CT-PRO: A Bottom-Up Non Recursive Frequent Itemset Mining Algorithm Using Compressed FP-Tree Data Structure

Yudho Giri Sucahyo Raj P. Gopalan
Department of Computing, Curtin University of Technology
Kent St, Bentley
Western Australia 6102
{sucahyoy, raj}@cs.curtin.edu.au

Abstract

Frequent itemset mining (FIM) is an essential part of association rules mining. Its application for other data mining tasks has also been recognized. It has been an active research area and a large number of algorithms have been developed. In this paper, we propose another pattern growth algorithm which uses a more compact data structure named Compressed FP-Tree (CFP-Tree). The number of nodes in a CFP-Tree can be up to half less than in the corresponding FP-Tree. We also describe the implementation of CT-PRO which utilize the CFP-Tree for FIM. CT-PRO traverses the CFP-Tree bottom-up and generates the frequent itemsets following the pattern growth approach non-recursively. Experiments show that CT-PRO performs better than OpportuneProject, FP-Growth, and Apriori. A further experiment is conducted to determine the feasible performance range of CT-PRO and the result shows that CT-PRO has a larger performance range compared to others. CT-PRO also performs better compared to LCM and kDCI that are known as the two best algorithms in FIMI Repository 2003.

1. Introduction

Since its introduction in [1] the problem of efficiently generating frequent itemsets has been an active research area and a large number of algorithms have been developed for it; for surveys, see [2-4]. Frequent itemset mining (FIM) is an essential part of association rules mining (ARM). Since FIM is computationally expensive, the general performance of ARM is determined by it. The frequent itemset concept has also been extended for many other data mining tasks such as classification [5, 6], clustering [7], and sequential pattern discovery [8].

The data structures used play an important role in the performance of FIM algorithms. The various data structures used by FIM algorithms can be categorized as either array-based or tree-based. An example of a successful array-based algorithm for FIM is *H-Mine* [9]. It uses a data structure named *H-struct*, which is a combination of arrays and hyper-links. It was shown in [9] that *H-struct* works well for sparse datasets as *H-Mine* outperforms *FP-Growth* [10] on these datasets (note that both *H-Mine* and *FP-Growth* follows the same pattern growth method). However, the hyper-structure is not efficient on dense datasets and therefore *H-Mine* switches to *FP-Growth* for such datasets.

FP-Growth [10] shows good performance on dense datasets as it uses a compact data structure named *FP-Tree*. *FP-Tree* is a prefix tree with links between nodes containing the same item. A tree data structure is suitable for dense datasets since many transactions will share common prefixes so that the database could be compactly represented. However, for sparse datasets the tree will be bigger and bushier, and therefore its construction cost and traversal cost will be higher than array-based data structures.

The strengths of *H-Mine* and *FP-Growth* were combined in the recent pattern growth FIM algorithm named *OpportuneProject* (*OP*) [11]. *OP* is an adaptive algorithm that opportunistically chooses an array-based or a tree-based data structure depending on the sub-database characteristics.

In this paper, we describe our new data structure named *Compressed FP-Tree* (*CFP-Tree*) and also the implementation of our new FIM algorithm named *CT-PRO* that was first introduced in [12]. Here we report the compactness of *CFP-Tree* with *FP-Tree* at several support levels on the various datasets generated using the synthetic data generator [13]. The performance of *CT-PRO* is compared with *Apriori* [14, 15], *FP-Growth* [10], and *OP* [11].

Sample datasets such as real-world *BMS* datasets [3] or *UCI Machine Learning Repository* datasets [16] do not cover the full range of densities from sparse to dense. Some algorithms may work well for a certain dataset but may not be feasible when the dimensions of the database change (i.e. number of transactions, number of items, average number of items per transaction etc.). Therefore, a further study has been done in this paper, to show the feasible performance range of the algorithms. The more extensive testing of the algorithms is carried out using a set of databases with varying number of both transactions and average number of items per transaction. For each dataset, all the algorithms are tested on supports of 10% to 90% in increments of 10%. The experimental results are reported in detail.

To show how well *CT-PRO* compares with algorithms in FIMI Repository 2003 [17], two best algorithms from the last workshop, *LCM* [18] and *kDCI* [19], are selected for comparison. The result shows that *CT-PRO* outperforms these and therefore all others.

The structure of the rest of this paper is as follows: In Section 2, we introduce the *CFP-Tree* data structure and report the results of experiments in evaluating its compactness. In Section 3, we describe the *CT-PRO* algorithm with a running example. We discuss the complexity of *CT-PRO* algorithm in Section 4. The performance of the algorithm on various datasets is compared against other algorithms in Section 5. Section 6 contains conclusions of our study.

2. *Compressed FP-Tree* Data Structure

In this section, a new tree-based data structure, named *Compressed FP-Tree* (*CFP-Tree*), is introduced. It is a variant of *CT-Tree* data structure that we introduced in [20] with the following major differences: items are sorted in descending order of their frequency (instead of ascending order, as in *CT-Tree*) and there is a link to the next node with the same item node (while links are not present in *CT-Tree*). The *CFP-Tree* is defined as follows:

Definition 1 (*Compressed FP-Tree* or *CFP-Tree*). *A Compressed FP-Tree is a prefix tree with the following properties:*

1. *It consists of an ItemTable and a tree whose root represents the index of the item with the highest frequency and a set of subtrees as the children of the root.*
2. *The ItemTable contains all frequent items sorted in descending order by their frequency. Each entry in the ItemTable consists of four fields, (1) index, (2) item-id, (3) frequency of the item, and (4) a pointer pointing to the root of the subtree of each frequent item.*
3. *If $I = \{i_1, i_2, \ldots i_k\}$ is a set of frequent items in a transaction, after being mapped to their index-id, then the transaction will be inserted into the Compressed FP-Tree starting from the root of a subtree to which i_1 in the ItemTable points.*
4. *The root of the Compressed FP-Tree is the level 0 of the tree.*
5. *Each node in the Compressed FP-Tree consists of four fields: node-id, a pointer to the next sibling, a pointer to the next node with the same id, and a count array where each entry corresponds to the number of occurrences of an itemset. If $C = \{C_0, C_1, \ldots C_k\}$ is a set of counts in the count array attached to a node and the index of the array starts from zero, then C_i is the count of a transaction with an itemset along the path from the node at level i to the node where C_i is located.* ■

The following lemma provides the worst-case space complexity of a *CFP-Tree*.

Lemma 1. Let *n* be the number of frequent items in the database for a certain support threshold. The number of nodes of the *CFP-Tree* is bounded by 2^{n-1}, which is half of the maximum for a full prefix tree.

