JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 28, 177-191 (2012)

A Search Space Reduced Algorithm for Mining Frequent Patterns

SHOW-JANE YEN[1], CHIU-KUANG WANG[1,2] AND LIANG-YUH OUYANG[2]
[1]*Department of Computer Science and Information Engineering*
Ming Chuan University
Taoyuan County, 333 Taiwan
[2]*Department of Management Sciences*
Tamkang University
New Taipei City, 251 Taiwan

Mining frequent patterns is to discover the groups of items appearing always together excess of a user specified threshold. Many approaches have been proposed for mining frequent patterns by applying the FP-tree structure to improve the efficiency of the FP-Growth algorithm which needs to recursively construct sub-trees. Although these approaches do not need to recursively construct many sub-trees, they also suffer the problem of a large search space, such that the performances for the previous approaches degrade when the database is massive or the threshold for mining frequent patterns is low. In order to reduce the search space and speed up the mining process, we propose an efficient algorithm for mining frequent patterns based on frequent pattern tree. Our algorithm generates a sub-tree for each frequent item and then generates candidates in batch from this sub-tree. For each candidate generation, our algorithm only generates a small set of candidates, which can significantly reduce the search space. The experimental results also show that our algorithm outperforms the previous approaches.

Keywords: data mining, frequent pattern, frequent itemset, FP-tree, transaction database

1. INTRODUCTION

Mining *frequent patterns* [1-3, 15-17] has become a common subject in the data mining research field. The very popular application domain using the frequent pattern discovery is the market basket analysis. We can analyze past transaction data to discover customer behaviors such that the quality of business decision can be improved. In most cases, enormous frequent patterns are generated if the specified threshold is low. Therefore, it becomes problematic to discover these patterns for reasons such as: high memory dependencies, huge search space, and massive I/O required.

The definitions about frequent patterns are described as follows. A *transaction database* consists of a set of transactions (*e.g.*, Table 1). A *transaction* is a set of items purchased by a customer at the same time. A transaction t contains an itemset X if every item in X is in t. The *support* for an itemset is defined as the ratio of the total number of transactions which contain this itemset to the total number of transactions in the database. The *support count* for an itemset is the total number of transactions which contain the itemset. A *frequent pattern* or a *frequent itemset* is an itemset whose support is no less than a certain user-specified minimum support threshold. An itemset of length k is called a k-itemset and a frequent itemset of length k a *frequent k-itemset*.

Received February 28, 2011; revised August 21, 2011; accepted August 28, 2011.
Communicated by I-Chen Wu.

Table 1. A transaction database TDB.

TID	Items	TID	Items	TID	Items	TID	Items
1	BGDCA	2	ACHED	3	ADEBM	4	CEFBN
5	BANOP	6	BCQRG	7	BCHIG	8	LEFKA
9	BFMNO	10	CFPGR	11	BDAHI	12	DEACL
13	ECAO	14	CFPQJ	15	DEFC	16	JEABD
17	KBEFC	18	CDLAB				

Various algorithms [2, 7, 11, 14] have been proposed to generate frequent itemsets from a large amount of transaction data. These algorithms generate candidate *k*-itemsets for frequent *k*-itemsets, scan each transaction in a database to count the supports for these candidate *k*-itemsets and find all the frequent *k*-itemsets in the *k*th iteration based on a minimum support threshold. However, because the size of the database can be very large, it is very costly to repeatedly scan the database to count supports for the candidate itemsets. The algorithm-DIC (Dynamic Itemset Counting) [4] can reduce the number of database scans. For each database scan, DIC counts the supports of the candidates whose lengths can be different. However, DIC has to take a lot of time to count the supports for a large number of candidate itemsets in each pass. Different from the above algorithms, the algorithm PAPG (Primitive Association Pattern Generation) [14] scans database once to record the related information and constructs an association graph. After constructing the graph, PAPG generates all the frequent itemsets by traversing the association graph. However, PAPG needs to take a lot of memory space to record the related information and spends a lot of time to perform intersections.

Although FP-Growth algorithm [5, 6] does not need to generate candidate itemsets, it has to take a lot of times to recursively construct many sub-trees which may not fit in main memory when the number of frequent itemsets is large. In order to avoid recursively constructing many sub-trees, COFI-tree algorithm [10] only builds a sub-tree for each frequent item, and generates candidate itemsets and counts their supports from the sub-tree. Since COFI-tree algorithm generates candidate itemsets with any length containing a specific item, the search space for counting the large number of candidates is very large. Algorithm PP-Mine [18] finds all the frequent itemsets through a coded prefix-path tree (PP-tree). PP-tree is similar to FP-tree and is a node-link-free tree structure. Moreover, each node in the PP-tree is arranged by the frequency order and is assigned a calculated code. PP-Mine does not need to construct any sub-tree, but it needs to recursively construct a large number of sub-header-tables and take a lot of time to search from the sub-header-tables when push-right and push-down operations occur. Therefore, PP-Mine faces a large search space if the set of frequent 1-itemsets is large.

In this paper, we investigate how to improve the efficiency for mining frequent itemsets. Since the database scans can be significantly reduced by constructing an FP-tree and it is fast to search for a small set of candidates, we propose an algorithm SSR (Search Space Reduced algorithm) for generating frequent patterns, which combines the advantages of FP-tree and candidate generation. Our SSR algorithm first constructs an FP-tree to store all the information in the transaction database. After building a compact sub-tree for each frequent item from the constructed FP-tree, SSR generates a small set of candidates in batch from the sub-tree, such that the search time and storage space can be reduced. Different from COFI algorithm [10] which searches for a large number of candi-

dates generated from a sub-tree, the sub-tree built by our algorithm is smaller than the sub-tree built by COFI and the search space for our algorithm is also much less than that of COFI. Therefore, our algorithm is more efficient than COFI in terms of execution times and memory storages, which are shown in section 4.

