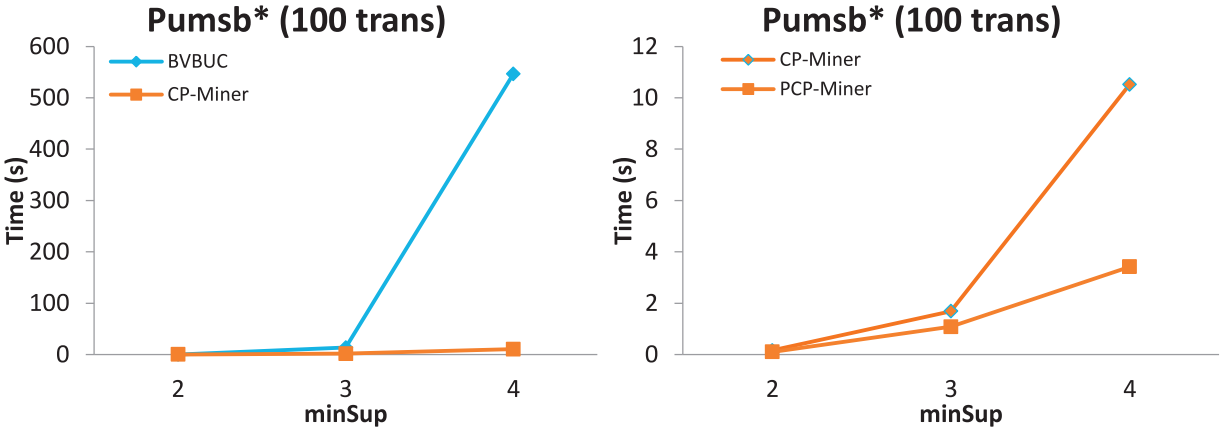

Fig. 18. Runtimes in the Pumsb* database when the number of transactions is 150 (the number of items is 456).

5.2. Comparison of the runtimes

We choose the first 50, 100, and 150 transactions in each database, as shown in Table 7, to compare the performances of the algorithms in high dimensional databases, with Figs. 8–22 showing the runtimes for BVBUC, CP-Miner and PCP-Miner.

(i) The results show that CP-Miner and PCP-Miner are always faster than BVBUC in all experiments. For dense databases such as Connect, Accidents, and Pumsb*, CP-Miner and PCP-Miner are very efficient compared to BVBUC. BVBUC runs very slowly (786 (s) with (#Rows, minSup)=(150, 4) in Connect (Fig. 16 left) or more than 1000 (s) in Accidents with (#Rows, minSup)=(50, 5) (Fig. 11 left), (100, 4) (Fig. 12 left) and (150, 4) (Fig. 13 left) in Pumsb* with (150, 4) (Fig. 19 left). For sparse databases, such as Retails and T10I4D100K, CP-Miner and PCP-Miner are also faster than BVBUC.

(ii) PCP-Miner is more efficient than CP-Miner in all experiments. However, we only run these experiments with a number of small transactions and items, so the gap between