Rationale. *If $I_F = \{i_{F1}, \ldots i_{Fn}\}$ is a set of distinct items in a CFP-Tree where $i_{F1}, i_{F2,} \ldots i_{Fn}$ are lexicographically ordered. The maximum number of nodes under subtrees $i_{F1}, i_{F2,} \ldots i_{Fn}$ is $2^{n-1}, 2^{n-2} \ldots 2^0$ respectively. Since the CFP-Tree is actually the subtree i_{F1} then the maximum number of nodes of the CFP-Tree is 2^{n-1}.* ■

Compared to *FP-Tree*, *CFP-Tree* has some important differences, as follows:

1. *FP-Tree* stores the item id in the tree while, in *CFP-Tree,* item ids are mapped to an ascending sequence of integers that is actually the array index in *ItemTable*.
2. The *FP-Tree* is compressed by removing identical subtrees of a complete *FP-Tree* and succinctly storing the information from them in the remaining nodes. All subtrees of the root of the *FP-Tree* (except the leftmost branch) are collected together at the leftmost branch to form the *CFP-Tree*..
3. Each node in the *FP-Tree* (except the root) consists of three fields: *item-id*, *count* and *node-link*. *Count* registers the number of transactions represented by the portion of the path reaching this node. *Node-link* links to the next node with the same *item-id*. Each node in the *CFP-Tree* consists of three fields: *item-id*, *count array* and *node-link*. The *count array* contains counts for item subsets in the path from the root to that node. The index of the cells in the array corresponds to the level numbers of the nodes above.
4. *FP-Tree* has a *HeaderTable* consisting of two fields: *item-id* and a pointer to the first node in the *FP-Tree* carrying the nodes with the same *item-id*. *CFP-Tree* has an *ItemTable* consisting of four fields: *index*, *item-id*, *count of the item* and *a pointer to the root of the subtree of each item*. The root of each subtree has a link to the next node with the same-item-node. Both *HeaderTable* and *ItemTable* store only frequent items.

Figure 1 shows the *FP-Tree* and the *CFP-Tree* for a sample database. In this case, the *FP-Tree* is a complete tree for items 1-4. In this example, the number of nodes in the *FP-Tree* is twice that of the corresponding *CFP-Tree*. However, most datasets do not have such an extreme characteristic as in this example.

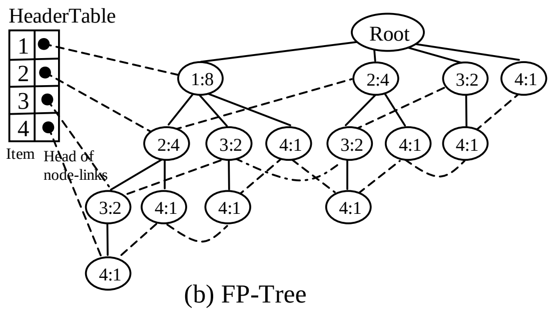
Figure 2 shows the compactness of *CFP-Tree* compared to *FP-Tree* on several synthetic datasets at various support levels (the characteristics of the datasets are explained later in Section 5.2). *CFP-Tree* has a smaller number of nodes compared to *FP-Tree* in all cases.

3. CT-PRO Algorithm

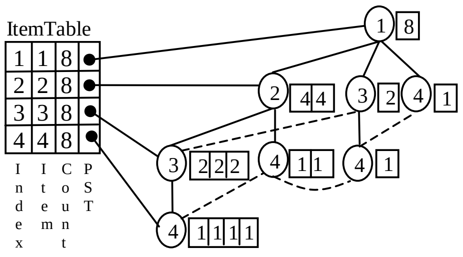
In this section, a new method that traverses the tree in a bottom-up strategy, and implemented non-recursively, is presented. The *CFP-Tree* data structure is used to compactly represent transactions in the memory. The algorithm is called *CT-PRO* and it has three steps in it: finding the frequent items, constructing the *CFP-Tree*, and mining. Algorithm 1 shows the first two steps in *CT-PRO*.

Tid	Items	Tid	Items	Tid	Items
1	1 2 3 4	6	2	11	1
2	2 4	7	1 4	12	2 3 4
3	1 3 4	8	1 2 3	13	1 2
4	3	9	3 4	14	1 2 4
5	2 3	10	4	15	1 3

(a) Sample Database



(b) FP-Tree



(c) CFP-Tree

Figure 1: FP-Tree and CFP-Tree

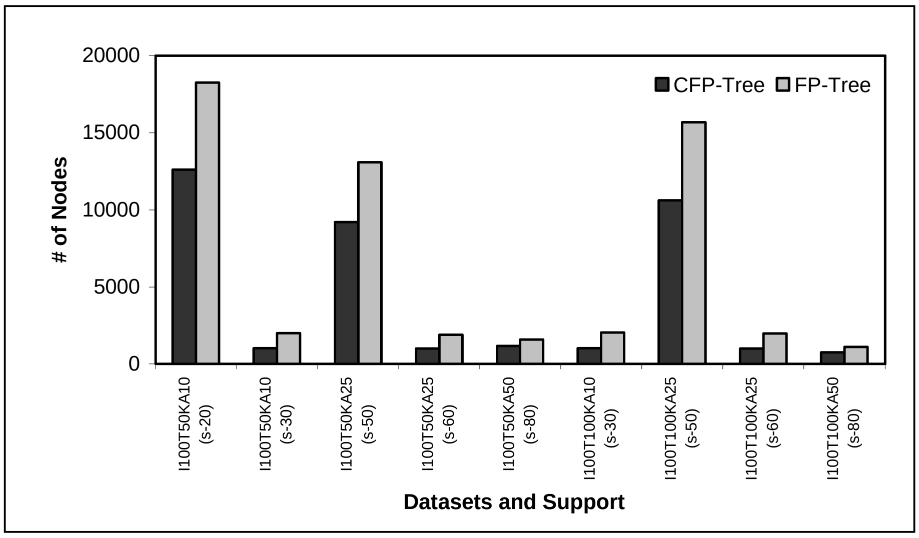


Figure 2: Compactness of CFP-Tree Compared to FP-Tree on Various Synthetic Datasets at Various Support Levels

Algorithm 1 CT-PRO Algorithm: Step 1 and Step 2

```
input   Database D, Support Threshold σ
output  CFP-Tree

1   begin
2     // Step 1: Identify frequent items
3     for each transaction t ∈ D
4       for each item i ∈ t
5         if i ∈ ItemTable
6           Increment count of i
7         else
8           Insert i into GlobalItemTable with count = 1
9         end if
10      end for
11    end for
12    Sort GlobalItemTable in
      frequency descending order
13    Assign an index for each frequent item in the
      GlobalItemTable
14    // Step 2: Construct CFP-Tree
15    Construct the left most branch of the tree
16    for each transaction t ∈ D
17      Initialize mappedTrans
18      for each frequent item i ∈ t
19        mappedTrans = mappedTrans ∪ GetIndex(i)
20      end for
21      Sort mappedTrans in ascending order of item ids
22      InsertToCFPTree(mappedTrans)
23    end for
24  end
25  Procedure InsertToCFPTree(mappedTrans)
26    firstItem := mappedTrans[1]
27    currNode := root of subtree pointed by
      ItemTable[firstItem]
28    for each subsequent item i ∈ mappedTrans
29      if currNode has child represent i
30        Increment count[firstItem-1] of the child node
31      else
32        Create child node and set its
          count[firstItem-1] to 1
33        Link the node to its respective node-link
34      end if
35    end for
36  end
```

