International Journal of Computer Information Systems and Industrial Management Applications (IJCISIM)
ISSN 2150-7988 Vol.1 (2009), pp. 58–67
http://www.mirlabs.org/ijcisim

Efficient Mining of Frequent Closed Itemsets without Closure Checking

Chih-Hsien Lee[1], Kuo-Cheng Yin[1,2], Don-Lin Yang[1], Jungpin Wu[3]
[1]*Dept. of Information Engineering and Computer Science, Feng Chia University, Taiwan*
[2] *Dept. of Information Management, Jen-Teh Junior College, Taiwan*
[3]*Dept. of Statistics* and *Dept. of Public Finance, Feng Chia University, Taiwan*
bartscott@selab.iecs.fcu.edu.tw, inn@selab.iecs.fcu.edu.tw, {dlyang, cwu}@fcu.edu.tw

Abstract

Most existing algorithms for mining frequent closed itemsets have to check whether a newly generated itemset is a frequent closed itemset by using the subset checking technique. To do this, a storing structure is required to keep all known frequent itemsets and candidates. It takes additional processing time and memory space for closure checking. To remedy this problem, an efficient approach called closed itemset mining with no closure checking algorithm is proposed. We use the information recorded in an FP-tree to identify the items that will not constitute closed itemsets. Using this information, we can generate frequent closed itemsets directly. It is no longer necessary to check whether an itemset is closed or not when it is generated. We have implemented our algorithm and made many performance experiments. The results show that our approach has better performance in the runtime and memory space utilization. Moreover, this approach is also suitable for parallel mining of frequent closed itemsets.

Keyword: *Data mining, association rule, frequent closed itemset, closure checking*

1. Introduction

Data mining is an important research field for finding information in a large volume of data [13]. The function of data mining is to find important information that can be used to make decisions and action plans. Data mining has already been applied extensively in various industries.

Association rule mining is a useful data mining technique to find frequent itemsets and generate useful association rules [12]. Generating all frequent itemsets [1-3], [15-16] in brute force is not an efficient task, so many closed itemsets [4-10] and maximal itemsets [11] approaches were derived. However, the information of maximal itemsets is incomplete such that association rules cannot be generated directly. Frequent closed itemsets can solve the problems that frequent itemsets and frequent maximal itemsets have. A frequent itemset is closed if none of its proper supersets have the same support. All closed frequent itemsets contain complete information to generate association rules.

However, most known frequent closed itemset mining algorithms must check the closure at the end of the process or during the process [4-6], especially for the algorithms using the horizontal dataset directly. The closure checking examines whether a newly found itemset is a subset or a superset of an already found frequent closed itemset with the same support. For this purpose, it is necessary to create a structure to store all frequent closed itemsets and closed itemset candidates. The closure checking step of the existing closed itemset mining algorithms is computational expensive and requires additional memory space for generating, storing and removing non-closed itemsets.

Our work completely eliminates the closure checking step during the closed itemset generation. We propose an algorithm called Closed Itemsets Mining with No closure Checking (CIMNC) to directly produce frequent closed itemsets without closure checking and unnecessary storage structure. CIMNC won't generate unnecessary conditional FP-trees [4] if possible. We only use an attribute called *record* in the node of FP-tree to keep necessary information in each branch of the tree such that redundancy can be avoided. Especially, only frequent closed itemsets are produced and each one is produced exactly once. Since CIMNC can generate frequent closed itemsets directly in each processor, it is also suitable for

parallel mining.

2. Related work

The FP-growth method [2] is a depth-first and divide-and-conquer algorithm. In this method, a structure called FP-tree is used to obtain a compact representation of the original transactions. Every branch of the FP-tree represents a transaction composed of the subset of frequent items. The nodes along the branches are stored in decreasing order of the frequency of all frequent items. Compression is achieved by overlapping itemsets which share prefixes of the corresponding branches to build the FP-tree. The FP-tree has sufficient information to mine complete frequent patterns. Each node in the FP-tree has three fields: *item-name*, *count*, and *node-link*. The frequent itemsets can be found from the FP-tree quickly without having to scan the database on the disk frequently. A frequent-item header table is built to make traverse the tree more easily. All frequent items are stored in the header table in decreasing order of their frequency. Each item points to its occurrence in the tree via a head of node-link. Nodes with the same item are linked via node-links. Each entry in the header table has two fields: *item-name* and *head of node-link*.

The FP-growth method scans the database only twice. In the first scan it finds all frequent items and inserts them into the header table in decreasing order of their counts. In the second scan, the root of FP-tree is created with "null." The set of frequent items in each transaction is inserted into the FP-tree as a branch. If an itemset has the same prefix with another itemset already in the tree, this part of branch will be shared. A count in a node stores the number of the item which appears in this path. When a transaction is inserted into a new branch, the count is updated. After all nodes are linked from the header table, the FP-tree is completely constructed. The next step is to find all frequent itemsets. It collects all the patterns which a node participates by starting from its head in the header table and following its node-link. The mining process starts from the bottom of the header table. Paths with the same prefix item in the FP-tree construct the *conditional pattern base* of the prefix item with its support. Frequent items in the conditional pattern base construct the *conditional FP-tree* of the prefix item. It keeps constructing the conditional FP-tree until a single path is found. Frequent itemsets with the same prefix are generated by the single path.