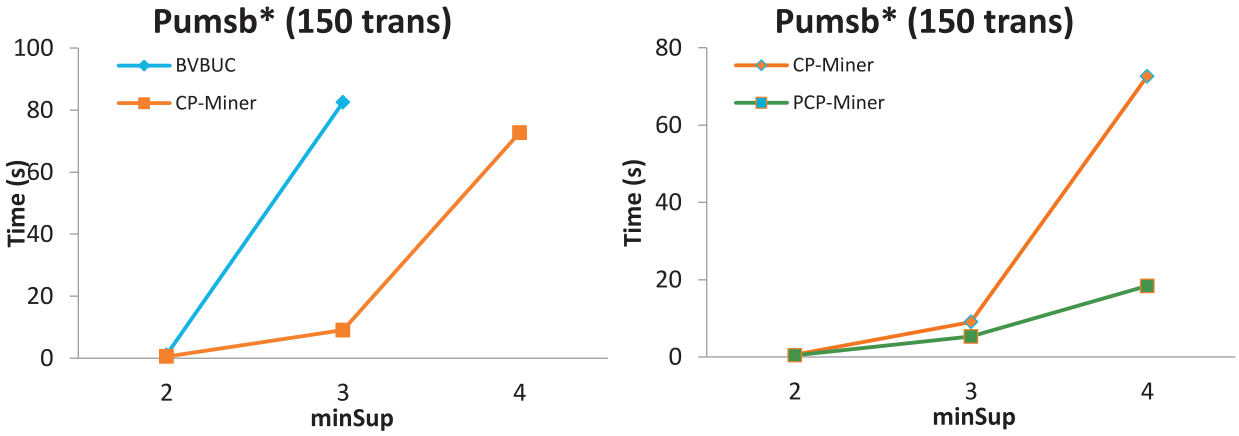

Fig. 19. Runtimes in the Pumsb* database when the number of transactions is 150 (the number of items is 574).

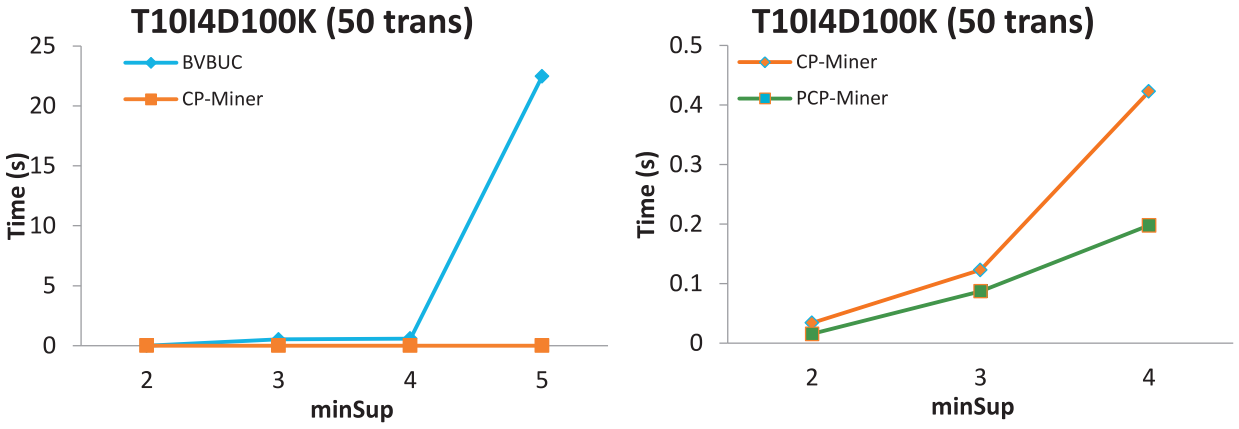

Fig. 20. Runtimes in the T10I4D100K database when the number of transactions is 50 (the number of items is 335).

