Mining Maximal Frequent ItemSets Using Combined FP-Tree

Yuejin Yan†, Zhoujun Li, Tao Wang, Yuexin Chen, and Huowang Chen

School of Computer Science, National University of Defense Technology, Changsha 410073, China

† Corresponding author: Phn 86-731-4532956
yanyuejin2003@hotmail.com
http://www.nudt.edu.cn

Abstract. Maximal frequent itemsets mining is one of the most fundamental problems in data mining. In this paper, we present CfpMfi, a new depth-first search algorithm based on CFP-tree for mining MFI. Based on the new data structure CFP-tree, which is a combination of FP-tree and MFI-tree, CfpMfi takes a variety pruning techniques and a novel item ordering policy to reduce the search space efficiently. Experimental comparison with previous work reveals that, on dense datasets, CfpMfi prunes the search space efficiently and is better than other MFI Mining algorithms on dense datasets, and uses less main memory than similar algorithm.

1 Introduction

Since the frequent itemsets mining problem (FIM) was first addressed [1], frequent itemsets mining in large database has become an important problem. And the number of frequent itemsets increases exponentially with the increasing of frequent itemsets' length. So the large length of frequent itemset leads to no feasible of FI mining. Furthermore, since frequent itemsets are upward closed, it is sufficient to discover only all maximal frequent itemsets. As a result, researchers now turn to find MFI (maximal frequent itemsets) [4,5,6,7,9,10,13]. A frequent itemset is called maximal if it has no frequent superset. Given a set of MFI, it is easy to analyze some interesting properties of the database, such as the longest pattern, the overlap of the MFI, etc. There are also applications where the MFI is adequate, for example, the combinatorial pattern discovery in biological applications [3].

This paper introduces a new algorithm for mining MFI. We use a novel combined FP-tree in the process of mining, where the right represents sub database containing all relevant frequency information, and the left stores information of discovered MFI that is useful for *superset frequency* pruning. Based on the combined FP-tree, our algorithm takes a novel item ordering policy, and integrates a variety of old and new prune strategies. It also uses a simple but fast superset checking method along with some other optimizations.

G.I. Webb and Xinghuo Yu (Eds.): AI 2004, LNAI 3339, pp. 475–487, 2004.

The organization of the paper is as follows: Section 2 describes the basic concepts and the pruning techniques for mining MFI. Section 3 gives the MFI mining algorithm, CfpMfi, which does the MFI mining based on the combined FP-tree. In section 4, we compare our algorithm with some previous ones. Finally, in section 5, we draw the conclusions.

2 Preliminaries and Related Works

This section will formally describe the MFI mining problem and the set enumeration tree that represents the search space. Also the related works will be introduced in this section.

2.1 Problem Statement

The problem of mining maximal frequent itemsets is formally stated by definitions 1-4 and lemmas 1-2.

Let $I = \{i_1, i_2, \ldots, i_m\}$be a set of m distinct items. Let D denote a database of transactions, where each transaction contains a set of items.

Definition 1:(Itemset)
A set $X \subseteq I$ is called an *itemset*. An itemset with k items is called a k-itemset.

Definition 2:(Itemset's Support)
The support of an itemset X, denoted as $\delta(X)$, is defined as the number of transactions in which X occurs as a subset.

Definition 3:(Frequent Itemset)
For a given D, Let ξ be the threshold minimum support value specified by user. If $\delta(X) \geq \xi$, itemset X is called a frequent itemset.

Definition 4:(Maximal Frequent Itemset)
If $\delta(X) \geq \xi$ and for any $Y \supseteq X$, we have $\delta(Y) < \xi$, then X is called a maximal frequent itemset.

According to definitions 3-4, the following lemmas hold.

Lemma 1: A proper subset of any frequent itemset is not a maximal frequent itemset.

Lemma 2: A subset of any frequent itemset is a frequent itemset, a superset of any infrequent itemset is not a frequent itemset.

Given a transactional database D, supposed I is an itemset of it, then any combination of the items in I would be frequent and all these combinations compose the search space, which can be represented by a set enumeration tree [5]. The root of the tree represents the empty itemset, and the nodes at level k contain all of the k-itemsets. The itemset associated with each node, n, will be referred as the node's *head* (n). A complete set enumeration tree of $\{a,b,c,d,e,f\}$ in given order of a,b,c,d,e is shown in figure 1.