Suppose the user wants to mine all frequent itemsets from the transaction database shown in Figure 3a with a support threshold of two transactions (or 40%). First, we need to identify frequent items by reading the database once (lines 3-11). The frequent items are stored in frequency descending order in the *GlobalItemTable* (line 12). In a second pass over the database, only frequent items are selected from each transaction (see Figure 3b), mapped to their index id in *GlobalItemTable* on-the-fly, sorted in ascending order of their index id (see Figure 3c) and inserted into the *CFP-Tree* (see Figure 3d). The pointer in *GlobalItemTable* also acts as the start of the links to other nodes with the same item ids (indicated by the dashed lines in Figure 3d). For illustration, at each node we also show the index of the array, the transaction represented at each index entry and its count. In the implementation of *CFP-Tree*, however, only the second column that represents the count is stored.

Tid	Items
1	3 4 5 6 7 9
2	1 3 4 5 13
3	1 2 4 5 7 11
4	1 3 4 8
5	1 3 4 10

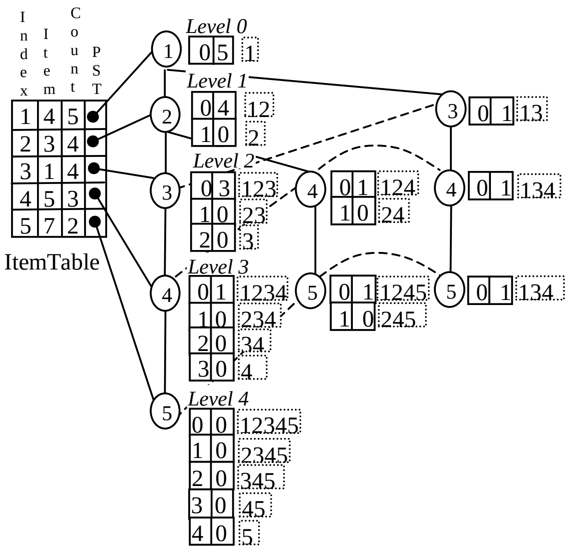
(a) Sample Database

Tid	Items
1	3 4 5 7
2	1 3 4 5
3	1 4 5 7
4	1 3 4
5	1 3 4

(b) Frequent Items

Tid	Items
1	1 2 4 5
2	1 2 3 4
3	1 3 4 5
4	1 2 3
5	1 2 3

(c) Mapped



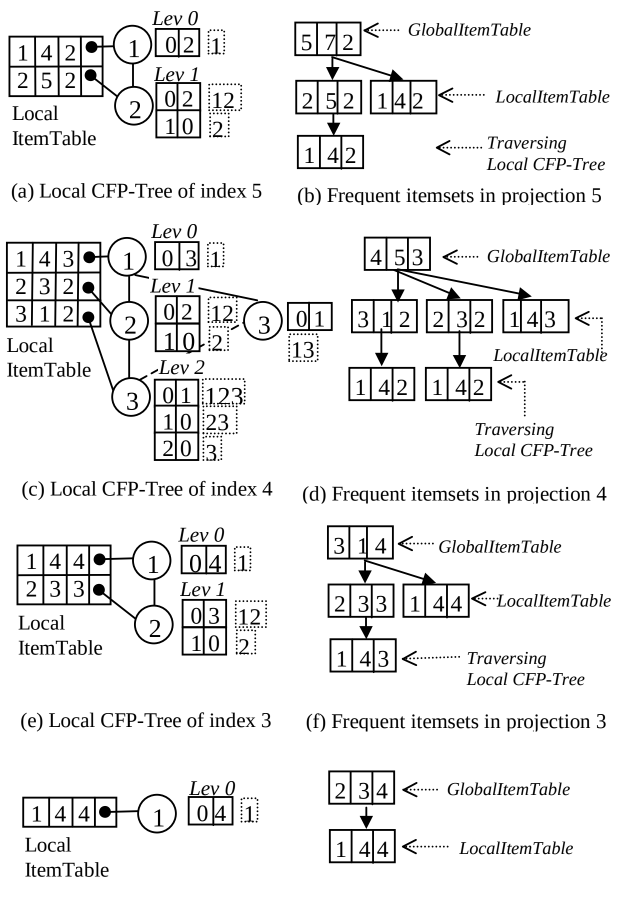
(d) Global CFP-Tree

Figure 3: CFP-Tree for the Sample Dataset

The mining process in *CT-PRO* is shown in Algorithm 2 and illustrated by the following example.

Example 1. Let the *CFP-Tree*, as shown in Figure 3d, be the input for the mining step in *CT-PRO* and suppose the user wants to get all the frequent itemsets with minimum support of two transactions (or 40%).

Figure 4 shows the *LocalCFP-Tree* and *LocalFrequentPatternTree* at each step during the mining process. *CT-PRO* starts from the least frequent item (index: 5, item: 7) in the *GlobalItemTable* (line 2). Item 7 is frequent and it will be the root of the *LocalFrequentPatternTree* (line 3). Then *CT-PRO* creates a projection of all transactions ending with index 5. This projection is represented by a *LocalCFP-Tree* and only contains *locally frequent items*. Traversing the *node-link* of index 5 in the *GlobalCFP-Tree* identifies the local frequent items that occur together with it. There are three nodes of index 5 and the path to the root for each node is traversed counting the other indexes that occur together with index 5 (lines 13-23). In all, we have 1 (2), 2 (1), 3 (1) and 4 (2) for index 5. As indexes 1,4 (item id: 4,5) are *locally frequent*, they are registered in the *LocalItemTable* and assigned new index ids (see Figure 4a). They also become the child of the *LocalFrequentPatternTree*'s root (lines 5-7). Together, the root and its children form frequent itemsets with length two.