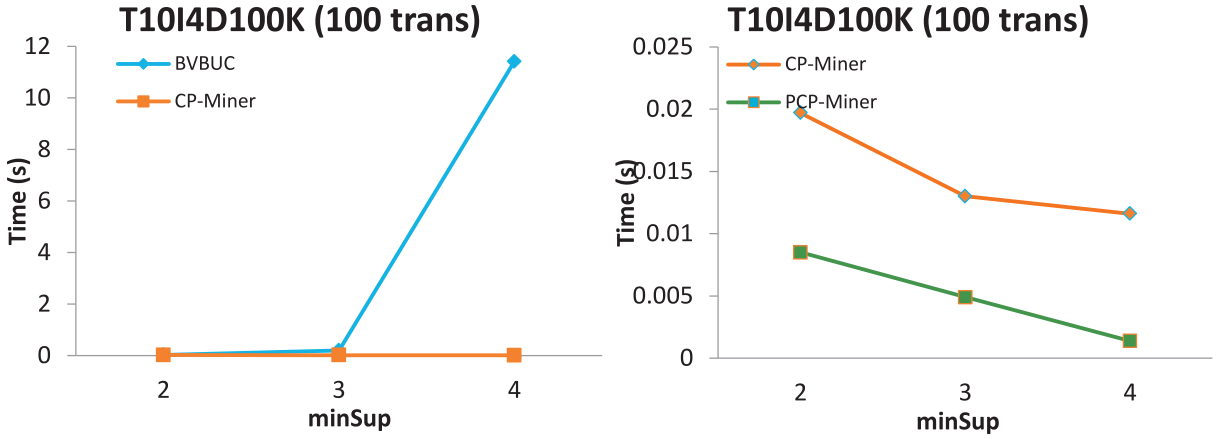

Fig. 21. Runtimes in the T10I4D100K database when the number of transactions is 100 (the number of items is 470).

Table 7
Characteristics of the experimental databases.

Database	# of items	# of transactions
Accidents	468	340,183
Connect	130	67,557
Retails	16,470	88,162
Pumsb*	7117	49,046
T10I4D100K	1000	100,000

CP-Miner and PCP-Miner is not very large (we will compare the scalability of these two algorithms in Section 5.4).

5.3. Comparison of the number of nodes in the trees

We also compare the number of nodes in the search space (using BVBUC, CP-Miner and PCP-Miner). Table 8 shows that the search space of BVBUC is very large (for example, the highest number of nodes in the search space for Retails, Connect and T10I4D100K is more than 470 million), while the search space of PCP-Miner is very small (for example, consider Connect with (#Rows, minSup)=(150, 4), the number of nodes of BVBUC is 470,169,066 while that of PCP-Miner is 119,076). Similarly, the

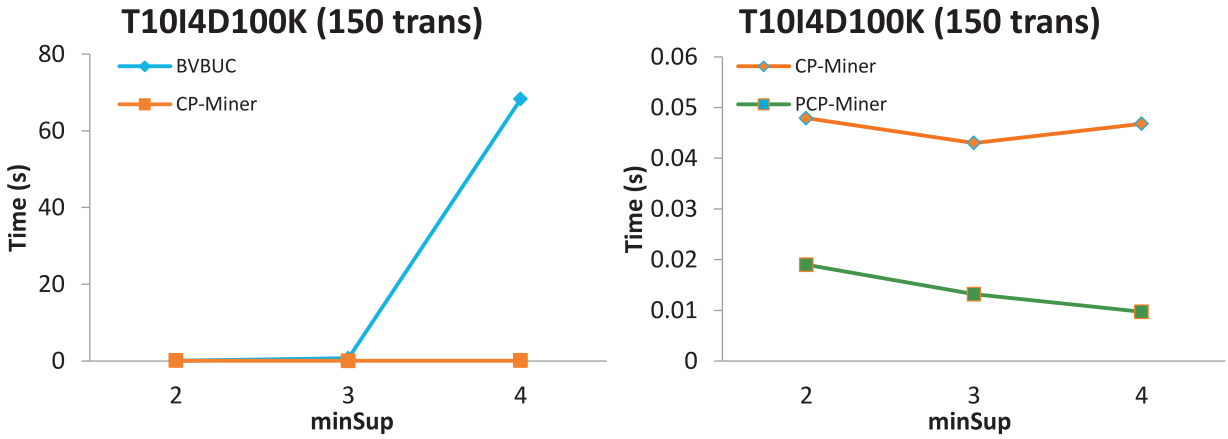

Fig. 22. Runtimes in the T10I4D100K database when the number of transactions is 150 (the number of items is 554).

Table 8
Comparison of the number of nodes.