The possible extensions of the itemset is denoted as *ctail*(*n*), which is a set of items after the last item of *head*(*n*). The frequent extensions denoted as *ftail*(*n*) is a set of items that can be appended to *head*(*n*) to build the longer frequent itemsets. In depth-first traversal of the tree, *ftail*(*n*) contains just the frequent extensions of *n*. The itemset associated with each children node of node *n* is set up by appending one of *ftail*(*n*) to *head* (*n*). If a children node of *n* is formed by appending item *i* to *head*(*n*), the children node is called a child of *n* at *i*.

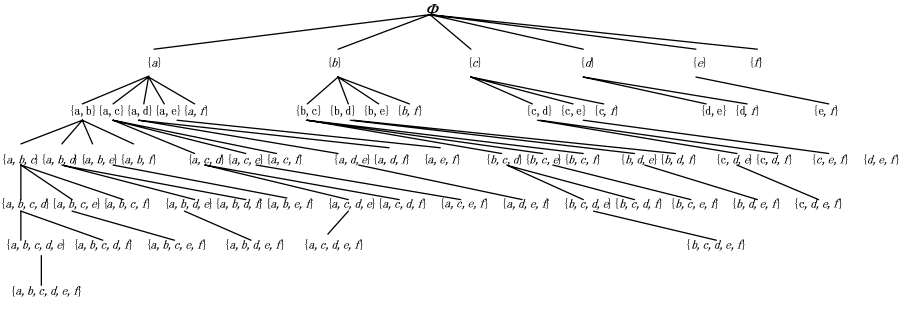

Fig. 1. A Complete Set Enumeration tree of {*a*,*b*,*c*,*d*,*e*,*f*}

The process of mining is a searching process of the enumeration tree, and it is important to prune the tree efficiently while searching. According to the lemmas 1-2, we introduce theorem 1-3 on pruning.

Theorem 1:(Subset Infrequency Pruning)
If $head(n) \cup \{i\}$ is infrequent($i \in ctail(n)$), the node that is the child of *n* at *i* can be pruned from the enumeration tree.

Theorem 2:(Superset Frequency Pruning)
If $head(n) \cup ftail(n)$ is frequent, all the children nodes of *n* can be pruned from the enumeration tree, and so do all the offspring nodes of *n*.

Superset frequency pruning is also called *looksahead* pruning in MaxMiner. Here we called it *looksahead* pruning with frequent extensions. If $head(n) \cup ctail(n)$ is frequent, there is a *looksahead* pruning with candidate extensions.

Theorem 3:(*PEP*)
If $\delta(head(n) \cup \{i\}) = \delta(head(n))$, the node that is the child of *n* at *i* can be pruned from the enumeration tree.

Proof: As each transaction contains *head*(*n*) will also contain item *i*, we can say that any frequent itemset *Z* containing *head*(*n*) but not *i*, has the frequent superset $Z \cup \{i\}$, and *i* can be moved from *ftail*(*n*) to *head*(*n*). *PEP* (Parent Equivalence Pruning) can also be seen as that the children node of *n* at *i* is been composed into *n* and does not need to been searched any more.

2.2 Related Work

Based on the set enumeration tree, we can describe the most recent approaches to MFI mining problem.

The MaxMiner [5] employs a breadth-first traversal policy for the searching. To reduce the search space, it introduced a new pruning technique named as *lookaheads* pruning. To increase the effectiveness of *lookaheads* pruning, MaxMiner dynamically reorders the children nodes, which was used in most of the MFI algorithms after that [4,6,7,9,10].

Mafia [7] is a depth-first algorithm. It uses a vector bitmap representation as in [6], where the count of an itemset is based on the column in the bitmap. All the three pruning methods mentioned in section 2.1 are used in Mafia.

Both MaxMiner and Mafia mine a superset of the MFIs, and require a post-pruning to eliminate non-maximal frequent itemsets. GenMax [9] integrates the pruning with mining to finds the exact MFIs by using two strategies. First, just like that transaction database is projected on current node, the discovered MFI set can also be projected on the node and thus yields fast superset checking; Second, GenMax uses *Diffset* propagation to do fast support computation.

AFOPT [3] uses a data structure called AFOPT tree in which items are ascending frequency ordered to store the transactions in conditional databases with top-town tree traversal strategy. It employs MFI projection generated by pseudo projection technique to test whether a frequent itemset is a subset of one of the discovered MFIs.FPMax* is an extension of the FP-growth method, for MFIs mining only. It uses a FP-tree to store the transaction projection of the original database for each node in the tree. In order to test whether a frequent itemset is the subset of any discovered MFI in *lookaheads* pruning, another tree structure, named MFI-tree, is utilized to keep the track of all discovered MFI, which makes effective superset checking. FPMax* uses an array for each node to store the counts of all 2-itemsets that is a subset of the frequent extensions itemset, this makes the algorithm scan each FP-tree only once for each recursive call emanating from it. The experiment results in FIMI'03 [10] shows that FPMax* has the best performance then for almost all the tested database. FIMfi [13] is also an algorithm base on FP-tree and MFI-tree, and it employs a new item ordering policy and a new method to do superset checking to improve performance.