The remainder of this paper is organized as follows: Section 2 describes the related work about mining frequent patterns. Our algorithm for mining frequent patterns is proposed in section 3. Section 4 shows the experimental results. Finally, we conclude this paper in section 5.

2. RELATED WORK

The early approaches for mining frequent itemsets [2, 11-14] are based on Apriori-like approaches, which iteratively generate candidate $(k + 1)$-itemsets from frequent k-itemsets ($k \geq 1$) and check if these candidate itemsets are frequent. However in the cases of extremely large input sets or low minimum support threshold, the Apriori-like algorithms may suffer from two main problems of repeatedly scanning the database and searching for a large number of candidate itemsets.

In order to avoid generating a large number of candidate itemsets and scanning the transaction database repeatedly to count supports for the candidate itemsets, Han *et al*. [5, 6] proposed an efficient algorithm *FP-Growth*. This algorithm constructs a frequent pattern tree structure which is called *FP-tree*. FP-tree consists of a null root, a set of nodes and a *header table*. Each node, except the root node, in the FP-tree consists of three fields: *item-name*, *count*, and *item-link*. The *item-name* registers which item this node represents, *count* registers the number of transactions represented by the portion of the path reaching this node, and *item-link* links to the next node in the FP-tree carrying the same item or null if there is none. There is an item-link structure for each frequent item. Each entry in the header table consists of two fields: *item-name* and *head of item-link* which points to the first node in the FP-tree carrying the same item-name.

The construction of an FP-tree is described as follows: First, a null root node is created for the FP-tree T. For each transaction t in the database D, the frequent items in the transaction t are sorted by their supports in support descending order and the infrequent items in t are removed. Table 2 shows the frequent items and their support counts in Table 1, and Table 3 shows the sorted transactions after removing infrequent items from Table 1. Let n_0 be the root node of the FP-tree T. For the sorted transaction $\{t_1, t_2, \ldots, t_m, t_{m+1}, \ldots, t_q\}$, if the item contained in node n_i is t_i ($\forall i$, $1 \leq i \leq m$) and the item contained in node n_{m+1} is not t_{m+1}, then the count of node n_i for the path $n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m \rightarrow \ldots \rightarrow n_r$ ($r \geq m$) in the FP-tree T adds 1 and a new node with item t_{k+1} is created as a child of the node with item t_k ($\forall k$, $m \leq k \leq q - 1$), and the counts of these nodes are set to 1. After scanning all the transactions in the database D, the FP-tree with the associated item-links starting from the header table is shown in Fig. 1. FP-Growth algorithm requires only two full I/O database scans to build an FP-tree in main memory and then recursively mines frequent patterns from this structure by building *conditional FP-trees* [5, 6]. However, this massive creation of conditional FP-trees makes this algorithm not scalable to mine large datasets.

The Co-Occurrence Frequent Item Tree or COFI for short algorithm [10] is based on the core idea of the FP-Growth [5, 6]. One small tree (called COFI-tree) is built for each frequent item x by traversing all the paths with leaf node x from an FP-tree and the sup-

Table 2. Frequent items and their support counts with minimum support 25%.

Item	Count	Item	Count	Item	Count
C	12	B	11	A	10
E	9	D	8	F	7

Table 3. A sorted transaction database with minimum support 25%.

TID	Items	TID	Items	TID	Items	TID	Items
1	CBAD	2	CAED	3	BAED	4	CBEF
5	BA	6	CB	7	CB	8	AEF
9	BF	10	CF	11	BAD	12	CAED
13	CAE	14	CF	15	CEDF	16	BAED
17	CBEF	18	CBAD				

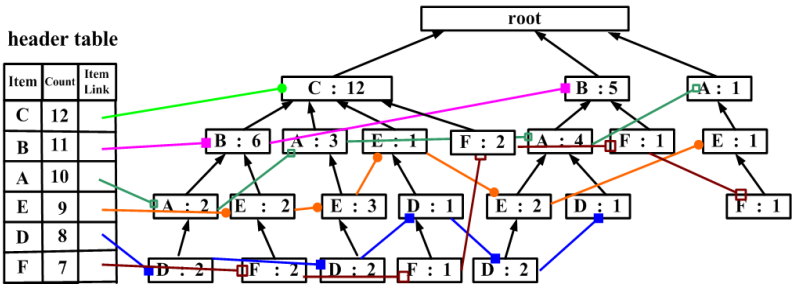


Fig. 1. The FP-tree for Table 1 with minimum support 25%.

port-count and participation-count are registered on each node in the COFI-tree. COFI algorithm generates a large number of candidate itemsets by combing the frequent items from each path in the COFI-tree for an item x and count support for each candidate itemsets to generate all the frequent itemsets with item x. The experiments in [10] have shown that COFI algorithm outperforms FP-Growth. However, the constructed COFI-trees are not condensed and all the combinations of the items in each path of the COFI-tree are generated, such that there are many combinations generated at a time and a large amount of search time needs to be taken to count these combinations.