Database	#Rows	minSup	BVBUC	CP-Miner	PCP-Miner
Retails	50	3	112,994	6735	185
		4	4988,066	48,884	232
		5	210,641,640	276,018	309
	100	2	9901	2492	1226
		3	950,994	44,673	926
		4	89,461,066	590,114	609
	150	2	22,351	5806	3025
		3	3263,994	162,501	2924
		4	470,169,066	3390,976	1996
Accidents	50	3	112,994	20,875	14,704
		4	4988,066	251,174	82,061
		5	#NA	2369,884	323,455
	100	2	9901	5050	5050
		3	950,994	166,750	97,209
		4	#NA	4087,974	717,911
	150	2	22,351	11,325	11,325
		3	3263,994	552,625	282,457
		4	#NA	20,822,899	2293,813
Connect	50	2	2451	1275	1275
		3	112,994	20,875	4745
		4	4988,066	251,174	10,987
	100	2	9901	5050	5050
		3	950,994	166,750	18,745
		4	89,461,066	4087,974	43,405
	150	2	22,351	11,325	11,325
		3	3263,994	562,625	46,645
		4	470,169,066	20,822,899	119,076
Pumsb*	50	2	2451	1275	1275
		3	112,994	20,703	10,330
		4	4988,066	236,988	37,168
		5	210,641,640	2013,532	92,279
	100	2	9901	5050	4974
		3	950,994	164,477	55,045
		4	89,461,066	3746,281	252,885
	150	2	22,351	11,325	11,216
		3	3263,994	554,700	156,112
		4	#NA	19,081,259	795,839
T10I4D100K	50	2	2451	301	298
		3	112,994	376	317
		4	4988,066	312	144
		5	210,641,640	223	71
	100	2	9901	1020	1995
		3	950,994	2017	1431
		4	89,461,066	2747	1350
	150	2	22,351	2164	2081
		3	3263,994	5681	3373
		4	470,169,066	10,527	3889

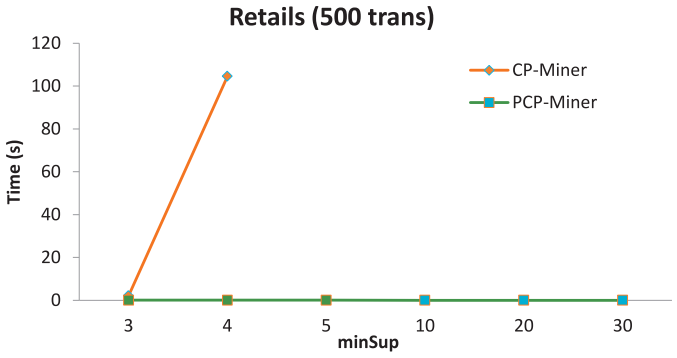

Fig. 23. Runtimes in the Retails database when the number of transactions is 500 (the number of items is 2058).

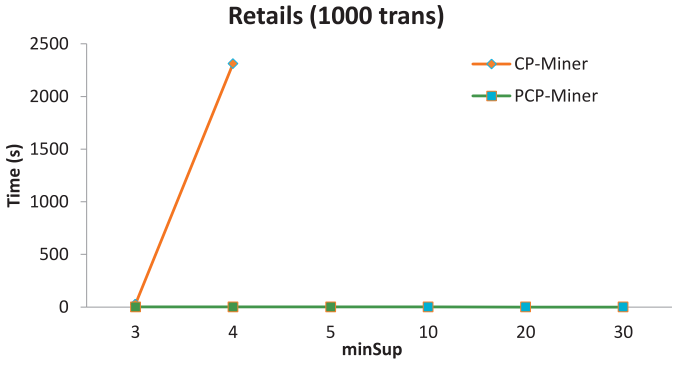

Fig. 24. Runtimes in the Retails database when the number of transactions is 1000 (the number of items is 3182).

search space of CP-Miner is smaller than that of BVBUC in all experiments.

5.4. Scalability of CP-Miner and PCP-Miner

We increase the number of transactions (m) to examine the scalability of two of the proposed algorithms (BVBUC cannot run with these cases), and the results are shown in Figs. 23–25. First, we run two algorithms with $m = 500$ (the first 500 transactions contain 2058 items). Fig. 23 shows that CP-Miner only runs to minSup $= 4$, when we increase minSup it runs very slowly (more than 3000 (s)) while the run time of the PCP-Miner algorithm does not change much (from 0.023 (s) to 0.109 (s)). Next, we increase m to 1000 (and the number of items is 3182), CP-Miner also cannot run