The main work of FP-growth method is traversing FP-trees and constructing new conditional FP-trees from the global FP-tree. It needs to traverse the original FP-tree twice to construct a new conditional FP-tree. The first traversal finds all frequent items in the conditional pattern base and constructs a new header table for new conditional FP-tree. The second traversal constructs the new tree. FPclose [4] can omit the first traversal by adopting an FP-array technique.

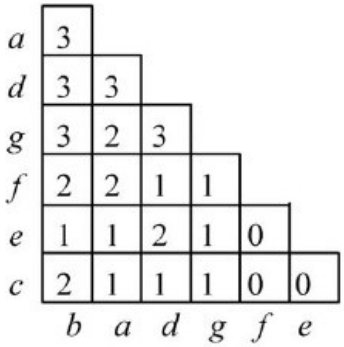


Fig. 1 An FP-array example.

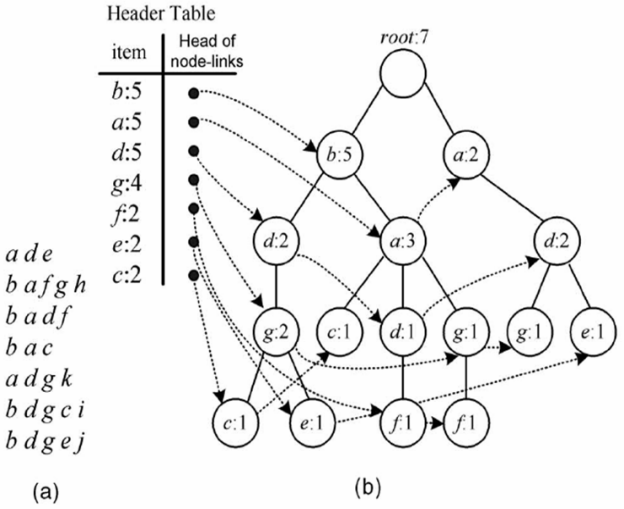


Fig.2 An FP-tree example.

Using the example of Fig. 1 from [4], an array structure called FP-array is used in this method. During the second scan of the database of Fig. 2(a), it constructs the FP-tree and the FP-array as shown in Fig. 2(b) and Fig. 1 respectively. The mining method of FPclose [4] is different from CLOSET+ [6]. It won't do subset checking to avoid generating redundancies of a prefix, but generates a lot of candidates to check their closure. A structure called CFI-tree [4], as shown in Fig. 3, is used to store frequent itemsets efficiently and to do closure checking.

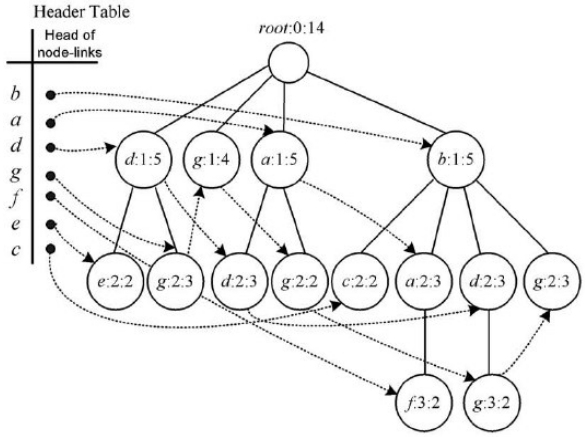


Fig. 3 Construction of CFI-tree

The CLOSET+ algorithm [6] is based on the FP-tree and uses the *two-level hash indexed result tree* to store frequent closed itemsets for closure checking. When it finds a single path in the conditional FP-tree, it must perform *subset checking* to check if it should generate the closed itemset candidate with a prefix itemset from all known frequent closed itemsets. If the prefix itemset is a subset of a known frequent itemset and has the same support, the itemsets of this prefix itemset will not be generated. On the other hand, the closed itemset candidates are generated and then it must check the closure in the *result tree* to determine if they are really frequent closed itemsets. CLOSET+ needs to do many times of subset checking and closure checking. Since the result tree is searched many times, it takes a lot of time.

The processes of CLOSET+ algorithm have seven steps.

Step 1:

Scan the database to find the counts of all items in the database. Find all frequent items by using the *min_sup* and sort these items with their supports in decreasing order. Build the header table with *item-name*, *support count* and *head of node link* to store the frequent items.

Step 2:

Scan the database again to build the FP-tree, where each node has the fields of *item-name*, *count*, and *node-link*, according to the order of the header table. Each node of the same item-name is linked from the header table by node-links.

Step 3:

According to the order of items in the header table, it gets an item as the prefix item each time. Merge the prefix item and the prefix item of this tree as a prefix itemset. Check if the prefix itemset with the support of the prefix item is in the two level hash indexed result tree. If not, get all paths which contain the prefix item linked from the header table. Set the support of the prefix node as the support of all items in the path to form the conditional pattern base.

Step 4:

Prune the items which are not frequent from the conditional pattern base. And then use the remaining items to build the header table. Finally create the conditional FP-tree with the header table. Set the next item in the header table as the prefix item. Use Step 5 to process the conditional FP-tree. And repeat Step 3 and Step 4 until the last item of the header table is completed.

Step 5:

According to the order of items in the header table of the conditional FP-tree, it can get an item as the prefix item each time. Repeat Step 3 and Step 4 to create the conditional FP-tree for each prefix item until the conditional FP-tree becomes a single path.