Algorithm PP-Mine [18] proposes a novel *coded prefix-path* tree (PP-tree) and finds all the frequent itemsets through the constructed PP-tree. PP-tree is constructed as follows. First, PP-Mine scans the database to find all the frequent items. For each transaction, the infrequent items are removed. The remaining frequent items are sorted in descending frequency order and are inserted into PP-tree. PP-tree is similar to FP-tree and is the item-link-free tree structure. Moreover, each node in the PP-tree is arranged by the frequency order and is assigned a calculated code. Although PP-tree does not need to build item-link in the tree initially, all the sibling nodes need to be sorted in a total order. Algorithm PP-Mine mines patterns in a subtree following a depth-first traversal order and all patterns in a subtree will be mined vertically. PP-Mine needs to recursively construct a large number of *sub-header-tables* when push-right and push-down operations occur. For any node in a PP-tree, PP-Mine checks if the itemset from the root to the node is frequent. Push-down is

a depth-first traversal with building a sub-header-table and put the children of the node to the sub-header-table. The push-right strategy is to push the children of the node to their corresponding siblings which lie at the right side of the child nodes. For each push-right operation, PP-Mine needs to search for all the children for a set of nodes from a sub-header-table. Therefore, it faces a large search space if the set of frequent 1-itemsets is large.

The algorithm TFP (mining frequent patterns by Traversing Frequent Pattern tree) [15] is different from FP-Growth algorithm [5, 6] which needs a large amount of memory space to recursively generate conditional FP-trees. TFP first constructs an FP-tree without header table and item-links, and discovers frequent patterns by traversing the constructed FP-tree without building any subtree. TFP applies merging techniques on the tree after generating all the frequent itemsets for a specific item, which makes the FP-tree become smaller and smaller. By this way, TFP can dramatically condense the kernel memory space and reduce the search space without losing any frequent patterns. However, TFP generates a large number of candidate itemsets since the candidates are generated by traversing the large FP-tree which causes a large search space. Moreover, the sub-tree merging is very time consuming since for each merged node, TFP needs to search for all the children of this merged node to find out which children need to be merged.

3. OUR ALGORITHM

In this section, we describe our algorithm SSR for generating frequent patterns. The storage structure of SSR is based on an FP-tree [5, 6]. In order to enhance the efficiency for mining from an FP-tree and generate frequent patterns as fast as possible, we combine the advantages of FP-Growth [6] and Apriori [2], and design an algorithm SSR for mining frequent patterns. SSR first scans the transaction database once to count the support for each item. The frequent items can be obtained if their supports are no less than the minimum support threshold. After generating all the frequent items, SSR constructs an FP-tree [5, 6], which is described in section 2. Therefore, an FP-tree is constructed by the sorted transactions which only contain the frequent items. The item-link for each frequent item in the header table is also built to help searching for the nodes with the same item. After constructing an FP-tree, there are two steps for each frequent item: The first step is to construct an *item-prefix tree* for the frequent item and the second step is to generate frequent itemsets in batch from the constructed item-prefix tree. After generating all the frequent itemsets from an item-prefix tree, the item-prefix tree can be released. Therefore, there is always only an item-prefix tree in the memory at the same time. Our algorithm SSR is shown in **Algorithm SSR** in which **function Item-prefix-tree-construction** is the algorithm for item-prefix tree construction and **function Frequent-pattern-generation** is the algorithm for candidate generation and frequent pattern generation. In the following, we describe the two main steps for our algorithm SSR in details.

3.1 Item-prefix Tree Construction

For any frequent item x in the header table H of the FP-tree T, all the possible frequent itemsets that contain x can be obtained by following item-links for item x, starting

from item-link for item x in the header table of the FP-tree. Let node x denote the node with item x. Therefore, SSR collects each path of the FP-tree T, from the parent node of node x to the child of the root via item-links for item x (lines 2-5 of Algorithm SSR) and the *frequency count p* of every item in the path carries the same count as node x which is attached in the path. For example, from the item-link of item F in the header table of Fig. 1, a path E → B → C:2 can be obtained and the frequency count of every item in the path is $p = 2$. The set of all the collected paths for item x form a small database which is called an *item-prefix pattern base B* for item x (lines 6-7 of **Algorithm SSR**). The item-prefix tree T_x for item x can be constructed from the item-prefix pattern base B for item x. The frequent items in the item-prefix pattern base for item x can also be obtained and are put into the header table H_x of the item-prefix tree T_x. Therefore, the frequent 2-itemsets containing item x can be obtained by combing each item in H_x and item x (lines 9-12 of **Algorithm SSR**). For example, the item-prefix pattern base B for item F is {(E, B, C):2, (D, E, C):1, (C):2, (B):1, (E, A):1}, and the frequent items and their total frequency counts in the item-prefix pattern base B are E:4, B:3 and C:5 if the minimum support count is 2 Therefore, the frequent 2-itemsets containing item F and their support counts in the transaction database (Table 1) are EF:4, BF:3 and CF:5. Then we can find all the frequent itemsets containing item x from the item-prefix pattern base for item x by constructing a small FP-tree which is called item-prefix tree T_x for item x (line 13 of **Algorithm SSR**).

SSR constructs an item-prefix tree T_x for a frequent item x from the item-prefix pattern base B for the item x. The construction of an item-prefix tree T_x for a frequent item x is similar to the construction of an FP-tree T. First, a null root is created for the item-prefix tree T_x. For each record in the item-prefix pattern base for item x, the items whose total frequency counts are no less than minimum support count, in the record are sorted by their total frequency counts in descending order and the other items are removed (line 6 of **Function Item-prefix-tree-construction**). Let the sorted record be $(y_1, y_2, \ldots, y_k):q$ in which q indicates the frequency count of the sorted record, node y_i denote the node with item y_i and node y_0 be the root node of T_x. If node y_{i-1} exists a child node y_i ($1 \le i \le k$) in the item-prefix tree T_x, then the count of node y_i is incremented by q. Otherwise, a child node y_j is created for node y_{j-1} and the count of node y_j is set to q ($\forall j$, $i \le j \le k$) (lines 8-11 of **Function Item-prefix-tree- construction**).