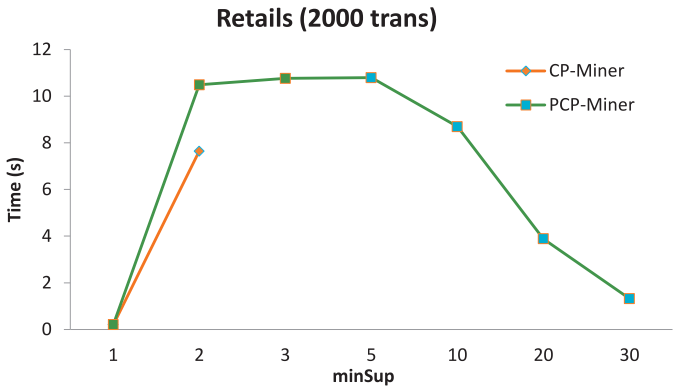

Fig. 25. Runtimes in the Retails database when the number of transactions is 2000 (the number of items is 4775).

with minSup greater than 4, while PCP-Miner can run to minSup = 30. Besides, when we increase minSup {3, 4, 5, 10, 20, 30}, the runtime decreases {1.068(s), 0.913(s), 0.776(s), 0.445(s), 0.113(s), 0.112(s)}. This is because PCP-Miner prunes a lot of nodes in the CP-tree. Finally, we increase m to 2000 (the number of items is 4775), and CP-Miner also cannot run with minSup greater than 2, while the runtime of PCP-Miner increases to {10.49(s), 10.766(s), 10.791(s)} at minSup = {2, 3, 4}. However, when we increase minSup to {5, 10, 20, 30}, the runtime decreases significantly, as shown in Fig. 25. The reason is that when minSup is large, the algorithm can prune more patterns than when the minSup is small (by Line 8, Fig. 4). Besides, when the level is high, PCP-Miner prunes more nodes (by Theorem 4) than when the level is low. Therefore, at the maximal level (minSup), the number of nodes is small, leading to the number of nodes that need to check colossal (by Line 4, Fig. 4) is small as well.

As we see in Fig. 25, CP-Miner cannot run with minSup > 2 when we increase the number of transactions to 2000. Therefore, we do not increase the number of transactions more than 2000.

6. Conclusions and future work

This paper has proposed a new method for mining colossal patterns. First of all, the CP-tree structure is developed. Then, two Theorems (1 and 2) for quickly mining colossal patterns and pruning nodes are designed. Based on these, we propose the CP-Miner algorithm for mining colossal patterns. CP-Miner mines colossal patterns using the CP-tree and pre-processing techniques to reduce the search space. Based on CP-Miner, a theorem for quickly checking duplicated patterns and one for rapidly checking non-colossal patterns are developed. These two theorems provide support for PCP-Miner, an improved version of the CP-Miner algorithm, which is able to efficiently mine colossal patterns.

BVBUC uses bit vectors to represent itemsets, when the number of items in the database is large, it is time consuming to join itemsets. the paper uses dynamic bit vectors to improve the performance and memory usage.

One of the weaknesses of bit vectors is that when the database is very sparse and the first position of bit 1 at the beginning and the last position of bit 1 at the end of each bit vector is very far, the number of bits in each bit vector is very large, although it contains a small number of bits 1. Although we have used DBV to solve this problem it only removes bits 0 at the beginning and end. In future work, we will study how to compress this case of databases in the tree. We will also apply the CP-tree to mining frequent closed patterns and frequent maximal patterns in high dimensional databases.

Acknowledgment

This work was funded by Vietnam's National Foundation for Science and Technology Development (NAFOSTED) under grant number 102.05-2015.10.