Step 6:

Closed itemset candidates can be generated from the single path with the prefix items of the conditional FP-trees which have been processed before. Check if an itemset is a closed itemset using the two-level hash indexed result tree method (for simplicity we assume dense datasets). Store the itemset in the tree if it is closed.

Step 7:

After processing the last item of the original header table, we can obtain all frequent closed itemsets.

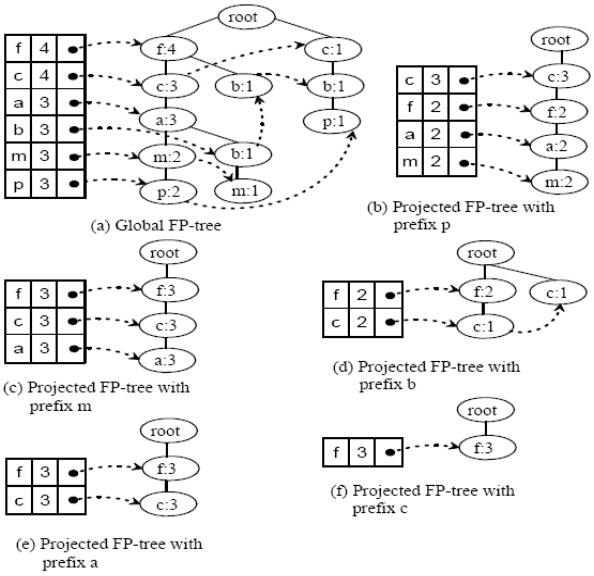


Fig. 4 Bottom-up physical tree-projection [6]

An example from [6] for the bottom-up physical tree-projection is shown in Fig. 4 where the *min_sup* is 2.

3. Proposed algorithm: CIMNC

A. Main Concept of CIMNC

Mining itemsets in bottom-up order of item supports is the property of FP-tree. One can not find supersets of the found frequent closed itemsets by using this property. If it can avoid generating the subsets of known frequent closed itemsets, frequent closed itemsets can be generated directly.

According to the properties of FP-tree and closed itemset, it is easy to find that if an itemset is a subset of the other known itemset and both of their prefix items have the same support, they are both found in the same path. Therefore, if a branch has more than two items that have the same support, it may generate subsets with the same support. In order to avoid generating these subsets, it should keep off items which have been processed with the same support in the same path and avoid constructing them again.

A *record* is used in CIMNC to identify if an item has been processed before. If the nodes of an item have the same item in the *record*, it means the item has completed the process. Using this idea, subsets with the same support won't be generated. Thus, it can speed up the run time by not building the conditional FP-tree for invalid items.

B. The CIMNC Algorithm

Based on CLOSET+, CIMNC algorithm has seven steps:

Step 1:

Scan the database to find the counts of all items. Find all frequent items by using the *min_sup* and sort them in decreasing order of their supports. Build the header table with *item-name*, *support count* and *head of node link* fields.

Step 2:

Scan the database again to build the FP-tree, in which each node has four fields: *item-name*, *count*, *node-link* and *record*, according to the order of the header table. Each node with the same item-name is linked together.

Step 3:

According to the order of items in the header table, each item is used as the prefix item one at a time. If all nodes of the prefix item do not have the same items in their records, get all paths containing the prefix item being linked from the header table. When getting a path, the records in this path are noted if the prefix item is a leaf of the tree. Then set the support and record of the prefix node as the support and record of all items in the path to form the conditional pattern base.

Step 4:

Prune the items which have the same item in their records and the items which are not frequent from the conditional pattern base. Then, use the remaining items to build the header table. Finally, create the conditional FP-tree with the header table. Set the next item in the header table as the prefix item. Use Step 5 to process the conditional FP-tree. Repeat Step 3 and Step 4 until the last item of the header table is processed.

Step 5:

According to the order of items in the header table of the conditional FP-tree, each item is used as the prefix item one at a time. Repeat Step 3 and Step 4 to create the conditional FP-tree for each prefix item until the conditional FP-tree becomes a single path.

Step 6:

Note the records for this path and obtain all itemsets whose prefix items have no item in their records. It can get closed itemsets from the single path with the prefix items of the conditional FP-trees which have been processed before.

Step 7:

After processing the last item in the original header table, it generates all frequent closed itemsets.

C. Two Lemmas of CIMNC

If the support of a prefix item is equal to the support of the parent item, they will be constructed completely when the prefix item generates the conditional pattern base. After these items have been processed, our method uses the *record of the node* to note items that have the same support as their child items in the same branch. Using an array as a record in a node of FP-tree will waste much space. Instead, the linked list technique is used to store items. When considering an item if it is necessary to build the conditional FP-tree or to perform closure checking, one can simply check if the records in all the nodes of this item have the same item. If they all have the same item, there is no need to process again because it has been done before. So itemsets generated from this item are not closed.

Lemma 1. If all nodes of an item have the same items in their *records*, the itemsets generated by this item are not closed.

Proof: Let X be a prefix item and all the X nodes of the tree have the same item Y in their *records*. Assume X may generate a closed itemset, then the support of the itemset generated by X will not equal to the support of the superset generated by XY. Because all records of X contain Y, the itemsets generated by X must be subsets of the itemsets generated by XY, and their support must be

the same. It contradicts the assumption. So X can't generate any closed itemset. ■

After building the FP-tree, each item is considered as a prefix item following the order of items in the header table. When it searches the related paths of a prefix item by using the node-links of the header table, the records of these paths are completed.