In the following example, we illustrate the item-prefix tree construction for item D. In Fig. 1, the item-prefix pattern base for item D is {(A, B, C):2, (E, A, C):2, (E, C):1, (E, A, B):2, (A, B):1}, and the frequent items in the item-prefix pattern base are A, B, C and E, which means that {DA}, {DB}, {DC} and {DE} are frequent 2-itemsets in Table 1. For the first record (A, B, C):2 in the item-prefix pattern base for item D, a node with item A is created as a child node of the root and the nodes with items B and C are created as the children of nodes A and B, respectively. The counts for the two nodes are 2. The order of the items in the second record is adjusted as (A, C, E):2. Since the root exists a child node A, the count of node A is incremented by 2. A node with item C is created as a child node of node A and a node with item E is created as a child node of item C. The counts of nodes C and E are set to 2 and 2, respectively. After processing all the records in the item-prefix pattern base for item D, the constructed item-prefix tree for item D is shown in Fig. 2.

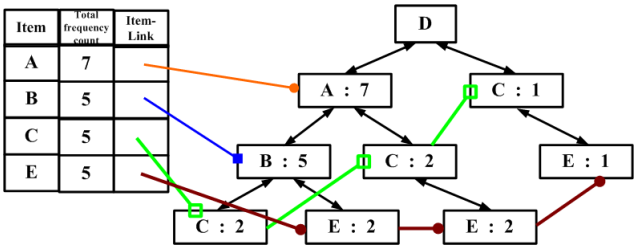


Fig. 2. The item-prefix tree for item D.

3.2 Frequent Pattern Generation

After constructing the item-prefix tree T_x for a frequent item x, the next step is to generate the frequent itemsets with item x. For each item y in the header table of the item-prefix tree T_x for item x, our algorithm SSR retrieves each path from the parent node of node y to the child of the root x in T_x via the item-links of item y and the frequency count of the path carries the same count as node y (lines 2-4 of **Function Frequent-pattern-generation**). For example, from the item-link of item E in the header table of Fig. 2, paths $B \rightarrow A$, $C \rightarrow A$ and C are retrieved and the frequency counts for the three paths are 2, 2 and 1, respectively.

For each retrieved path $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n{:}c$ in which c is the frequency count of the path, our algorithm generates all the combinations of the items in the path, such as $\{a_1\}$, $\{a_2\}$, …, $\{a_n\}$, $\{a_1a_2\}$, …, $\{a_{n-1}a_n\}$, $\{a_1a_2a_3\}$, … and $\{a_1a_2\ldots a_n\}$, and increments the support count for each combination by c (lines 5-6 of **Function Frequent-pattern-generation**). For example, the generated combinations and their frequency counts are $\{B\}$:2, $\{A\}$:2 and $\{BA\}$:2 for path $B \rightarrow A$:2. For each combination, the itemset $\{yx\}$ is implicitly attached to form a candidate itemset, such as $\{ya_1x\}$, $\{ya_2x\}$, … $\{ya_nx\}$, $\{ya_1a_2x\}$, …, $\{ya_{n-1}a_nx\}$, $\{ya_1a_2a_3x\}$, … and $\{ya_1a_2\ldots a_nx\}$. For the above example, the combinations $\{B\}$, $\{A\}$ and $\{BA\}$ mean the three candidates $\{DBE\}$, $\{DAE\}$ and $\{DBAE\}$ and the frequency counts 2 are added to their support counts. Since the number of the candidates about itemset $\{yx\}$, is much fewer than that of the candidates about item x, our algorithm can reduce the search space for the candidate itemsets generated by COFI-tree algorithm [13].

After generating all the combinations of all the paths for each item in the header table of item-prefix tree T_x, all the frequent itemsets with item x can be generated and are put into the set of the frequent patterns (lines 7-9 of **Function Frequent-pattern-generation**). For example, the frequent itemsets with item D and their support counts are $\{DAB\}$:5, $\{DAC\}$:4, $\{DAE\}$:4, $\{DA\}$:7, $\{DB\}$:5, $\{DC\}$:5 and $\{DE\}$:5, if the minimum support count is 4.

Algorithm SSR
Input: An FP-tree T and a header table H, a minimum support threshold *min-sup*, the total number $|TDB|$ of the transactions in the transaction database TDB.
Output: Full set of frequent patterns.

1. Begin
2. For each frequent item x in the header table H of the FP-tree T
3. Let node n be the node pointed by the item-link of x in H

```
        While n is not null and there is a path x1 → x2 → … → xm:p
        from the parent node of n to the child node of the root
        Insert the record (x1, x2, …, xm):p into x's item-prefix pattern base B
        and count the frequency for each item z in the record
        Reset node n to the next node pointed by the item-link of current node n
        For each counted item z
          If the frequency count of z ≥ min-sup × |TDB|
            then insert z into a frequent item list F
            and put {zx} into the set of frequent patterns
        Tx = Item-prefix-tree-construction (B, F)
        Frequent-pattern-generation (Tx)
        Release the space occupied by Tx
    End
```