3 Mining Maximal Frequent Itemsets by CfpMfi

In this section, we discuss algorithm CfpMfi in details.

3.1 Combined FP-Tree

For each node to be searched in the enumeration tree, CfpMfi builds a CFP-tree (combined FP-tree) that is a combination of FP-tree and MFI-tree. The right of the CFP-tree is a FP-tree that stores all relevant frequency information in database and the left of the CFP-tree is a MFI-tree which is used to keep the track of discovered

MFIs for current node. And in the left of the CFP-tree, there is not the field *level* for each node as in MFI-tree of FPMax*. There is also a header for each CFP-tree, and each head entry of CFP-tree has five fields: *item-name*, *right-node-link*, *left-node-link*, *maximal-level* and *maximal-type*. Fields maximal-level and maximal-type are used for subset testing and what they record is defined by definition 5. For an item i in header, we can define the maximal subset of i in CFP-tree as follows.

Definition 5: (i's Maximal Subset in CFP-Tree)
Let S_1 be the itemset represented by the maximal conditional pattern base of i in the left of CFP-tree, for all the conditional pattern bases of i in which the last item's count is more than the threshold ξ in the right of the CFP-tree, and let S_2 be the itemset represented by the maximal base. If $|S_2| > |S_1|$, S_2 is called the right maximal subset of i in CFP-tree, otherwise S_1 is called the left maximal subset of i in CFP-tree.

The field maximal-level at each header entry represents the items' number in S_1 or S_2, and the field maximal-type records the type of the maximal subset of the item in CFP-tree. Figure 2 shows an example of CFP-tree of root when considering extending to the child node $\{e\}$ of root. In the header of the tree, the "T" means the corresponding item that has one left maximal subset, and the "F" means the corresponding item has one right maximal subset.

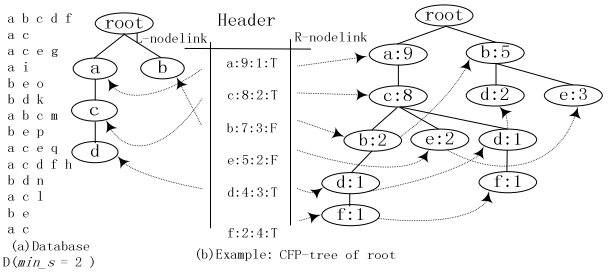

Fig. 2. Example of CFP-tree

According to definition 5, the theorem is given as follows.

Theorem 4:
For a node n, let i be the last item in $head(n)$ and S be its maximal subset in n's CFP-tree, then $head(n) \cup S$ is frequent. Furthermore, if S is the right maximal subset of i in the CFP-tree, $head(n) \cup S$ is a new maximal frequent itemset.

The building process of the CFP-tree of node n in search space is as follows: (1) The header is built after having found all the items that is frequent after combined with $head(n)$. (2) The right part of the CFP-tree associated with the parent node of n is scanned once, the items that do not occur in the header are removed from each conditional pattern base, then the base is inserted into the right of current CFP-tree after being reordered according the order of the header, the extra work needed to be done in the insertion is that the two maximal fields in header entries of the items in

the base will be updated after the insertion. (3) Building of the left of CFP-tree is similar to the building of the right in the second step. The first step and the second step are described in procedure 1, and the third step is shown in procedure 2. Figure 3(a) gives the examples of the building of CFP-tree.

Procedure 1: RightBuild
Input : *n* : current node
***i* : the item needed to be extended currently**
fretail*: the frequent extensions itemset for the child of *n* at *i
***pepset*: a null itemset to store *PEP* items**
Output : the new CFP-tree with initialized header and left

(1) Sort the items in *fretail* in the increasing order of support
(2) $pepset = \{i \mid \delta(head(n) \cup \{i\}) = \delta(head(n))$ and i is a frequent extension of head(n)$\}$
(3) Build a head entry for each item in *fretail* but not in *pepset* for the new CFP-tree
(4) For each itemset s according to a conditional pattern base of i in the right of *n.cfptree*
(5) Delete the items in s but not in *fretail*
(6) Sort the remaining items in s according to the order of *fretail*
(7) Insert the sorted itemset into the right of the new CFP-tree
(8) Return the new CFP-tree