The way to note the record is shown as follows. When the node-link of a prefix item is linked to the first node from the header table, it can find the first path which contains the prefix item. When searching this path upward to look for items in the path, it can check if its child nodes have a total count that is equal to its count. If their counts are not the same, it won't note anything. Otherwise, if the node has only one child node, it can note the item and record its child node in its *record*. If the node has more than one child node, it should check if any node has the same items in item-name or *records*. If each branch has the same items, it will note the item in the record of the node. Otherwise, nothing is noted. Repeat this process to the root until every node of the prefix items of the header table has been processed. If a node of a prefix item is not at the bottom of the path, it means this path has been noted before. A simple example is shown in Fig. 5.

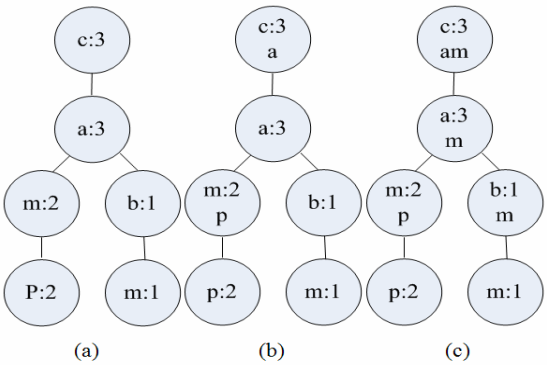


Fig. 5 The recording method.

Fig. 5(a) is a part of the original FP-tree in Fig. 4(a). When searching the prefix item *p*, it can find the path *camp*. The path is processed as shown in Fig. 5(b). First, go up and find the item *m*, then note *p* in the record of node *m* because it has the same support as its child *p*. The next item *a* has two child nodes, where their total support is equal to the support of *a*. Since none of their item and record is the same, it does not note anything. Then it reaches item *c*. Because item *c* has the same support as item *a*, its record is updated with a note of item *a*.

Next, the prefix item *m* is searched and the result is shown in Fig. 5(c). Item *b* is the first item above item *m* and its support is equal to the support of item *m*, so item *b* is updated with a note of item *m*. The next item *a* has two child nodes, where their total support is equal to the support of *a*. Since they both have a record of *m*, item *m* is noted in node *a*. Because node *a* adds a record *m*, the parent node *c* is also updated with a note of *m* to its node because the pair has the same support.

The way to use the records is briefly explained as follows. When searching the paths for a prefix item, it will set the support and records of the prefix item to all items in a path. If each record of the prefix item does not have the same items, the process of this item will continue, else it will stop.

When getting the conditional pattern base of a prefix item, it can prune the infrequent items. An item will be pruned if all of its nodes have the same item(s) in their records, because they can not generate any closed itemset by Lemma 1. When getting a single path in the conditional FP-tree, it is easy to know if the itemsets have to be generated by the records.

The tree of Fig. 5(c) is used to show a simple example. With a prefix item *m*, it can get two paths, *cam* and *cabm*. The conditional pattern base has two prefix paths, *ca*:2/*p*(*ca*:2 with a record *p*) and *cabm*:1. Because no item has the same records, it can derive *m*'s conditional FP-tree, <*c*:3, *a*:3>. If the prefix item is *a*, it can get a path *ca*. Because the conditional pattern base only has a path *c*:3/*m*, it is unnecessary to generate *a*'s conditional FP-tree.

It is easy to prove that our method won't generate more or less itemsets by the following Lemma 2.

Lemma 2. CIMNC won't generate non-closed itemsets or miss any closed itemsets.

From Sections 2 and 3.B, one knows the process of CIMNC is almost the same as the steps of CLOSET+. CIMNC uses a record to identify if an item with the item in its record can generate a closed itemset by Step 3 and Step 4. CLOSET+ checks a prefix itemset from the two level hash indexed result tree to identify if the itemset can generate a closed itemset by Step 3 and Step 6. This shows that CIMNC can generate all closed itemsets as CLOSET+ does. Since CLOSET+ won't generate extra itemsets that are not closed and miss any closed itemsets, CIMNC should have exactly the same properties as well.

For example, when CLOSET+ processes the prefix item *m* in Fig. 4(c), the two level hash indexed result tree has two paths, <*f*:2, *c*:2, *a*:2, *m*:2, *p*:2> <*c*:3, *p*:3>. When it begins to generate the itemsets with the prefix item *a*, it checks the prefix itemset *am*:3 to identify if any path has this itemset with a support of 3. However, the result tree

has no item *m* with support 3, so itemsets with the prefix itemset will be generated. The result tree has an additional path <*f*:3, *c*:3, *a*:3, *m*:3>. The next prefix item *c* will be merged with item *m* as prefix itemset *mc*:3, and *mc*:3 can be found in the result tree. So it is not necessary to continue the process. The next item *f* is unnecessary to be processed. For CIMNC, the first item *a* has no record, so it generates the itemsets with *a*. For the next item *c*, because it has a record *a*, meaning *ac*:3 has been processed, it is unnecessary to be processed again. The record of item *f* is not empty, so it is not necessary to be processed. Although these two algorithms use different methods to identify the prefix item, their processes are the same. One can conclude that they all generate the same result.