Function Item-prefix-tree-construction (B, F)
Input: Item-prefix pattern base B for item x and a frequent item list F.
Output: An item-prefix tree Tx.

```
    Begin
      Create a null root of item-prefix tree Tx for item x
      Sort F in frequency count descending order as L and insert the items in L
      into the header table Hx of item-prefix tree for item x
      For each record in item-prefix pattern base B for item x
        Remove the item ∉ L and sort the other items according to the order of L
        Let the ordered record be (y1, y2, …, yk):q and y0 be the root node
        For each i from 1 to k
          If node yi-1 exists a child node yi in Tx
            then increment yi's count by q
          Else create a child node yi for node yi-1 and set the count of yi to q
            Set the parent link of yi to node yi-1
            Join yi to the item-link structure of yi
    End
```

Function Frequent-pattern-generation (T_x)
Input: Item-prefix tree T_x for item x, a minimum support threshold *min-sup*, the total number $|TDB|$ of the transactions in the transaction database *TDB*.
Output: The set of frequent patterns containing item x with length l ($l \geq 3$).

```
    Begin
      For each item y in the header table of Tx
        For each path a1 → a2 → … → an:c from the parent node of y
        to the child node of the root via item-links of item y
          Generate all (2^n − 1) combinations of items in the path and
          increment the frequency count for each combination by c
      For each combination Z
        If the frequency count of Z ≥ min-sup × |TDB|
        Then put {yx} ∪ Z into the set of frequent patterns
      Release the space occupied by the generated combinations
    End
```

4. EXPERIMENTAL RESULTS

For our experiments, we evaluate the performances of FP-Growth [6], COFI [10], PP-Mine [18], TFP [15] and our SSR algorithms. All our experiments were conducted on Intel® Core (TM) 2 CPU 6320 1.86 GHz, 1.99GB memory using C Programming Language and running on Microsoft windows XP environment. Synthetic datasets are generated from IBM Data Generator [19] with parameters as follows: *T* is the average size of the transactions, *I* is the average size of the maximal potentially frequent itemsets and *D* is the number of transactions. The number of the distinct items used for generating all of the synthetic datasets is 2000. In the experiments, we compare our SSR algorithm with the other four algorithms on the six synthetic datasets: T10I2D100K, T10I4D100K, T10I6D100K, T20I4D100K, T20I6D100K and T20I8D100K, which the number of distinct items is 2000, and two real datasets: chess and Mushroom from the public UCI datasets provided on the FIMI workshop website [20].

Figs. 3-5 show the execution times for the algorithms COFI, PP-Mine, TFP, FP-Growth and SSR on the datasets T10I2D100K, T10I4D100K and T10I6D100K with minimum support from 0.1% to 1.0%. The execution times for our algorithm SSR in all the experiments are the total execution times including FP-tree construction, item-prefix-tree construction and frequent-pattern generation. From the three figures, we can see that SSR significantly outperforms the other four algorithms. The performance gaps increase as the

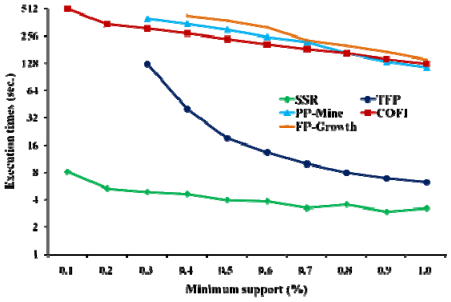


Fig. 3. Execution times for the five algorithms on dataset T10I2D100K.

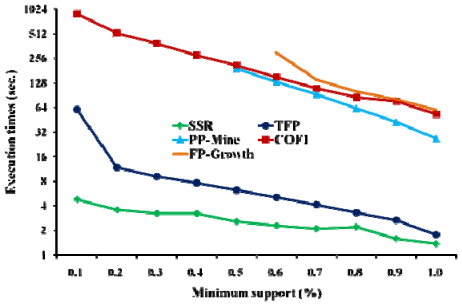


Fig. 4. Execution times for the five algorithms on dataset T10I4D100K.

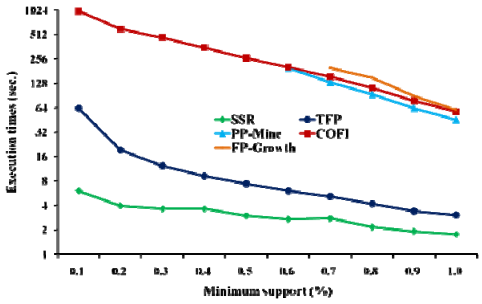


Fig. 5. Execution times for the five algorithms on dataset T10I6D100K.

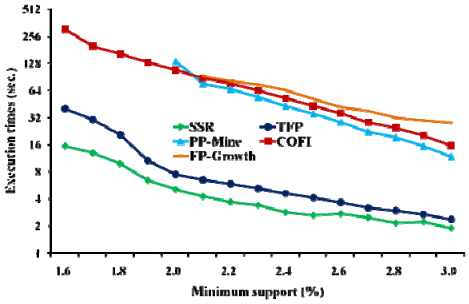


Fig. 6. Execution times for the five algorithms on dataset T20I4D100K.

minimum support decreases, since PP-Mine takes a lot time to recursively build sub-header-tables and search from each sub-header-table when the minimum support decreases. Moreover, the large number of sub-header-tables causes PP-Mine running out of memory when minimum supports are less than 0.3%, 0.5% and 0.6% on the three datasets, respectively. FP-Growth needs to recursively construct conditional FP-tree for each frequent item until the sub-tree includes only one path, such that there are many sub-trees of FP-tree in main memory at a time. It is very time consuming to recursively construct those sub-trees and a large amount of memory space need to be used to store those sub-trees. Therefore, FP-Growth runs out of memory before PP-Mine.