Procedure 2: LeftBuild
Input : *n* : current node
***i* : the item needed to be extended currently**
n'*: the child of *n* at *i
Output: the CFP-tree with the left initialized

(1) For each itemset s according to a conditional pattern base of i in the left of *n.cfptree*
(2) Delete the items in s but not in *n'.cfptree.header*
(3) Sort the remaining items in s according to the order of *n'.cfptree.header*
(4) Insert the sorted itemsets into the left of *n'.cfptree*
(5) Return *n'.cfptree*

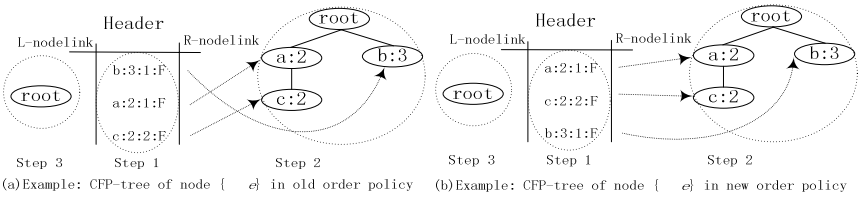

Fig. 3. Examples of constructing CFP-tree

If a new MFI is found, it is used to update the CFP-trees' left of each node within the path from root to current node in search space tree. For example, after considering

the node {*f*}, {*f*,*d*,*c*,*a*}-{*f*} is inserted to the left of CFP-trees of root, and after considering the node {*d*}, {*d*,*b*}-{*d*} is inserted to the left of CFP-trees of root, the left of root's CFP-tree is null when it is initialized, after the two insertion, the CFP-tree is shown as in Figure 2.

3.2 CfpMfi

The pseudo-code of CfpMfi is shown in Figure 6. In each call procedure, each newly found MFI maybe be used in superset checking for ancestor nodes of the current node, so we use a global parameter called *cfpTreeArray* to access the Cfp-trees of ancestor nodes. And when the top call (CfpMfi(*root*,ϕ)) is over, all the MFIs to be mined are stored in the left of *root*'s Cfp-tree in the search space tree.

Lines (4), (9) and (20) are the simple superset checking that will be described in detail in section 3.3. When *x* is the end item of the header, there is no need to do the checking, for the checking has already been done by the procedure calling current one in line (9) and/or line (20). Lines from (7) to (8) use the optimization array technique introduced in section 3.5. The *PEP* technique is used by call procedure rightbuild in line (18). Lines (5)-(6) and lines (21)-(24) are two *lookaheads* prunings with candidate extensions. The *lookaheads* pruning with frequent extensions is done in lines from (14) to (20). When the condition in lines (11), (15) and (22) is true, all the children nodes of *n*' are pruned and *ftail*(*n'*) or ctail(*n'*) need not to be inserted into the left of *n.CFP-tree* any more. The novel item ordering policy will be introduced in section 3.4 and is used in procedure 1 in line (1). Line (18) builds the header and the right of *n*'.*CFP-tree*. The *return* statements in line (5), (6), (12), (16) and (23) mean that all the children nodes after *n'* of *n* are pruned there. And the *continue* statements in line (13), (17) and (24) tell us that node *n'* will be pruned, then we can go to consider the next child of *n*. The left of *n*'.*CFP-tree* is built by call procedure leftbuild in line (25). After the constructing of the whole *n'.CFP-tree* and the updating of *cfpTreeArray*, CfpMfi will be called recursively with the new node *n'* and the new *cfpTreeArray*.

Note CfpMfi doesn't employ single path trimming used in FPMax* and AFOPT. If, by having constructed the right of *n'.CFP-tree*, we find out that the right of *n'.CFP-tree* has a single path, the superset checking in line (20) will return *true*, there will be a *lookaheads* pruning instead of a single path trimming.