(a) Local CFP-Tree of index 5 (b) Frequent itemsets in projection 5

(c) Local CFP-Tree of index 4 (d) Frequent itemsets in projection 4

(e) Local CFP-Tree of index 3 (f) Frequent itemsets in projection 3

(g) Local CFP-Tree of index 2 (h) Frequent itemsets in projection 2

Figure 4: Local CFP-Tree during Mining Process

After *local frequent items* for the projection have been identified, the *node-link* in the *GlobalCFP-Tree* is re-traversed and the path to the root from each node is revisited to get the *local frequent items* occurring together with index 5 in the transactions. These *local frequent items* are mapped to their index in the *LocalItemTable* on-the-fly, sorted in ascending order of their index id and inserted into the *LocalCFP-Tree* (lines 24-33). The first path of index 5 returns nothing. From the second path of index 5, a transaction 14 (1) is inserted into the *LocalCFP-Tree* and another transaction 14 (1) from the third path of index 5 also is inserted. In total, there are two

Algorithm 2 CT-PRO Algorithm: Mining Part

```
input   CFP-Tree
output  Frequent Itemsets FP

1   Procedure Mining
2     for each frequent item i ∈ GlobalItemTable
      from the least to the most frequent
3       Initialize LocalFrequentPatternTree
        with i as the root
4       ConstructLocalItemTable(x)
5       for each frequent item j ∈ LocalItemTable
6         Attach i as a child of x
7       end for
8       ConstructLocalCFPTree(x)
9       RecMine(x)
10      Traverse the LocalFrequentPatternTree
         to print the frequent itemsets
11    end for
12  end
13  Procedure ConstructLocalItemTable(i)
14    for each occurrence of node i in the CFP-Tree
15      for each item j in the path to the root
16          if j ∈ LocalItemTable
17            Increment count of j
18          else
19            Insert j into LocalItemTable with count = 1
20          end if
21      end for
22    end for
23  end
24  Procedure ConstructLocalCFPTree(i)
25    for each occurrence of node i in the CFP-Tree
26      Initialize mappedTrans
27      for each frequent item j ∈ LocalItemTable
        in the path to the root
28        mappedTrans = mappedTrans ∪ GetIndex(j)
29      end for
30      Sort mappedTrans in ascending order of item ids
31      InsertToCFPTree(mappedTrans)
32    end for
33  end
34  Procedure RecMine(x)
35    for each child i of x
36      Set all counts in LocalItemTable to 0
37      for each occurrence of node i in
        the LocalCFPTree
38        for each item j in the path to the root
39            Increment count of j in LocalCFPTree
40        end for
41      end for
42      for each frequent item k ∈ LocalItemTable
43        Attach k as a child of i
44      end for
45      RecMine(i)
46    end for
47  end
```

occurrences of transaction 14. Indexes 1 and 4 in the *GlobalItemTable* represent items 4 and 5 respectively. The indexes of items 4 and 5 in the *LocalItemTable* are 1 and 2 respectively and so the transaction 14 is inserted as transaction 12 in the *LocalCFP-Tree*. As the item index in the *GlobalItemTable* and *LocalItemTable* are different, the item id is always maintained for output purposes.

Longer frequent itemsets, with length greater than two, are extracted by calling the procedure *RecMine* (line 9). For simplicity, we have described this procedure (lines 34-47) using recursion but, in the program, it is implemented as a non-recursive procedure. Starting from the least frequent item in the *LocalItemTable*, (line 35), the *node-link* is traversed (lines 37-41). For each node, the path to the root in the *LocalCFP-Tree* is traversed counting the other items that are together with the current item. For example, in Figure 4a, traversing the *node-link* of node 2 will return the index 1 (2) and, since it is frequent, an entry is created and attached as the child of index 2 in the *LocalFrequentPatternTree* (lines 42-44). All frequent itemsets containing item 7 can be extracted by traversing the *LocalFrequentPatternTree* (line 10): 7 (2), 75 (2), 754 (2), 74 (2).

The process is continued to mine the next item in the *GlobalItemTable* in the *GlobalCFP-Tree* with indexes 4, 3, 2 and finally, when the mining process reaches the root of the tree of Figure 3d, it outputs 4 (5).

One major advantage of *CT-PRO* compared to *FP-Growth* is that *CT-PRO* avoids the cost of creating *conditional FP-Trees. FP-Growth* needs to create a *conditional FP-Tree* at each step of its recursive mining. This overhead adversely affects its performance, as the number of *conditional FP-Trees* corresponds to the number of frequent itemsets. In *CT-PRO*, for each frequent item (not frequent itemsets), only one *LocalCFP-Tree* is created and traversed non-recursively to extract all frequent itemsets beginning with the frequent item.

4. Time Complexity

In this section, the best-case and worst-case time complexity of *CT-PRO* algorithm is presented. Let $I = \{i_1, i_2, \ldots, i_n\}$ be the set of all n items, let transaction database D be $\{t_1, t_2, \ldots, t_m\}$, and let v be the total number of items in all transactions.

Lemma 2. In the best-case, the cost of generating frequent itemsets is $O(v + n)$.

Proof. *The best-case for the CT-PRO algorithm occurs when there is no frequent item. The algorithm has to read v items in all transactions and add the count of n items. The count of all n items are stored in the ItemTable and checked to determine whether there is any frequent item or not. If there is no frequent item, the process stops.* ■

Lemma 3. In the worst-case, the cost of generating frequent itemsets is

$$(v + n) + (v + 2^{n-1}) + \sum_{k=2}^{n} ((2^{(n-k)}-1)(2^{(n-k)}) + 2^{(k-2)}) = O(2^{2n}).$$

Proof. *The worst-case happens when all n items are frequent and all combinations of them are present in m transactions. CT-PRO has three steps: finding frequent items, constructing the CFP-Tree, and mining. The cost of finding frequent items has been provided by Lemma 2. The worst case for the GlobalCFP-Tree corresponds to a situation where all the possible paths exist. In constructing the GlobalCFP-Tree, all the transactions in the database are read (the cost is v) and inserted into the tree (the total number of nodes is* 2^{n-1}*). For the mining process, for each frequent item* f_k *where* $2 \le k \le n$, $2^{(k-2)}$ *nodes in the GlobalCFP-Tree are visited to construct a LocalCFP-Tree. The LocalCFP-Tree has* $(2^{(n-k)}-1)$ *paths that correspond to, at most,* $2^{(n-k)}$ *candidate itemsets. So the worst case mining cost is:*

$$\sum_{k=2}^{n} ((2^{(n-k)}-1)(2^{(n-k)}) + 2^{(k-2)}).$$

Therefore, the total worst-case cost of CT-PRO is

$$(v+n)+(v+2^{n-1})+\sum_{k=2}^{n} ((2^{(n-k)}-1)(2^{(n-k)}) + 2^{(k-2)}) = O(2^{2n})$$ ■

5. Experimental Evaluation

This section contains three sub-sections. In Section 5.1, we compare *CT-PRO* against other well-known algorithms including with *Apriori* [14, 15], *FP-Growth* [10] and recently proposed *OpportuneProject* (OP) [11] on the various datasets available at FIMI Repository 2003 [17]. In Section 5.2, we report the result of more comprehensive testing to determine the feasible performance range of the algorithms. Finally, in Section 5.3, we compare *CT-PRO* with the two best algorithms in the FIMI Repository 2003 [17], *LCM* [18] and *kDCI* [19].