For COFI algorithm, a non-condensed sub-tree is constructed and a huge number of candidates are generated for a frequent item, such that it needs to take a lot of time to generate a large number of combinations to search from these candidates when minimum support is low. Since SSR constructs a compact item-prefix tree for a frequent item and generates candidates in batch, the number of the combinations which need to be searched for SSR is much smaller than that of COFI. Although TFP does not need to construct any sub-tree, it still generates a large number of candidates and a lot of time needs to be taken to search these candidates and merge sub-trees when the minimum support is small.

In addition, since SSR generates a small set of candidates each time for a frequent item, the search space for SSR is much smaller than the other four algorithms. Therefore, the execution times for SSR is much less than the execution times for the other four algorithms and the performances of SSR are quite stable even in low minimum support threshold on the three synthetic datasets. The contributions of our algorithm SSR are that both the search space and used memory space can be significantly reduced, since there is only an item-prefix tree in main memory at a time and SSR generates a small set of the candidates.

Figs. 6-8 show the execution times for the algorithms COFI, PP-Mine, TFP, FP-Growth and SSR on the three synthetic datasets T20I4D100K, T20I6D100K and T20I8D-100K with minimum support from 1.5% to 3.0%. From the three figures, we can see that SSR also significantly outperforms the other four algorithms, since SSR only constructs an item-prefix tree for each frequent item. Instead of recursively generating many sub-trees or sub-header-tables, SSR generates candidates in batch from an item-prefix tree in order to reduce the search space and memory space. PP-Mine takes a lot of time to recursively build sub-header-tables and search from each sub-header-table when the minimum support

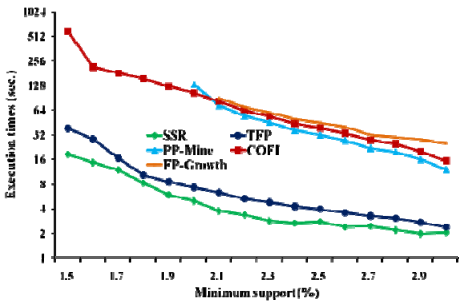


Fig. 7. Execution times for the five algorithms on dataset T20I6D100K.

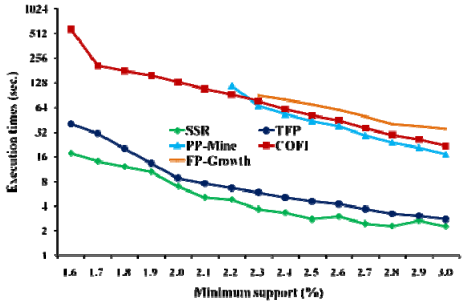


Fig. 8. Execution times for the five algorithms on dataset T20I8D100K.

decreases. Moreover, the large number of sub-header-tables causes PP-Mine running out of memory when minimum supports are less than 2.0%, 2.1% and 2.2% on the three datasets, respectively. FP-Growth also takes a lot of time to recursively construct conditional FP-tree and keeps all the recursively constructed sub-trees in main memory at a time, such that FP-Growth also runs out of memory before PP-Mine.

The memory usages for the five algorithms on the six synthetic datasets are shown in Tables 4 and 5. The recursively generated sub-trees and sub-header-tables cause the large memory space needed by FP-Growth and PP-Mine, respectively, since the two algorithms need to keep all the recursively constructed sub-trees or sub-header-tables in main memory at a time. Our algorithm SSR only constructs an item-prefix tree for an item. After generating all the frequent itemsets for the item, the item-prefix tree for the item can be released. Therefore, there is only an item-prefix tree in main memory at a time, such that the memory space can be reduced. Although SSR needs to create item-prefix pattern bases, the sizes of the sub-trees constructed by SSR are smaller than the sizes of the sub-trees constructed by COFI. TFP does not need to construct any sub-trees and header tables, but the number of candidates generated by TFP is similar to the number of candidates generated by COFI. Since the numbers of the candidates generated by TFP and COFI algorithms are much more than that of SSR each time, the memory space used by TFP and COFI are much larger than SSR. Because SSR generates a small set of candidates each time and the memory space can be released on the next candidate generation, the memory space used by SSR is few.

Table 4. The memory usages and execution times for the five algorithms on the datasets with average transaction size $T = 10$.

Dataset	T10I2D100K		T10I4D100K		T10I6D100K	
Min-sup (%)	2.30		1.18		1.00	
	Memory in MB	Time in Seconds	Memory in MB	Time in Seconds	Memory in MB	Time in Seconds
SSR	0.1	1.25	0.3	1.234	0.5	1.766
TFP	12	2.457	10	2.825	20	3.063
PP_Mine	62	11.125	60	11.437	61	45.672
COFI	13	14.266	11	22.406	21	58.484
FP-Growth	75	16.345	71	23.125	80	59.095

Table 5. The memory usages and execution times for the five algorithms on the datasets with average transaction size $T = 20$.

Dataset	T20I4D100K		T20I6D100K		T20I8D100K	
Min-sup (%)	3.00		3.11		3.20	
	Memory in MB	Time in Seconds	Memory in MB	Time in Seconds	Memory in MB	Time in Seconds
SSR	0.2	1.891	0.1	1.782	0.1	1.875
TFP	15	2.797	13	2.259	14	2.305
PP_Mine	62	11.813	60	9.641	59	11.14
COFI	16	15.656	14	12.563	15	14.594
FP-Growth	75	20.936	68	13.412	66	13.134

In order to evaluate the performance of our algorithm SSR and the other algorithms on the datasets with larger number of items, we reset the number of items to 5000 and 10000, and generate the two synthetic datasets T10I4D100K with number of items 5000, say T10I4D100K (item 5K), and T10I4D100K with number of items 10000, say T10I4D-100K (item 10K). We perform the experiments on the two synthetic datasets.