D. An Example of applying CIMNC

The database in Table 1 is used to present a simple example for CIMNC. After scanning the database, it finds all frequent items with *min_sup*=2. They are *a*:3, *c*:4, *f*:4, *m*:3, *p*:3, *b*:3. Then sort these items with support in decreasing order to get *f*:4, *c*:4, *a*:3, *b*:3, *m*:3, *p*:3. When transactions are inserted into the FP-tree, items in each transaction will be sorted by the order as shown in the last column of Table 1. To find each item in all branches easily, a header table is created with the sorted items in Fig. 6.

Table 1 A sample transaction database

Tid	Items	Ordered frequent item list
100	a, c, f, m, p	f, c, a, m, p
200	a, c, d, f, m, p	f, c, a , m, p
300	a, b, c, f, g, m	f, c, a, b, m
400	b, f, i	f, b
500	b, c, n, p	c, b, p

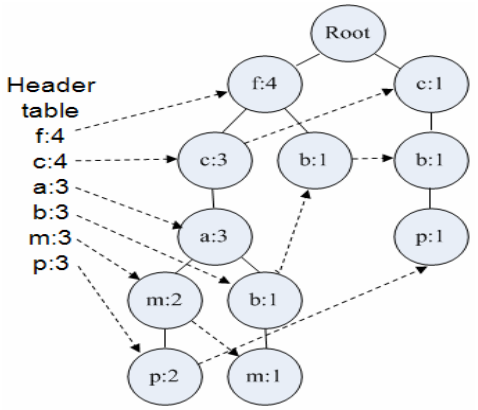
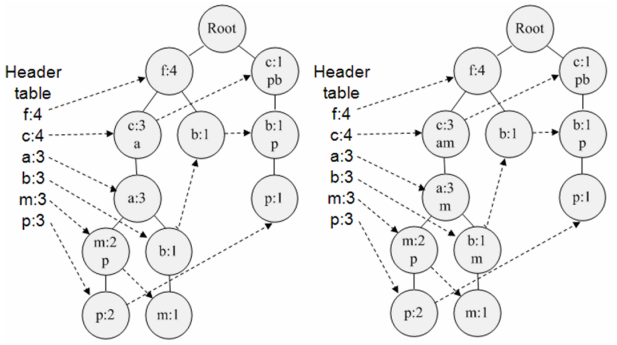


Fig. 6 The global FP-tree for Table 1.

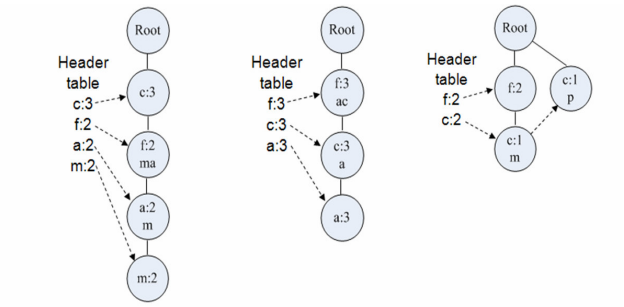
After scanning the database again, the global FP-tree is built in Fig. 6. All nodes which have the same item-name are linked from the header table. After constructing the FP-tree, it can be traversed to find all frequent closed itemsets. According to the FP-growth method, it can take item *p* as a prefix itemset to get two paths, *fcamp* and *cbp*, by using the node links of item *p* in the header table. When a node is linked, it will search the path to find what items are in this path. At this time, it will note items in the record of each node if a node has the same support as the items below it.



(a) The recorded global FP-tree with prefix item p. (b) The recorded global FP-tree with prefix item m.

Fig. 7 The recorded global FP-tree.

In Fig. 7(a), it notes *p* in the record of item *m* because item *m* has the same support as prefix item *p* in the path *fcamp*. Similarly, item *c* has a record *a* in its node. In the path *cbp*, it notes *p* in the node *b* and notes *pb* in the node *c*. The conditional pattern base of prefix item *p* has *fcam*:2 and *cb*:1 to create the conditional FP-tree shown in Fig. 8(a). The conditional FP-tree only contains frequent items and it also has a header table. We can see only one path in this tree. Following the bottom-up order in the header table, we first make item *m* as a prefix item, and find what items are above it. At this time, we note the record in each node of this tree, as shown in Fig. 8(a). It can find a frequent closed itemset *cfamp*:2 in this tree with prefix *m*. Then it makes item *a* as a prefix. Because it has an item *m* in its record of the only one node, one knows it has been mined with prefix *m*. It won't generate itemsets with prefix *pa*. Similarly, the prefix *pf* won't be used neither. Finally, item *c* is the last item and its record is empty. So we can find *cp*:3 as a frequent closed itemset. Because *c* is the last item and the support of *c* is equal to the support of *p*, *p*:3 is not a closed itemset. At this time, all frequent closed itemsets with prefix *p* have been found.



(a) The conditional FP-tree for prefix p. (b) The conditional FP-tree for prefix m. (c) The conditional FP-tree for prefix b.

Fig. 8 The conditional FP-tree.