Procedure: CfpMfi
Input:
n: a node in search space tree ;
cfpTreeArray: CFp-trees of all ancestor nodes of *n* in the path of search space tree from root to *n*.

```
( 1)For each item x from end to beginning in header of n.CFP-tree
( 2)  h'=h ∪{x}                                   //h' identifies n'
( 3)  if x is not the end item of the header
( 4)     if | ctail(n') | == n.CFP-tree.header.x.maximal-level
( 5)        if  n.CFP-tree.header.x.maximal-type == "T"          return
( 6)        else insert h'∪ ctail(n') into cfpTreeArray          return
```

(7) if *n.array* is null count *ftail*(*n*') using the *n.array*
(8) else count *ftail*(*n*') by a scan of the right of *n.CFP_tree*
(9) if | *ftail*(*n*') | == *n.CFP-tree.header.x.maximal-level*
(10) if *n.CFP-tree.header.x.maximal-type* == "T"
(11) if the number of items before *x* in the *n.CFP-tree.header* is | *ftail*(*n*')|
(12) return
(13) else continue
(14) insert *h*'∪ *ftail*(*n*') into *cfpTreeArray*
(15) if the number of items before *x* in the *n.CFP-tree.header* is | *ftail*(*n*')|
(16) return
(17) else insert *ftail*(*n*') into *n.CFP-tree* continue
(18) *pepset* =ϕ; *n*'.*CFP-tree* = rightbuild(*n*,*x*, *ftail*(*n*'),*pepset*)
(19) *h*' = *h*'∪ *pepset*
(20) if | *n*'.*header* | == *n*'.*CFP-tree.header.lastitem.maximal-level*
(21) insert *h*' ∪ *ftail*(*n*') into *cfpTreeArray*
(22) if the number of items before *x* in the *n.CFP-tree.header* is | *ftail*(*n*')|
(23) return
(24) else insert *ftail*(*n*') into *n.CFP-tree* continue
(25) *n*'.*CFP-tree* = leftbuild(*n*, *n*',*x*)
(26) *cfpTreeArray* = *cfpTreeArray* ∪ {*n.CFP-tree*}
(27) call CfpMfi(n' , *cfpTreeArray*)

3.3 Implementation of Superset Checking

According to Theorem 2, if *head*(*n*) ∪ *ftail*(*n*) or *head*(*n*) ∪ *ctail*(*n*) is frequent, there will be a *lookaheads* pruning, There are two existing methods for determining whether the itemset *head*(*n*) ∪ *ftail*(*n*) or *head*(*n*) ∪ *ctail*(*n*) is frequent. The first one is to count the support of *head*(*n*) ∪ *ftail*(*n*) directly, and this method is normally used in an bread-first algorithms such as in MaxMiner. The second one is to check whether a superset of *head*(*n*) ∪ *ftail*(*n*) has already been in the discovered MFIs, which is used by the depth-first MFI algorithms commonly [4,7,9,10]. When implementing the superset checking, GenMax uses LMFI to store all the relevant MFIs, and the mapping item by item for *ftail*(*n*) in the LMFI is needed; In MFI-tree, FPMax* needs only map *ftail*(*n*) item by item in all conditional pattern bases of *head*(*n*). The simple but fast superset checking for *head*(*n*) ∪ *ctail*(*n*) or *head*(*n*) ∪ *ftail*(*n*) is firstly introduced in [13]. In CfpMfi, the implementation of superset checking is based on the theorems as follows:

Theorem 5: Let *n*' be the child of *n* at *i*, if the size of *i*'s maximal subset in CFP-tree of *n* is equal to the size of *ftail*(*n*') or *ctail*(*n*'), then *head*(*n*')∪*ftail*(*n*') or *head*(*n*') ∪ *ctail*(*n*') is frequent.

Proof: Let *S* be *i*'s maximal subset in CFP-tree of *n*. (1): If *S* is in the left of the CFP-tree, *head*(*n*') ∪ *S* is an subset of some discovered MFIs, then *head*(*n*) ∪ *S* is frequent; According to theorem 4, when *S* is in the right of the CFP-tree, *head*(*n*) ∪ *S* is frequent too. (2): According to the definition of *frq_tail* and *ctail*, we have *ftail*(*n*') =

{x| x is a item that is bigger than i in header of n' CFP-tree and $head(n') \cup \{x\}$ is frquent} and $ftail(n') \subseteq ctail(n')$, then $S \subseteq ftail(n') \subseteq ctail(n')$. According to (1) and (2), when the assumption in theorem 5 is right, $S = ftail(n')$ or $S = ctail(n')$ can be hold. Hence, we obtain the theorem.

According to theorem 5, in Cfp-tree, the field *maximal-level* of each *header* entry records the size of the maximal subset, so the superset checking becomes very simple and only needs to check the field with the size of $ftail(n')$ or $ctail(n')$. Note that superset checking is a frequent operation in the process of mining MFIs. It is because that each new MFI needs to be checked before being added into the discovered MFIs. Then the implementation of superset checking can improve the performance of *lookaheads* pruning efficiently. Furthermore, when the superset checking returns true and the maximal subset is in the right of CFP-tree, it is no need to construct the CFP-trees of n' and n'' offspring nodes, there is a *lookaheads* pruning, but the itemset $head(n') \cup ftail(n')$ or $head(n') \cup ctail(n')$ is a new MFI, and, as described in procedure 3 in lines (6),(14),(17),(21) and (24), it is used to update the relevant CFP-trees.