5.1. Comparison with Apriori, FP-Growth and OpportuneProject

Six real datasets are used in this experiment including two dense datasets: *Chess* and *Connect4*; two less dense datasets: *Mushroom* and *Pumsb**; and two sparse datasets: *BMS-WebView-1* and *BMS-WebView-2*. The first four datasets are originally taken from UCI ML Repository [16] and the last two datasets are donated by Blue Martini Software [3]. All the datasets are also available at FIMI Repository 2003 [17].

We used the implementation of *Apriori* created by Christian Borgelt [21] by enabling the use of the prefix tree data structure. As for *FP-Growth*, we used the executable code available from its authors [10]. However, for comparing the number of nodes of *FP-Tree* to our proposed data structure, we modified the source code of *FP-Growth* provided by Bart Goethals in [22]. For *OpportuneProject* (*OP*), we used the executable code available from its author, Junqiang Liu [11].

All the algorithm were implemented and compiled using MS Visual C++ 6.0. All the experiments (except comparisons with algorithms in the FIMI Repository 2003 website [17]) were performed on a Pentium III PC 866 MHz with 512 MB RAM and 110 GB Hard Disk running on MS Windows 2000. All the reported runtime used in our charts is the total execution time, the period between input and output. It also includes the time of constructing all the data structures used in all programs.

Figure 5 shows the results of the experiment on various datasets. All the charts use a logarithmic scale for run time along the *y*-axis on the left of the chart. We did not plot the results in the chart if the runtime was more than 10,000 seconds. For a comprehensive evaluation of the algorithm's performance, rather than showing where our algorithm performed best at some of the support levels, all the algorithms were extensively tested on various datasets with a support level of 10% to 90% for dense datasets (e.g. *Connect4, Chess, Pumsb*, Mushroom*), a support level of 0.1% to 1% for the sparse dataset *BMS-WebView-1*, and a support level of 0.01% to 0.1% for the sparse dataset *BMS-WebView-2*. As the average number of items increases and/or the support level decreases, at some point, every algorithm 'hits the wall' (i.e. takes too long to complete).

CT-PRO outperforms others at *all* support thresholds on the *Connect4, Chess, Mushroom* and *Pumsb** datasets. On the sparse dataset *BMS-WebView-1, CT-PRO* is a runner-up, after *OP*, with only small performance differences (0.4 seconds to 0.49 seconds at a support level of 0.1% and 5.18 seconds to 7.69 seconds at 0.06%). Below the support level of 0.06%, none of the algorithms could mine the *BMS-WebView-1* dataset. On the sparse dataset *BMS-WebView-2*, a remarkable result is obtained. *Apriori*, which is known as a traditional FIM algorithm, outperforms *FP-Growth* at all support levels. *CT-PRO* is the fastest from a support threshold of 1% down to 0.4% and becomes the runner-up, after *OP*, at a support level of 0.3% down to 0.1% with small performance differences.

From these results, we can claim that, on dense datasets, *CT-PRO* generally outperforms others. On sparse datasets, the high cost of the tree construction reduces *CT-PRO* to runner-up. However, as the gap is very small, we can say that *CT-PRO* also works well for sparse datasets.

5.2. Determining the Feasible Performance Range

As mentioned earlier, sample datasets such as real-word *BMS* datasets [3] and the *UCI Machine Learning Repository* [16], which also are available at the FIMI Repository 2003 [17], have their own static characteristics and thus do not cover the full range of densities. An algorithm that works well for one dataset may not have the same degree of performance on other datasets with different dimensions. Dimensions, here, could be the number of transactions, number of items, average number of items per transaction, denseness or sparseness, etc. In this section, a more comprehensive evaluation of the performance of various algorithms is presented.

We generated ten datasets using the synthetic data generator [13]. The first five datasets contained 100 items with 50,000 transactions, and an average number of items per transaction of 10, 25, 50, 75, and 100. The second five datasets contained 100 items with 100,000 transactions, also with an average number of items per transaction of 10, 25, 50, 75, and 100. *CT-PRO, Apriori, FP-Growth* and *OP* were tested extensively on these datasets at a support level of 10% to 90%, in increments of 10%.

Figure 6 shows the performance comparisons of the algorithms on various datasets. The dataset name shows its characteristics. For example, *I100T100KA10* means there are 100 items, and 100,000 transactions with an average of 10 items per transaction. The experimental results show that the performance characteristics on databases of 50,000 to 100,000 transactions are quite similar. However, the runtime increases with the number of transactions.

The *Apriori* algorithm is very feasible for sparse datasets (with an average number of items in each transaction of 10 and 25). Its performance is good, as it consistently performs better than *FP-Growth* at all support levels. Although *Apriori* is slower than *CT-PRO* and *OP* using the two sparse datasets, its runtime is still acceptable to the user. (It needs only 60 seconds to mine the *I100T50KA25* dataset at the support level of 10%). However, on the datasets with an average number of items per transaction of 50, 75, and 100, *Apriori* performs worst and it can only mine down to a support level of 30%, 50%, and 70% respectively. These results confirm that, for dense datasets, if the support levels used are low, *Apriori* is infeasible.

FP-Growth performs worst at all support levels on the datasets with a low average number of items per transaction (i.e. 10 and 25). The fact that *FP-Growth* does not outperform *Apriori* on these two datasets shows that *Apriori* is more feasible than *FP-Growth* for sparse datasets. However, *FP-Growth* performs significantly better than *Apriori* for the larger average number of items in transactions.