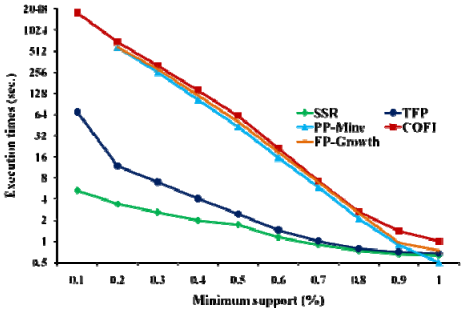


Fig. 9. Execution times for the five algorithms on dataset T10I4D100K (item 5K).

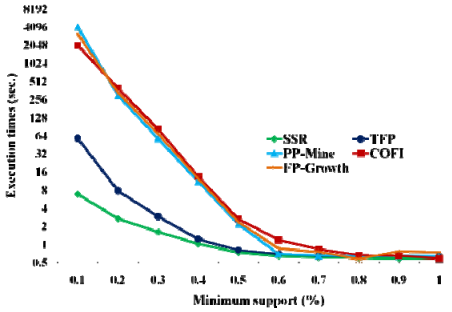


Fig. 10. Execution times for the five algorithms on dataset T10I4D100K (item 10K).

Figs. 9 and 10 show the execution times for the five algorithms on the two datasets, from which we can see that SSR also significantly outperforms the other four algorithms even the number of the distinct items is large, since SSR also generates a small set of the candidates each time. Since the dataset T10I4D100K (item 10K) is more sparse than the dataset T10I4D100K (item 5K), the number of the generated frequent itemsets on dataset T10I4D100K (item 10K) is less than that on dataset T10I4D100K (item 5K), but the number of the frequent items on T10I4D100K (item 10K) is more than that on T10I4D100K (item 5K). Therefore, the numbers of the sub-trees constructed by COFI and SSR for the frequent items are large when the number of the items is large. However, COFI still generates a lot of candidates to be searched for each sub-tree. SSR generates much fewer candidates than COFI each time. Although the number of the item-prefix trees on T10I4D100K (item 10K) is more than that on T10I4D100K (item 5K), the search space for the candidates on T10I4D100K (item 10K) is smaller than the search space on T10I4D100K (item 5K). Therefore, the execution times for SSR are similar on the two datasets. For PP-Mine and FP-Growth, although the number of the frequent items on T10I4D100K (item 10K) is more than that on T10I4D100K (item 5K), the number of the recursive calls, that is the number of the constructed sub-header-tables or sub-trees, for each frequent item on T10I4D100K (item 10K) is fewer than that on T10I4D100K (item 5K). Therefore, the two algorithms do not run out of memory even the minimum support is 0.1% on T10I4D100K (item 10K), but they run out of memory when the minimum support is less than 0.2% on T10I4D100K (item 5K).

Figs. 11 and 12 show the execution times for COFI, PP-Mine, TFP and our SSR algorithms on the two real datasets Chess and Mushroom, respectively. In the two experiments, since there are many frequent items and many siblings for each node on the PP-tree, PP-Mine creates a large number of sub-header-tables for push down operations and search for lots of items from each sub-header-table for push right operations. Therefore, the per-

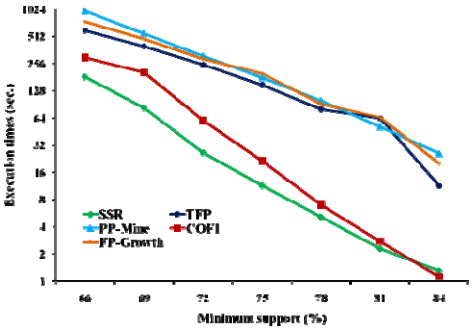


Fig. 11. Execution times for the five algorithms on the real dataset chess.

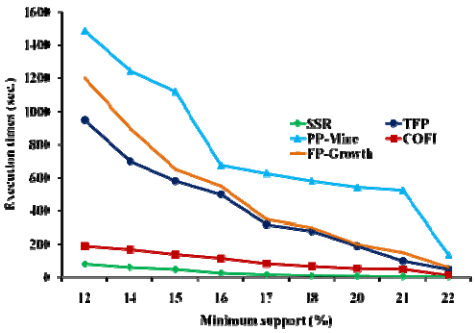


Fig. 12. Execution times for the five algorithms on the real dataset mushroom.

formance for PP-Mine degrades as the minimum support threshold decreases. On Mushroom dataset, since there are only a few candidates generated, the search space for COFI is small. Therefore, the execution times for COFI are near to SSR in Fig. 12. Since TFP needs to take more time to merge the sub-trees in a large FP-tree than to construct the sub-trees for COFI, the algorithm COFI is more efficient than TFP. For our SSR algorithm, although the sizes of the constructed sub-trees are smaller than that of COFI, it needs to take time to create item-prefix pattern base for each frequent item. Therefore, in the case of few candidates generated, the performances of COFI and SSR are similar.