Then it makes item *m* as a prefix of the global FP-tree. When searching the linked node of item *m*, it notes the records for nodes with item *m* having no child. There are two paths with prefix *m*, *fcam* and *fcabm*. In Fig. 7(b), the node *m* in path *fcam* has a child, so it is unnecessary to note records in this path. In path *fcabm*, it notes *m* in node *b*, notes *a* in node *c* and adds *m* to node *c*. The conditional pattern base containing *fca*:2/*p* and *fcab*:1 can form a conditional FP-tree as shown in Fig. 8(b). Then note the records in the tree when it makes item *a* as a prefix item. It can find a frequent closed itemset *fcam*:3, and itemsets with prefix items *c* and *m* are not generated. Because *f* is the last item and the support of *f* is equal to the support of *m*, *m*:3 is not a closed itemset. At this time, all frequent closed itemsets with prefix *m* have been found.

The third item *b* is used as a prefix item. It can get three paths, *fcab*, *fb* and *cb*. Each one of the paths *fcab* and *cb* has a child and the supports in path *fb* are not the same, so it is unnecessary to note records of the tree. The conditional pattern base, *fca*:1/*m*, *f*:1 and *c*:1/*p*, builds the conditional FP-tree as shown in Fig. 8(c). According to the header table, item *c* is the first prefix item. It is unnecessary to note records here because no support is the same in each path. It can find two paths *fc*:1 with record *m* and *c*:1 with record *p*. Only item *c* is frequent and the records of two paths are not the same, so *cb*:2 is a frequent closed itemset. Since *f* is the last item of the header table and it has no record, *fb*:2 is a frequent closed itemset. Because *f* is the last item and the support of *f* is not equal to the support of *b*, *b*:3 is a frequent closed itemset. At this time, all frequent closed itemsets with prefix *b* have been found.

Because prefix item *a* has only one path and it has a record *m*, it is unnecessary to create the conditional FP-tree with prefix item *a*. The prefix *a* will not generate any closed itemset.

For prefix item *c*, it has two paths, *fc* and *c*. The conditional pattern base, *f*:3/*am*, and the other path have nothing, so it is unnecessary to create the conditional FP-tree for prefix item *c* with item *f*. And *c*:4 is a frequent closed itemset.

Finally, the last item *f* has no record, so *f*:4 is a frequent closed itemset. At this time, the mining process is completed.

E. Parallel Method

Since FP-tree can be used in the parallel environment, CIMNC is suitable to mine frequent closed itemsets in a parallel manner. In addition, CIMNC can generate frequent closed itemsets directly; the mining process can speed up very well by using parallel techniques.

Each computer handles some items as prefix items in the parallel environment. When the application sends the data and commands to each computer, the percentage of continuous frequent items is allocated for each computer in the command. After the FP-tree is completed in each computer, it can traverse the FP-tree once to note records, or only check its child notes if they have the same support to decide the search range as an optimal way. If a child node is not one of the items allocated to this computer, it needs to note records if it is a valid item which corresponds to CIMNC. Otherwise, it has been noted before. After the downward noting process is done, the upward noting process starts until the root is reached. After all nodes of this prefix item have been processed, it begins building the conditional FP-tree repeatedly until deriving all frequent closed itemsets with this prefix item. When all items which are allocated to this computer are processed, all frequent closed itemsets with these prefix items are all generated. When all computers send back their results to the application, it is unnecessary to check the closure. This can save a lot of time. For an algorithm which needs to do closure checking, it would have to wait for each other because one does not know which computer will finish first. The closure checking in the parallel environment needs to do either subset checking or superset checking. It can derive all frequent closed itemsets with closure checking after all computers finish their work.

For example, assume three computers, namely A, B, and C, are used in the parallel environment and A is the slowest one of three computers. The runtime of CIMNC is the same as the runtime of computer A since CIMNC does not require closure checking after collecting information from all three computers. This is shown in formula (1). For a method requiring closure checking, the runtime is expressed in formula (2) where time $_{\text{closure checking}}$ is the time of closure checking after collecting the result from all three computers.

$$\text{Run time}_{\text{CIMNC}} = \text{run time}_{\text{CIMNC}}(A) \quad (1)$$

$$\text{Run time}_{\text{non-CIMNC}} = \text{run time}_{\text{non-CIMNC}}(A) + \text{time}_{\text{closure checking}} \quad (2)$$

Performing the closure check in each computer is not efficient because it may involve a lot of itemsets. However, it takes a lot of time to do closure checking after obtaining all itemsets from the three computers. Our method can generate closed itemsets directly, and it does not require time $_{\text{closure checking}}$. The time saved with CIMNC can be expressed in formula (3):

$$\text{Saved time} = \text{run time}_{\text{non-CIMNC}}(A) + \text{time}_{\text{closure checking}} - \text{run time}_{\text{CIMNC}}(A) \quad (3)$$

4. Experimental results

A. Environment and Datasets

Our method uses the horizontal database and FP-tree to find closed itemsets. CLOSET+ and FPclose are two popular algorithms that are appropriate to compare with our method. We know the purpose of using FP-array in FPclose is to accelerate the generation of FP-tree. Since arrays take up a large amount of space, FPclose is not a good candidate for comparison. Therefore, CLOSET+ is the most suitable one.

The CIMNC algorithm was implemented with Java programming language. In order to compare it with the CLOSET+ algorithm, we also implemented CLOSET+ in Java. Our experiments were performed on a personal computer of Intel Pentium 4 640 series, 3.2GHZ processor and DDR2 533MHz 2GB main memory. The experiments' datasets are produced by the IBM dataset generator [17]. We list the parameters in Table 2. We can generate datasets by selecting these parameters to evaluate the performance of our algorithm.