References

[1] R. Agrawal, T. Imielinski, A. Swami, Mining association rules between sets of items in large databases, SIGMOD (1993) 207–216.
[2] R. Agrawal, R. Srikant, Fast algorithms for mining association rules, VLDB (1994) 487–499.
[3] M. Dabbiru, M. Shashi, An efficient approach to colossal pattern mining, Int. J. Comput. Sci. Netw. Secur. 6 (2010) 304–312.
[4] Z.H. Deng, Z. Wang, J.J. Jiang, A new algorithm for fast mining frequent itemsets using N-lists, Sci. China Inf. Sci. 55 (9) (2012) 2008–2030.
[5] Z.H. Deng, S.L. Lv, Fast mining frequent itemsets using Nodesets, Expert Syst. Appl. 41 (10) (2014) 4505–4512.
[6] Z.H. Deng, S.L. Lv, PrePost+: an efficient N-lists-based algorithm for mining frequent itemsets via children-parent equivalence pruning, Expert Syst. Appl. 42 (13) (2015) 5424–5432.
[7] Z.H. Deng, DiffNodesets: an efficient structure for fast mining frequent itemsets, Appl. Soft Comput. 41 (2016) 214–223.
[8] J. Dong, M. Han, BitTableFI: an efficient mining frequent itemsets algorithm, Knowl. Based Syst. 20 (4) (2007) 329–335.
[9] G. Grahne, J. Zhu, Fast algorithms for frequent itemset mining using FP-trees, IEEE Trans. Knowl. Data Eng. 17 (10) (2005) 1347–1362.
[10] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, SIGMOD (2000) 1–12.
[11] T. Le, B. Vo, An N-list-based algorithm for mining frequent closed patterns, Expert Syst. Appl. 42 (19) (2015) 6648–6657.
[12] G. Lee, U. Yun, H. Ryang, An uncertainty-based approach: frequent itemset mining from uncertain data with different item importance, Knowl. Based Syst. 90 (2015) 239–256.
[13] J.C.W. Lin, W. Gan, P. Fournier-Viger, T.P. Hong, V.S. Tseng, Efficient algorithms for mining high-utility itemsets in uncertain databases, Knowl. Based Syst. 96 (2016) 171–187.
[14] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Efficient mining of association rules using closed itemset lattices, Inf. Syst. 24 (1) (1999) 25–46.
[15] G. Pyun, U. Yun, K.H. Ryu, Efficient frequent pattern mining based on linear prefix tree, Knowl. Based Syst. 55 (2014) 125–139.
[16] M.K. Sohrabi, A.A. Barforoush, Efficient colossal pattern mining in high dimensional datasets, Knowl. Based Syst. 33 (2012) 41–52.
[17] W. Song, B. Yang, Z. Xu, Index-BitTableFI: an improved algorithm for mining frequent itemsets, Knowl. Based Syst. 21 (6) (2008) 507–513.
[18] B. Vo, T.P. Hong, B. Le, DBV-Miner: a dynamic bit-vector approach for fast mining frequent closed itemsets, Expert Syst. Appl. 39 (8) (2012) 7196–7206.
[19] B. Vo, F. Coenen, B. Le, A new method for mining frequent weighted itemsets based on WIT-trees, Expert Syst. Appl. 40 (4) (2013) 1256–1264.
[20] B. Vo, N.Y. Tran, D.H. Ngo, Mining frequent weighted closed itemsets, Adv. Comput. Methods Knowl. Eng. (2013) 379–390.
[21] B. Vo, T. Le, F. Coenen, T.P. Hong, Mining frequent itemsets using the N-list and subsume concepts, Int. J. Mach. Learn. Cybern. 7 (2) (2016) 253–265.
[22] J. Wang, J. Han, J. Pei, CLOSET+: searching for the best strategies for mining frequent closed itemsets, KDD (2003) 236–245.
[23] M.J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, New algorithms for fast discovery of association rules, KDD (1997) 283–286.
[24] M.J. Zaki, K. Gouda, Fast vertical mining using diffsets, KDD (2003) 326–335.
[25] M.J. Zaki, C.J. Hsiao, Efficient algorithms for mining closed itemsets and their lattice structure, IEEE Trans. Knowl. Data Eng. 17 (4) (2005) 462–478.
[26] F. Zhu, X. Yan, J. Han, P. Yu, H. Cheng, Mining colossal frequent patterns by core pattern fusion, ICDE (2007) 706–715.