3.4 Item Ordering Policy

Item ordering policy appears firstly in [5], and is used by almost all the following MFI algorithms for it can increase the effectiveness of superset frequency pruning. In general, this type of item ordering policy works better in *lookaheads* by scanning the database to count the support of $head(n) \cup ftail(n)$ in breath-first algorithms, such as in MaxMiner. All the recently proposed depth-first algorithms do the superset checking instead to implement the *lookaheads* pruning, for the counting support of $head(n) \cup ftail(n)$ costs high in depth-first policy.

FIMfi tries to find a maximal subset of $ftail(n)$, then let the subset in $ftail(n)$ ahead when ordering to gain maximal pruning at a node in question. In CfpMfi, the maximal subset S in definition 5 is the exact subset, and it can be used for this purpose without any extra cost. For example, when considering the node n identified by $\{e\}$, we know $ftail(n)=\{a,c,b\}$, $S=\{a,c\}$, then the sorted items in $ftail(n)$ is in sequence of a,c,b, the CFP-tree will be constructed as in figure 2(b), the old decreasing order of supports is b,a,c, the CFP-tree will be constructed as in figure 2(a). In the old order policy, the CFP-trees for nodes $\{e\}$, $\{e,a\}$, and $\{e,c\}$ will have to be build, but CFpMfi with the new order policy only need to build FP-trees for nodes $\{e\}$ and $\{e,b\}$. Furthermore, for the items in $ftail(n)$-S, we also sort them in the decreasing order of $sup(head(n) \cup \{x\})$ ($x \in ftail(n)$-S).

4 Experimental Evaluations

In the first Workshop on Frequent Itemset Mining Implementations (FIMI'03) [11], which took place at ICDM'03, there are several algorithms presented recently, which are good for mining MFI, such as FPMax*, AFOPT, Mafia and etc, and FIMfi is a newly presented algorithm in ER 2004 [14], we now present the performance comparisons between CfpMfi and them. All the experiments are conducted on 2.4 GHZ Pentium IV with 1024 MB of DDR memory running Microsoft Windows 2000 Pro-

fessional. The codes of other three algorithms were downloaded from [12] and all codes of the four algorithms were complied using Microsoft Visual C++ 6.0. Due to the lack of space, only the results for four real dense datasets are shown here. The datasets we used are also selected from all the 11 real datasets of FIMI'03[12], they are chess, Connect, Mushroom and Pumsb_star, and their data characteristics can be found in [11].

4.1 The Pruning Performance of CfpMfi

CfpMfi adopts the new item ordering policy, along with the *PEP* and *lookaheads* pruning with frequent extensions and candidate extensions, to prune the search space tree. Since FPMax* is nearly the best MFI mining algorithm in FIMI'03 and employs FP-tree and MFI-tree structures similar to CFP-tree, we select FPMax* as a benchmark algorithm to test the performance of CfpMfi in pruning. The comparison of the number of CFP-tree' rights built by CfpMfi and FP-trees created by FPMax* is shown in figure 4, in which the number of CFP-tree's rights in CfpMfi is less half of that of FP-trees in FPMax* on all the four dense datasets at all supports. And figure 5 reveals the comparison of the number of CFP-tree' lefts built by CfpMfi and MFI-trees created by FPMax*. And in figure 5, the number of CFP-tree's lefts in CfpMfi is about 30% -70% of that of MFI-trees in FPMax* on all the four dense datasets at all supports. Hence, it is not difficult to conclude from the two figures that by using the new item ordering policy and the pruning techniques, CfpMfi makes the pruning more efficient.