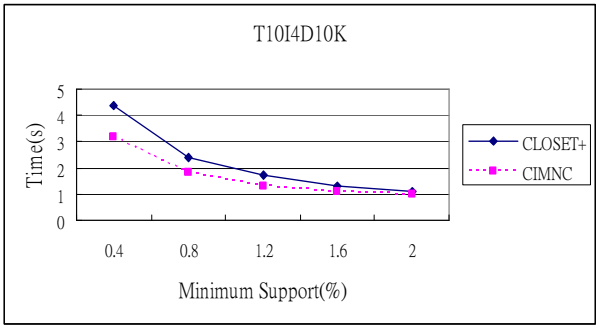
Table 1 Parameters used in the IBM dataset generator.

D	Number of transactions
T	Average size of transactions
I	Average size of maximal potentially-large itemsets
L	Number of potentially-large itemsets
N	Number of items

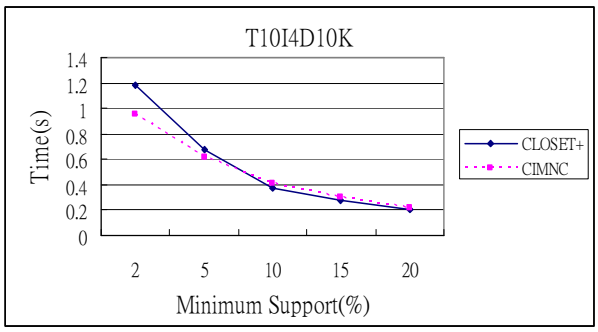
In order to show the difference between CLOSET+ and our method, we generate the databases by setting the number of items N = 50, the number of potentially-large itemsets L = 1000, and use the settings of the parameters T10I4D10K to generate the test data. We set the average length of the transaction T = 10, the average size of maximal potentially-large itemsets I = 4 and the number of transactions D =10000.

B. Experiment Result and Discussion

The dataset T10I4D10K was used to run two algorithms CIMNC and CLOSET+. The results of setting the average sizes of the potentially large itemset as 4 with the minimum supports from 0.4% to 2% are shown in Fig. 9(a) while the results for the minimum supports from 2% to 20% are shown in Fig. 9(b).



(a)



(b)

Fig. 9 The experiment results on T10I4D10K. (a) with smaller min_sup. (b) with larger min_sup.

In Fig. 9(a), one can see the execution time of our method is better than CLOSET+. When the minimum support is lower, the performance of our method gets better than CLOSET+. Because the number of closed itemset candidates is very big when the minimum support is very low, CLOSET+ needs more time to compare a lot of itemsets for closure checking. Since CIMNC does not need to compare many itemsets, it can get better performance in lower minimum supports. When the support increases, the number of candidates becomes smaller. Then the run time of CLOSET+ approaches the run time of CIMNC.

In Fig. 9(b), one can see the performance of CLOSET+ is better than CIMNC when the minimum support is near 10%. Because the number of candidates is very small and most itemsets are 1-itemset and 2-itemset, the time for closure checking becomes much less. Since CIMNC still needs to generate the records, it would take a little more time than CLOSET+. We know the time of generating records is based on the size of tree. Since the tree becomes smaller as the minimum support is larger than 10%, the performance of CIMNC is very close to CLOSET+.

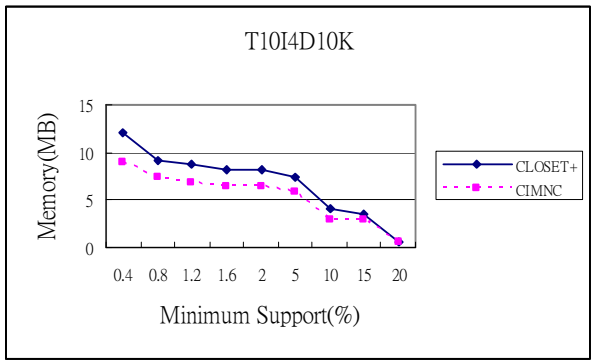


Fig. 10 Memory consumption on T10I4D10K.

Fig. 10 shows the memory consumption on T10I4D10K. The performance of CIMNC is 1.3 times better than CLOSET+ on average. Because CIMNC does not need to store candidates, it takes less memory space. Although noting records take up some memory space, it is still less than the space of storing all candidates. Besides, the storage used to store candidates may be larger than the original FP-tree in the worst case. Since not every record needs to note items, CIMNC takes less space than CLOSET+. In this figure, the line drops fast when the minimum support is from 5 to 10 and 15 to 20. This is because that the frequent items are reduced quickly, so the tree becomes very small. When the minimum support reaches 20, the required memory space drops to near 0. The reason is that the number of frequent items becomes very small and only frequent 1-itemsets can be found, such that less memory space is needed for the FP-tree.

5. Conclusions and future work

In this paper, an efficient approach called CIMNC is proposed to mine frequent closed itemsets without the need of closure checking. CIMNC has several advantages over other approaches. First, it is not necessary to do closure checking. However, it still can achieve the effect of subset pruning and reducing the number of conditional FP-trees. Because CIMNC doesn't generate any candidates, it is not necessary to store itemsets in the memory. It can output frequent closed itemsets directly and is suitable for parallel mining.