5. CONCLUSIONS

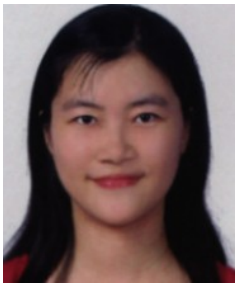
In order to avoid recursively constructing sub-trees for FP-growth [5, 6], three efficient approaches PP-Mine [18], TFP [15] and COFI [10] have been proposed for mining frequent patterns. Although PP-Mine does not need to construct any sub-tree, many sub-header-tables need to be recursively created for PP-Mine and many search operations need to be performed on each created sub-header-table. TFP algorithm also does not need to construct any sub-tree, but a large number of candidates need to be generated and many sub-tree merges need to be performed. For COFI algorithm, a large non-compact sub-tree for each frequent item needs to be constructed and a large number of candidates need to be generated from the large sub-tree for the frequent item. FP-Growth algorithm needs to recursively construct conditional FP-trees for each frequent item, such that it is very time consuming to recursively construct these sub-trees and a large amount of memory space need to be used to store these sub-trees.

Therefore, we propose an efficient algorithm SSR for mining frequent patterns to improve the above three algorithms. Our SSR algorithm first constructs a compact sub-tree for a frequent item and then generates candidates in batch from the compact sub-tree. Because SSR generates a small set of candidates every time, the search space is much smaller than COFI and TFP. Therefore, SSR can significantly reduce the search space. The experimental results also show that our algorithm SSR significantly outperforms the other three algorithms on both execution times and memory usages.

REFERENCES

1. R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad, "A tree projection algorithm for generation of frequent itemsets," *Journal of Parallel and Distributed Computing*, Vol. 61, 2001, pp. 350-371.
2. R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proceedings of International Conference on Very Large Data Bases*, 1994, pp. 487-499.
3. Y. Bastide, L. Lakhal, N. Pasquier, G. Stumme, and R. Taouil, "Mining frequent patterns with counting inference," *ACM SIGKDD Explorations Newsletter*, Vol. 2, 2000, pp. 66-75.
4. S. Brin, R. Motwani, J. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 1997, pp. 255-264.
5. J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD Record*, Vol. 29, 2000, pp. 1-12.
6. 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*, Vol. 8, 2004, pp. 53-87.
7. M. Houtsma and A. Swami, "Set-oriented mining for association rules in relational databases," in *Proceedings of International Conference on Data Engineering*, 1995, pp. 25-33.
8. C. K. S. Leung, P. P. Irani, and C. L. Carmichael, "FIsViz: A frequent itemset visualizer," in *Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining*, LNCS 5012, 2008, pp. 644-652.
9. H. Mannila, *et al*., "Efficient algorithm for discovering association rules," in *Proceedings of AAAI Workshop on Knowledge Discovery in Databases*, 1994, pp. 181-192.
10. M. El-Hajj and O. R. Zaiane, "Non recursive generation of frequent K-itemsets from frequent pattern tree representation," in *Proceedings of the 5th International Conference on Data Warehousing and Knowledge Discovery*, 2003, pp. 371-380.
11. J. S. Park, M. S. Chen, and P. S. Yu, "An effective hash-based algorithm for mining association rules," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Vol. 24, 1995, pp. 175-186.
12. S. Sarawagi, S. Thomas, and R. Agrawal, "Integrating association rule mining with relational database systems: Alternatives and implications," in *Proceedings of SIGMOD International Conference on Management of Data*, 1998, pp. 343-354.
13. A. Savasere, E. Omiecinski, and S. Navathe, "An efficient algorithm for mining association rules in large databases," in *Proceedings of International Conference on Very Large Data Bases*, 1995, pp. 432-443.
14. S. J. Yen and A. L. P. Chen, "A graph-based approach for discovering various types of association rules," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 13, 2001, pp. 839-845.
15. S. J. Yen, Y. S. Lee, C. K. Wang, J. W. Wu, and L. Y. Ouyang, "The studies of mining frequent patterns based on frequent pattern tree," in *Proceedings of the 13th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, LNAI 5476, 2009, pp. 232-241.
16. M. Zaki, "Scalable algorithms for association mining," *IEEE Transactions on Knowl-*

edge and Data Engineering, Vol. 12, 2000, pp. 372-390.
17. O. R. Zaiane, M. El-Hajj, and P. Lu, "Fast parallel association rule mining without candidacy generation," in *Proceedings of IEEE International Conference on Data Mining*, 2001, pp. 665-668.
18. Y. Xu, J. X. Yu, G. Liu, and H. Lu, "From path tree to frequent patterns: A framework for mining frequent patterns," in *Proceedings of IEEE International Conference on Data Mining*, 2002, pp. 514-521.
19. IBM Almaden, "Quest synthetic data generation code," http://www.almaden.ibm.com/cs/quest/syndata.html.
20. http://archive.ics.uci.edu/ml/datase.



Show-Jane Yen (顏秀珍) is an Associate Professor in the Department of Computer Science and Information Engineering in Ming Chuan University. She received the M.S. degree and the Ph.D. degree in Computer Science from National Tsing Hua University, Hsinchu, Taiwan, in 1993 and 1997, respectively. Her research interests include database management systems, data mining and data warehousing.

Chiu-Kuang Wang (王秋光) received the M.S. degree in Computer Science from Mankato State University, Minnesota, U.S.A. in 1984. She is a full time lecturer in the Department of Computer Science and Information Engineering at Ming Chuan University in Taiwan. She is currently a doctoral candidate at the Department of Management Sciences at Tamkang University in Taiwan.

Liang-Yuh Ouyang (歐陽良裕) is a Professor in the Department of Management Sciences at Tamkang University in Taiwan. He earned his B.S. in Mathematical Statistics, M.S. in Mathematics and Ph.D. in Management Sciences from Tamkang University. His research interests are in the field of production/inventory control, probability and statistics.