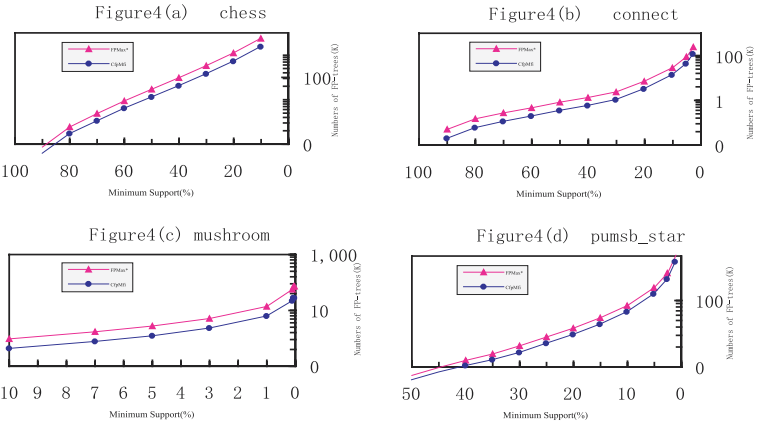

Fig. 4. Comparison of FP-trees' Number

4.2 Performance Comparisons

Figure 6 gives the results of comparison among the five algorithms on the selected dense datasets. For all supports on dense datasets Connect, Mushroom and Pumsb_star, CfpMfi has the best performance. CfpMfi runs around 40% -60% faster

than FPMax* on all of the dense datasets. AFOPT is the slowest algorithm on Chess, Mushroom and Pumsb_star and runs 2 to 10 times worse than CfpMfi on all of the

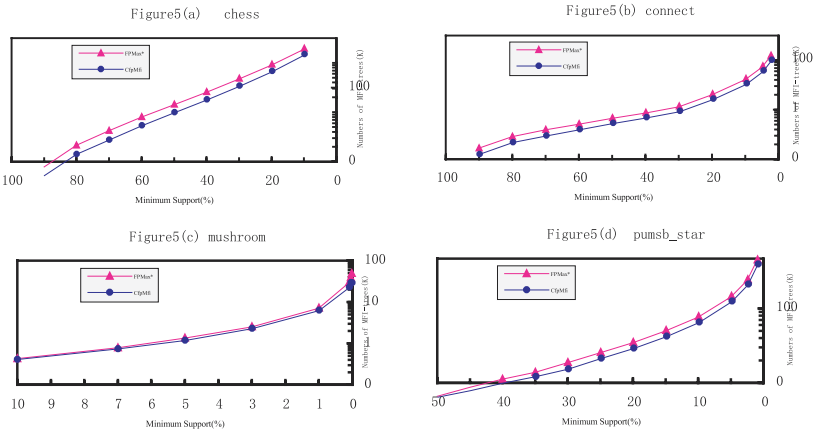

Fig. 5. Comparison of MFI-trees' Number

datasets across all supports. Mafia is the slowest algorithm on Connect, it runs 2 to 5 times slower than CfpMfi on Mushroom and Connect across all supports. On Pumsb_star, Mafia is outperformed by CfpMfi for all the supports though it outperforms FPMax* at lower supports, and on chess CfpMfi outperforms Mafia for the supports no less than 30% but Mafia outperforms FPMax* for the supports no less than 50%. CfpMfi outperforms FIMfi slightly until the lower supports where they cross over. In fact, with the lowest supports 10%, 2.5%, 0.025% and 1% for Chess, Connect, Mushroom and Pumsb_star, CfpMfi is %3, %25, %15 and 30% better than FIMfi respectively.

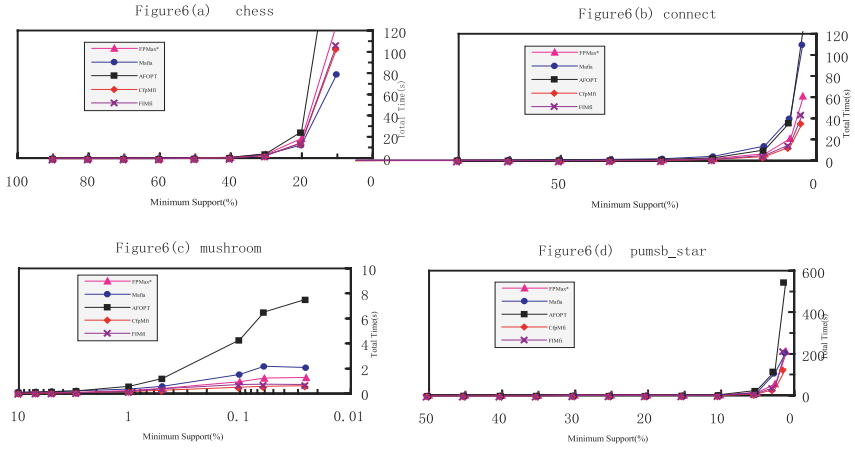

Fig. 6. Performance of Total Time

4.3 Maximal Main Memory Usage Comparisons

Figure 7 gives the results of maximal main memory used by the five algorithms on the selected dense datasets. From the figure, we can see that CfpMfi uses main memory more than FPMax* but less than FIMfi. The figure reveals that by using the compact data structure, CFP-tree, CfpMfi saves more main memory than FIMfi does.