After implementing the CIMNC algorithm, we have studied its performance to compare with CLOSET+ for large databases. The results of performance study show that our method outperforms this well-known approach.

Further improvements on the record mode are in the list of our future work. A simple format will be used to note the records to avoid keeping a long string in the record, and check if they have the same items quickly.

Acknowledgement

This work was supported partially by the National Science Council, Taiwan, under grant numbers NSC95-2221-E-035-068-MY3 and NSC97-3114-E-007 -001.

References

[1] R. Agrawal, and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. of the 20th International Conference on Very Large Data Bases (VLDB)*, 1994, pp. 487-499.

[2] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2000, pp. 1-12.

[3] D. Lin and Z. M. Kedem, "Pincer-search: an Efficient Algorithm for Discovering the Maximum Frequent Set," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 14, No. 3, 2002, pp. 553-566.

[4] G. Grahne, and J. Zhu, "Fast Algorithms for Frequent Itemset Mining Using FP-Trees," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, No. 10, 2005, pp. 1347-1362.

[5] J. Pei, J. Han, and R. Mao, "Closet: An Efficient Algorithm for Mining Frequent Closed Itemsets," *Proc. of the 5th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2000, pp. 11-20.

[6] J. Wang, J. Han, and J. Pei, "CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets," *Proc. of the 9th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 236-245.

[7] Mohanmmed J. Zaki, and C.-J. Hsiao, "Efficient Algorithms for Mining Closed Itemsets and Their Lattice Structure," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, No. 4, 2005, pp. 462-478.

[8] C. Lucchese, S. Orlando, and R. Perego, "Fast and Memory Efficient Mining of Frequent Closed Itemsets," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 18, No. 1, 2006, pp. 21-36.

[9] C. Liu, H. Lu, X. Yu, W. Wang, and X. Xiao, "AFOPT: An Efficient Implementation of Pattern Growth Approach," *Proc. of IEEE ICDM'03 Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, 2003, pp. 1-10.

[10] L. Ning, N. Wu, and J. Zhang, "A New Technique for Fast Frequent Closed Itemsets Mining," *IEEE International Conference on Systems, Man and Cybernetics*, Vol. 4, 2005, pp. 3640-3647.

[11] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu, "MAFIA: A Maximal Frequent Itemset Algorithm," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, No. 11, 2005, pp. 1490-1504.

[12] P.-N. Tan, M. Steinbach and V. Kumar, *Introduction to Data Mining*. Addison Wesley, 2006.

[13] U. Fayyad, G. PiatetskyShapiro and P. Smyth, "The KDD Process for Extracting Useful Knowledge from Volumes of Data," *Communications of the ACM*, Vol. 39, No. 11, 1996, pp. 27-34.

[14] D.-Y. Chiu, Y.-H. Wu, and A.L.P. Chen , "An Efficient Algorithm for Mining Frequent Sequences by a New Strategy without Support Counting," *Proc. of IEEE Conference on Data Engineering (ICDE'04)*, 2004, pp. 375-386.

[15] M. Song, and S. Rajasekaran, "A Transaction Mapping Algorithm for Frequent Itemsets Mining," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 18, No. 4, 2006, pp. 472-481.

[16] M. Seno, and G. Karypis, "LPMiner: An Algorithm for Finding Frequent Itemsets Using Length-Decreasing Support Constraint," *Proc. of the 2001 IEEE International Conference on Data Mining (ICDM '01)*, 2001, pp. 505-512.

[17] IBM Almaden Research Center, "Synthetic Data Generation Code for Associations and Sequential Patterns," URL:http://www.almaden.ibm.com/software/quest/, 2006.

Author biographies

Chih-Hsien Lee received his M.S. degree from the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan. His research interests include data mining and software engineering.

Kuo-Cheng Yin received his M.S. degree from the Department of Information Engineering and Computer Science at Feng Chia University, Taiwan. He is currently a Ph.D. candidate there and an instructor in the Department of Information Management at Jen-Teh Junior College, Taiwan. His research interests include data mining and image processing.

Don-Lin Yang received his B.E. degree in Computer Science from Feng Chia University, Taiwan, in 1973, an M.S. degree in Applied Science from the College of William and Mary in 1979, and a Ph.D. degree in Computer Science from the University of Virginia in 1985. He worked at IBM Santa Teresa Laboratory from 1985 to 1987 and at AT&T Bell Laboratories from 1987 to 1991. Since then, he joined the faculty of Feng Chia University and is currently a professor in the Department of Information Engineering and Computer Science. His research interests include data mining, software engineering, and computer networks.

Jungpin Wu received the B.S. degree in Applied Mathematics from Tatung University, Taiwan, in 1988, the M.S. degree in Statistics from the Graduate Institute of Statistics of National Central University in 1993, and the Ph.D. degree in Statistics from the North Carolina State University in 1998. He was a postdoctoral staff at Academia Sinica from 1998 to 1999. Since then, he joined the faculty of Feng Chia University, where he was an Assistant Professor in the Department of Statistics from 1999 to 2004. Dr. Wu is currently an Associate Professor in the Department of Public Finance. His research interests include spatial statistics, generalized estimating equations, empirical process approach, and data mining.