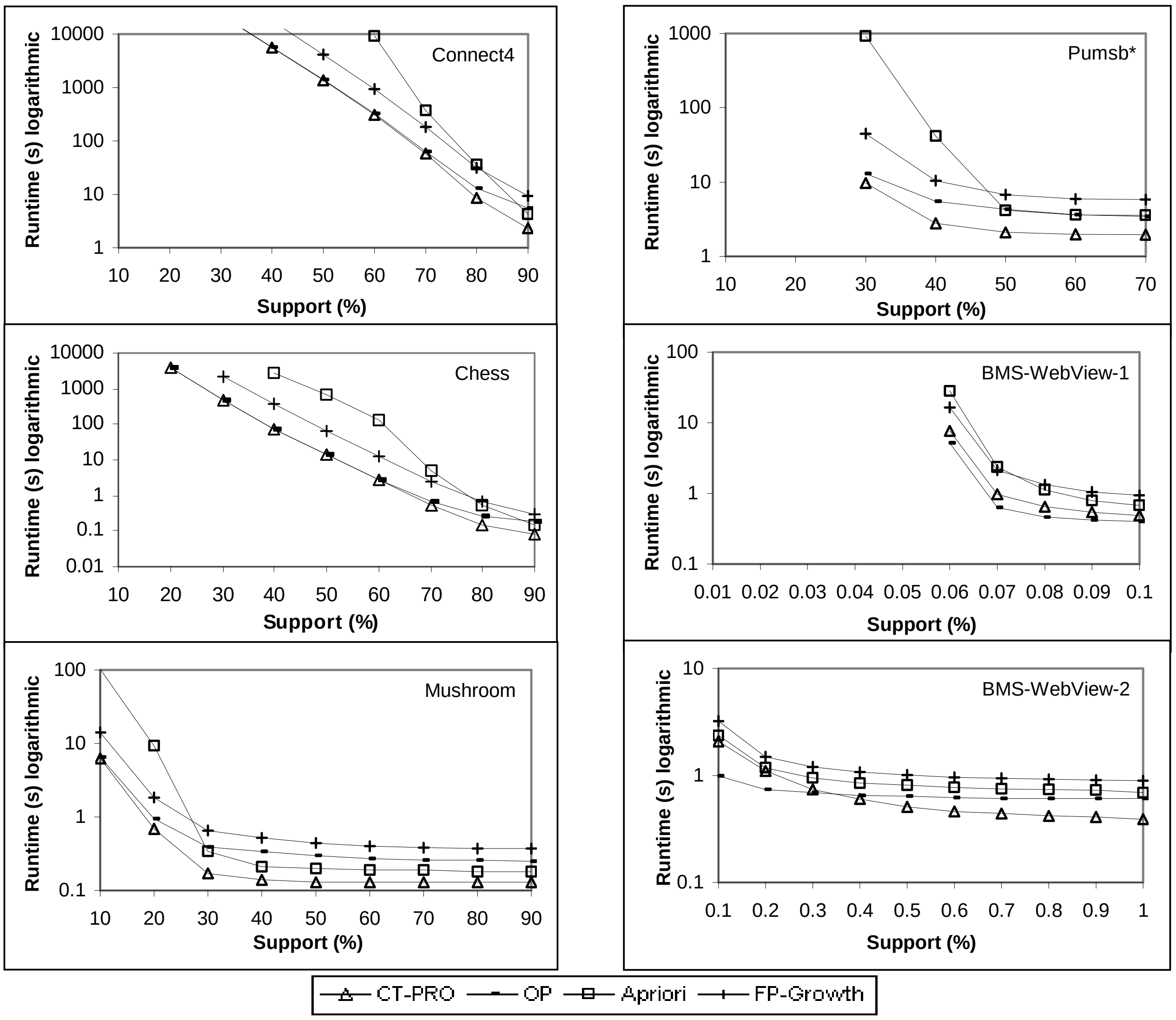


Figure 5: Performance Evaluation of *CT-PRO* Against Others on Various Datasets

Both *CT-PRO* and *OP* have larger feasible performance ranges compared to the other algorithms. *OP* does not perform well on the sparse datasets *I100T50KA10* and *I100T100KA10*. Its performance was even worse than *Apriori* on this dataset. However, it performs better than *Apriori* and *FP-Growth* on other datasets. On the datasets with an average number of items per transaction of 50, 75, and 100, *FP-Growth*, *CT-PRO* and *OP* can mine down to a support level of 20%, 40%, and 50% respectively.

CT-PRO can be considered the best among all other algorithms as it generally performs the best at most support levels. However, as the support level gets lower, its performance is similar to *OP*. Only at a support level of 10%, *OP* occasionally runs slightly faster than *CT-PRO* (e.g. at a support level of 10% on the *I100T50KA25* dataset).

5.3. Comparison with Best Algorithms in the FIMI Repository 2003

For comparison with the best algorithms in the FIMI Repository 2003 [17], we ported our algorithm *CT-PRO* to a Linux operating system and compared it with their two best algorithms: *LCM* [18] and *kDCI* [19]. We performed the experiments on a PC AMD Athlon™ XP 2000+ 1.6 GHz, 1 GB RAM, 2 GB Swap with 40GB Hard Disk running Fedora Core 1. All programs were compiled using g++ compiler.

Figure 7 shows the performance comparisons of algorithms that were submitted to FIMI 2003 on *Chess* and *Connect4* datasets. The figures are taken from [17]. On *Chess* dataset, *kDCI* is the best at a support level of 90% to a support level of 70%. Below that, *LCM* outperforms others. On *Connect4* dataset, at a support