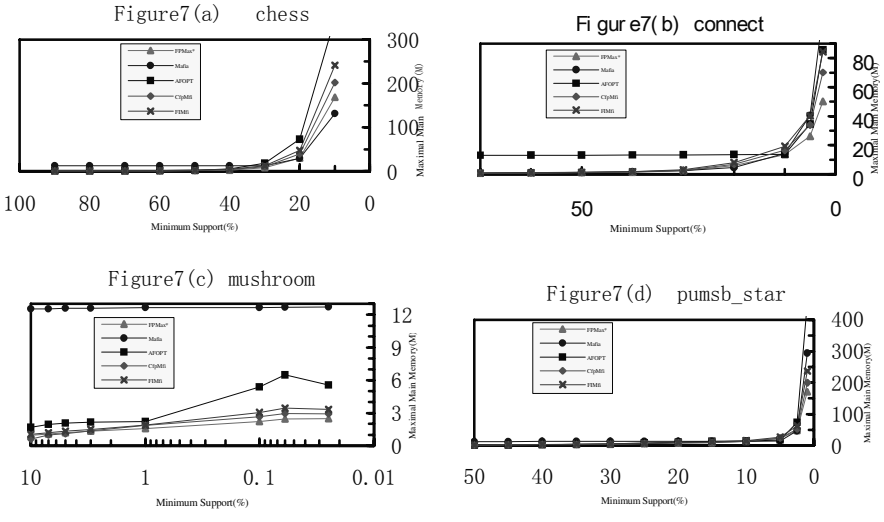

Fig. 7. Performance of Maximal Main Memory Usage

5 Conclusions

We presented CfpMfi, an algorithm for finding maximal frequent itemsets. Our experimental result demonstrates that, on dense datasets, FpMfi is more optimized for mining MFI and outperforms FPMax* by 40% averagely, and for lower supports it is about 10% better than FIMfi. The pruning performance and running time performance comparisons verify the efficiency of the novel ordering policy and the new method for superset checking that presented in [13], and the study of maximal main memory used indicates the compactness of CFP-tree structure. Thus it can be concluded that the new tree data structure, CFP-tree, along with the new ordering policy and some pruning techniques are well integrated into CfiMfi.

Acknowledgements

We would like to thank Guimei Liu for providing the code of AFOPT and Doug Burdick for providing the website of downloading the code of Mafia.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994.

[2] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation, Proc. 2000 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'00), Dallas, TX, May 2000.

[3] L. Rigoutsos and A. Floratos: Combinatorial pattern discovery in biological sequences: The Teiresias algorithm.Bioinformatics 14, 1 (1998), 55-67.

[4] Guimei Liu, Hongjun Lu, Jeffrey Xu Yu, Wei Wang and Xiangye Xiao. AFOPT: An Efficient Implementation of Pattern Growth Approach. In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA, November 19, 2003.

[5] Roberto Bayardo. Efficiently mining long patterns from databases. In ACM SIGMOD Conference, 1998.

[6] R. Agarwal, C. Aggarwal and V. Prasad. A tree projection algorithm for generation of frequent itemsets. Journal of Parallel and Distributed Computing, 2001.

[7] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A Performance Study of Mining Maximal Frequent Itemsets. In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations Melbourne, Florida, USA, November 19, 2003.

[8] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed association rule mining. TR 99-10, CS Dept., RPI, Oct. 1999.

[9] K. Gouda and M. J. Zaki. Efficiently Mining Maximal Frequent Itemsets. Proc. of the IEEE Int. Conference on Data Mining, San Jose, 2001.

[10] Gösta Grahne and Jianfei Zhu. Efficiently Using Prefix-trees in Mining Frequent Itemsets. In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA, November 19, 2003.

[11] Bart Goethals and M. J. Zaki. FIMI'03: Workshop on Frequent Itemset Mining Implementations. In Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA, November 19, 2003.

[12] Codes and datasets available at :http://fimi.cs.helsinki.fi/.

[13] Yuejin Yan, Zhoujunli and Huowang Chen. Fast Mining Maximal Frequent ItemSets Based on Fp-tree. In Proceedings of the 23rd International Conference on Conceptual Modeling (ER2004), ShangHai, China, November 8, 2004. (Accepted)

[14] ER 2004: http://www.cs.fudan.edu.cn/er2004/.