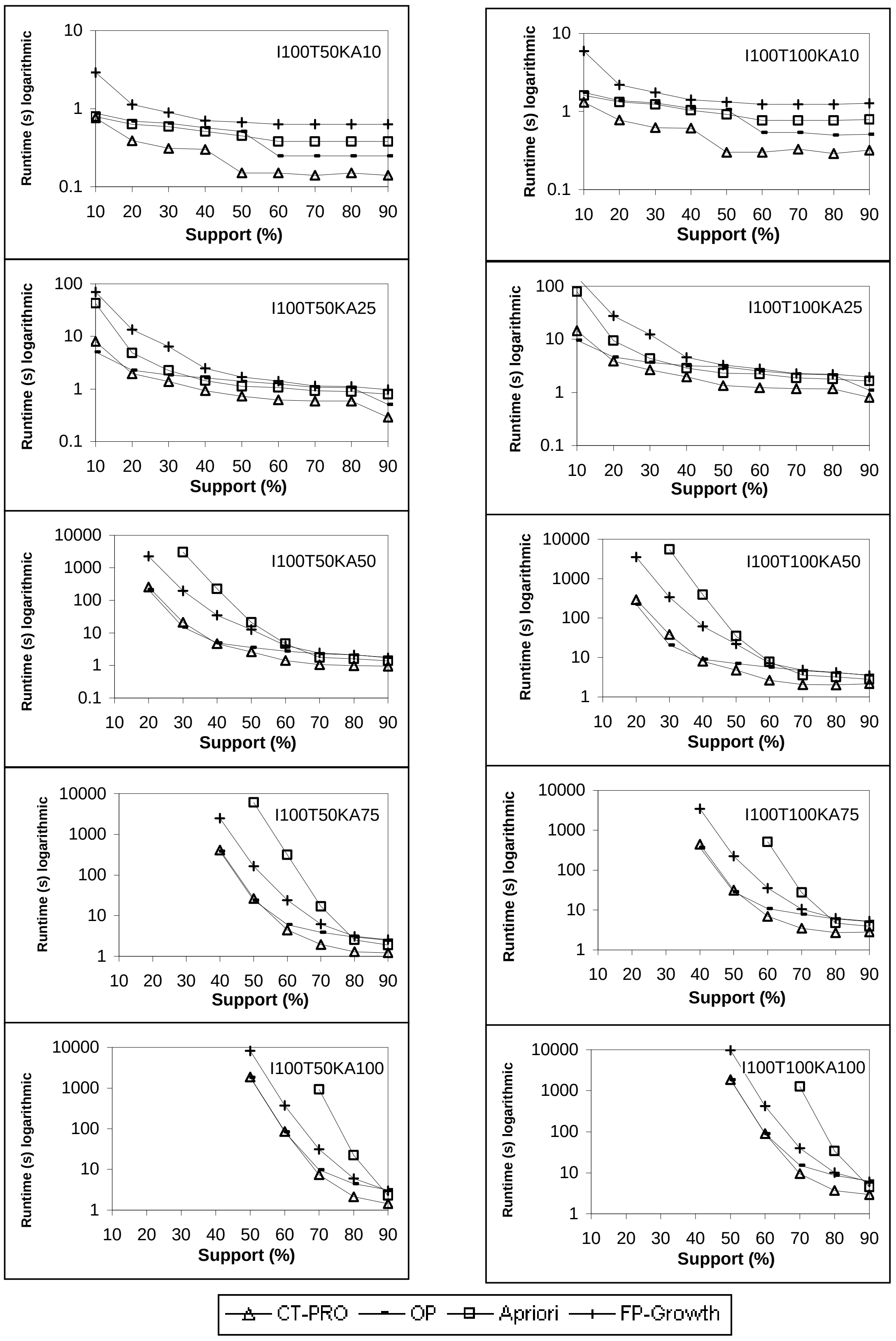


Figure 6: Performance Evaluation on Various Synthetic Datasets

level of 95% to a support level of 70%, *kDCI* is the best. Below that, *LCM* outperforms others. We can conclude that for higher support levels, *kDCI* is the best, but for lower support levels, *LCM* is the best. These best two algorithms are compared with *CT-PRO*.

The *kDCI* algorithm [19] is a multiple heuristics hybrid algorithm that able to adapt its behaviour during the execution. It is an extension of the *DCI* (Direct Count and Intersect) algorithm [23] by adding its adaptability to the dataset specific features. *kDCI* is also a resource aware algorithm which can decides mining strategy based on the hardware characteristics of the computing platform used. Moreover, *kDCI* also used counting inference strategy which originally proposed in [24].

The *LCM* (*Linear time Closed itemset Miner*) algorithm [18] uses the parent-child relationship defined on frequent itemsets. The search tree technique is adapted from the algorithms for generating maximal bipartite cliques [25, 26] based on reverse search [27, 28]. In enumerating all frequent itemsets, *LCM* uses hybrid techniques involving occurrence deliver or diffsets [29] according to the density of the database.

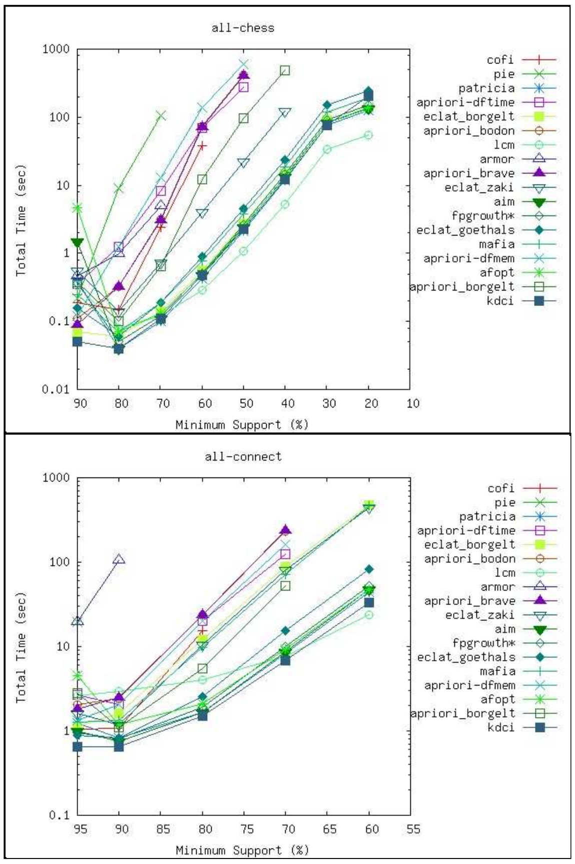


Figure 7: Performance Comparisons of Algorithms available in the FIMI 2003 Repository on *Chess* and *Connect4* Datasets [17]

Figure 8 shows the performance comparisons on *Chess* and *Connect4* datasets. From these results, *CT-PRO* always outperforms others at high support levels. For lower support levels, the performances of these three algorithms are similar. Since *LCM* and *kDCI* are the best algorithms in FIMI Repository 2003 on *Chess* and *Connect4* datasets, we can conclude that *CT-PRO* outperforms *all* other algorithms available in FIMI Repository 2003 [17] on *Chess* and *Connect4* datasets.

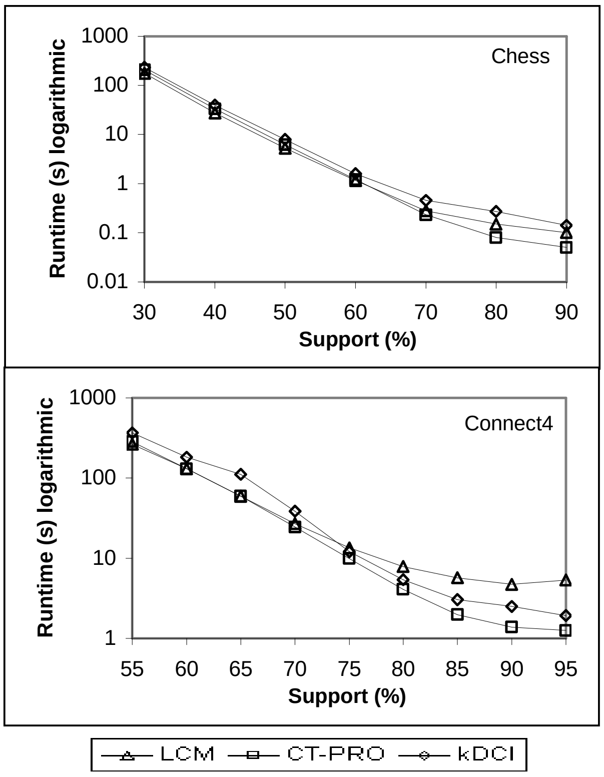


Figure 8: Performance Comparisons of *CT-PRO, LCM* and *kDCI* on *Chess* and *Connect4* Datasets

6. Conclusions

In this paper, we have described a new tree-based data structure named *CFP-Tree* that is more compact than *FP-Tree* used in *FP*-Growth algorithm. Depending on the database characteristics, the number of nodes in an *FP-Tree* could be up to twice as many as in the corresponding *CFP*-Tree for a given database. *CFP-Tree* is used in our new algorithm named *CT-PRO* for mining all frequent itemsets. *CT-PRO divides* the *CFP-Tree* into several projections represented also by *CFP-Trees.* Then *CT-PRO conquers* the *CFP-Tree* for mining all frequent itemsets in each projection.

CT-PRO was explained in detail using a running example and the best-case and worst-case time complexity of the algorithm also was presented. Performance comparisons of *CT-PRO* against other well-known algorithms, including *Apriori* [14, 15], *FP-Growth* [10] and *OpportuneProject* (*OP*) [11] also were reported. The results show that *CT-PRO* outperforms other algorithms at all support levels on dense datasets and also works well on sparse datasets.

Extensive experiments to measure the feasible performance range of the algorithms are also presented in this paper. A synthetic data generator is used to generate several datasets with varying number of both transactions and average number of items per transaction. Then the best available algorithms including *CT-PRO, Apriori, FP-Growth* and *OP* are tested on those datasets. The result shows that *CT-PRO* generally outperforms others.

In addition, to relate our research to the last workshop on frequent itemset mining implementations [17], we selected two best algorithms (*LCM* and *kDCI*) from FIMI Repository 2003 and compared their performance with *CT-PRO*. It was shown that *CT-PRO* performed better than the others.

7. Acknowledgement

We are very grateful to Jian Pei for providing the executable code of *FP-Growth*, Bart Goethals for providing his *FP-Growth* program, Christian Borgelt for the *Apriori* program, and Junqiang Liu for the *OpportuneProject* program.

8. References

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases", Proceedings of ACM SIGMOD International Conference on Management of Data, Washington DC, 1993, pp. 207-216.

[2] J. Hipp, U. Guntzer, and G. Nakhaeizadeh, "Algorithms for Association Rule Mining - A General Survey and Comparison", *SIGKDD Explorations*, vol. 2, pp. 58-64, July 2000.

[3] Z. Zheng, R. Kohavi, and L. Mason, "Real World Performance of Association Rule Algorithms", Proceedings of the 7th International Conference on Knowledge Discovery and Data Mining (KDD), New York, 2001.

[4] B. Goethals, "Efficient Frequent Pattern Mining", PhD Thesis, University of Limburg, Belgium, 2002.

[5] B. Liu, W. Hsu, and Y. Ma, "Integrating Classification and Association Rule Mining", Proceedings of ACM SIGKDD, New York, NY, 1998.

[6] Y. G. Sucahyo and R. P. Gopalan, "Building a More Accurate Classifier Based on Strong Frequent Patterns", Proceedings of the 17th Australian Joint Conference on Artificial Intelligence, Cairns, Australia, 2004.

[7] K. Wang, X. Chu, and B. Liu, "Clustering Transactions Using Large Items", Proceedings of ACM CIKM, USA, 1999.

[8] R. Agrawal and R. Srikant, "Mining Sequential Patterns", Proceedings of the 11th International Conference on Data Engineering (ICDE), Taipei, Taiwan, 1995.

[9] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang, "H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases", Proceedings of the IEEE International Conference on Data Mining (ICDM), San Jose, California, 2001.

[10] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation", Proceedings of the ACM SIGMOD International Conference on Management of Data, Dallas, TX, 2000.

[11] J. Liu, Y. Pan, K. Wang, and J. Han, "Mining Frequent Item Sets by Opportunistic Projection", Proceedings of ACM SIGKDD, Edmonton, Alberta, Canada, 2002.

[12] R. P. Gopalan and Y. G. Sucahyo, "High Performance Frequent Pattern Extraction using Compressed FP-Trees", Proceedings of SIAM International Workshop on High Performance and Distributed Mining (HPDM), Orlando, USA, 2004.

[13] IBM, "Synthetic Data Generation Code for Associations and Sequential Patterns", Intelligent Information Systems, IBM Almaden Research Center, 2002, http://www.almaden.ibm.com/software/quest/Resources/index.shtml.

[14] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules", Proceedings of the 20th International Conference on Very Large Data Bases, Santiago, Chile, 1994.

[15] C. Borgelt, "Efficient Implementations of Apriori and Eclat", Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI), Melbourne, Florida, 2003.

[16] C. L. Blake and C. J. Merz, "UCI repository of machine learning databases", Irvine, CA: University of California, Department of Information and Computer Science, 1998.

[17] FIMI, "FIMI Repository", 2003, http://fimi.cs.helsinki.fi.

[18] T. Uno, T. Asai, Y. Uchida, and H. Arimura, "LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets", Proceedings of the IEEE ICDM Workshop of Frequent Itemset Mining Implementations (FIMI), Melbourne, Florida, 2003.

[19] C. Lucchese, S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, "kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets", Proceedings of the IEEE ICDM Workshop of Frequent Itemset Mining Implementations (FIMI), Melbourne, Florida, 2003.

[20] Y. G. Sucahyo and R. P. Gopalan, "CT-ITL: Efficient Frequent Item Set Mining Using a Compressed Prefix Tree with Pattern Growth", Proceedings of the 14th Australasian Database Conference, Adelaide, Australia, 2003.

[21] C. Borgelt and R. Kruse, "Induction of Association Rules: Apriori Implementation", Proceedings of the 15th Conference on Computational Statistics, Berlin, Germany, 2002.

[22] B. Goethals, "Home page of Bart Goethals", 2003, http://www.cs.helsinki.fi/u/goethals.

[23] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, "Adaptive and Resource-Aware Mining of Frequent

Sets", Proceedings of the IEEE International Conference on Data Mining (ICDM), Maebashi City, Japan, 2002.

[24] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal, "Mining Frequent Patterns with Counting Inference", *ACM SIGKDD Explorations*, vol. 2, pp. 66-75, December 2000.

[25] T. Uno, "A Practical Fast Algorithm for Enumerating Cliques in Huge Bipartite Graphs and Its Implementation", Proceedings of the 89th Special Interest Group of Algorithms, Information Processing Society, Japan, 2003.

[26] T. Uno, "Fast Algorithms for Enumerating Cliques in Huge Graphs", Research Group of Computation, IEICE, Kyoto University 2003.

[27] D. Avis and K. Fukuda, "Reverse Search for Enumeration", *Discrete Applied Mathematics*, vol. 65, pp. 21-46, 1996.

[28] T. Uno, "A New Approach for Speeding Up Enumeration Algorithms", Proceedings of ISAAC'98, 1998.

[29] M. J. Zaki and C. Hsiao, "CHARM: An Efficient Algorithm for Closed Itemset Mining", Proceedings of SIAM International Conference on Data Mining (SDM), Arlington, VA, 2002.