CSCI6405 Project - Association rules mining

Xuehai Wang  Xiaobo Chen  Chen Shen
xwang@ca.dalc.ca  xiaobo@ca.dal.ca  cshen@cs.dal.ca
B00182688  B00123238  B00188996

December 7, 2003

Contents

1 Introduction: 2

2 System Design 2
  2.1 Dataset and implementation environment .............. 3
    2.1.1 Dataset ........................................... 3
    2.1.2 Implementation environment .................. 3
  2.2 Apriori Algorithm .................................. 3
    2.2.1 Algorithm .......................................... 3
    2.2.2 Design / Implementation issues ................. 4
    2.2.3 Overall architecture of code ................... 5
  2.3 Apriori with hash tree ................................ 5
    2.3.1 Algorithm .......................................... 5
    2.3.2 Design / Implementation issues ................. 6
    2.3.3 Overall architecture of code ................... 6
  2.4 FP-tree algorithm .................................. 7
    2.4.1 Data-Structure .................................. 7
    2.4.2 Construction of FP-tree ....................... 8
    2.4.3 A few concepts of FP-tree ..................... 8
    2.4.4 Mining frequent patterns ...................... 10
    2.4.5 Design / Implementation issues ............... 10
    2.4.6 Overall architecture of the code ........... 12

3 Evaluation 12
  3.1 Experiment results .................................. 12
    3.1.1 Apriori v.s. Hashtree ........................... 12
  3.2 Apriori v.s. FP tree ................................ 14
    3.2.1 Experiment I ..................................... 14
    3.2.2 Experiment II .................................... 16
    3.2.3 Experiment III ................................... 16
    3.2.4 Evaluation ....................................... 17
1 Introduction:

Association rule mining, first proposed by Agrawal, Imielinski and Swami, try to “Finding frequent patterns, associations, correlations, or casual structures sets of items or objects in transaction database, relational database, etc.” On the other word, we want to find out the relation or dependency of occurrence of one item based on occurrence of other items.

Finding association rule is a major part in data mining [2]. Apriori algorithm is the basic algorithm for association rule mining. Since association rule mining is dedicated to handle the ultra large amount of data, so the time complexity and resource complexity have to be carefully considered.

One of the most popular algorithms of finding association rules is Apriori. [1] But in this algorithm, the procedure of producing candidate set will consume big amount of time. Someone might think that Apriori algorithm has some defects since it is not fast. So there are some other algorithms proposed based on Apriori algorithm. Those algorithms try to introduce some useful ideas into the basic algorithm to benefit the performance. (Lots of algorithms and the comparison of them have been introduced in [2].)

Hence we want to know what is the key point for an algorithm to gain good performance, and what is its limitations. In this project, we will implement some association rule mining algorithms. And then we will try to compare them to find the virtue or defect of each algorithm, and hence to lean the core idea of association mining.

In this project, we will examine Apriori algorithm, Apriori algorithm with hash tree structure, and FP-tree algorithm [3]. Then we will implement each of them and do some simple comparison. Then we will give our conclusion.

We will describe three algorithms, and how will we implement them. In this section, we will first describe the implement environment and the experiment dataset; then we will present the experiment result and draw our conclusions.

2 System Design

Apriori algorithm is most general used in association rule mining. But it still has some drawbacks. To make a good performance, some other algorithms were proposed. In this project, we will examine Apriori algorithm, Apriori algorithm with a hash tree structure, and FP tree algorithms.
2.1 Dataset and implementation environment

2.1.1 Dataset

We choose IBM data generator to generate the data for program test. The data generator can generate the dataset with dedicated transaction size, item size, average number of items in a transaction and average minimum support.

We also use a fix dataset to test our analysis. In this dataset, the total number of transactions of the data used in this experiment is 683, and there are total 18896 items. Belows are some samples of the dataset:

```
19 12 438 957 976 1326 2011 2436 2813 3247 3768 3966 4042 4565 4703 5568 7976 9435 9452 9975
25 141 513 776 813 1081 1197 1565 1644 2073 2313 3794 3821 3907 4999 5604 6048 6077 6771 7423 7644 8151 8610 8774 9058 9160
33 92 457 1793 1827 2112 2117 2221 2341 3290 3356 3852 3884 4042 4836 4932 4980 5241 5401 6152 6160 6235 6354 6497 7111 7185 7285 7293
7717 7803 8557 9014 9050 9682
```

The first number of each transaction indicates how many items in this transaction. And 'tab' is used to seperate each item in a transaction.

2.1.2 Implementation environment

We implement and run our code on “torch”. “torch” is a multi-cpu system, and the operation system is unix. The drawback of “torch” is that it’s a multi-user system too. So the result will be different when the system is busy or idle. But we will see in the experiment that the performances in different time periods are in the same order. So the result can support our conclusion.

2.2 Apriori Algorithm

One of the most popular algorithms of finding association rules is Apriori [2]. The principle of apriori algorithm is: A candidate generation-and-test Approach. [1] Given a frequent itemset, its subset must be frequent. A set is infrequent, its super set will not be generated and tested

2.2.1 Algorithm

Apriori uses $k$ – itemsets, which has $k$ items belonging to the set of items $I$, to generate $(k+1)$ itemsets. The main idea of Apriori is: Any subset of a
frequent item set must be a frequent set, so if a subset of a frequent itemset is not a frequent set, the frequent itemset should not be generated or test. Following is the P-Code of Apriori: [2]

```c
// L_k is the set of frequent k-itemsets.
k_1 ← frequent1 − itemsets; k ← 2
while L_{k−1} ≠ φ do
    generate C_k from L_{k−1} // C_k is the set of candidate k-itemsets.
    for all t ∈ D do
        Increment the count of all candidates in C_k that are contained in t.
    end for
    L_k = All candidates in C_k with minimum support.
end while
return ∪_k L_k
{ Subroutine of generating C_k from L_{k−1} }
{ Step 1: Self-joining L_{k−1} }
insert into C_k
select p.item_1, p.item_2, ..., p.item_{k−1}, q.item_{k−1}
from L_{k−1}p, L_{k−1}q
where p.item_1 = q.item_1, ..., p.item_{k−2} = q.item_{k−2}, p.item_{k−1} < q.item_{k−1}
{ Step 2: Pruning }
for all itemsets c in C_k do
    for all (k − 1) − subsets s of c do
        if s ⊄ L_{k−1} then
            delete c from C_k
        end if
    end for
end for
```

2.2.2 Design / Implementation issues

C++ is used in our programing. k − itemset is stored in a double linked list, and a vector is used to hold all itemset list.

Key issues are how to implement join and count support. The dataset file is preprocessed to make sure that items in each transaction are sorted in ascending order. And the candidate set in k − itemset is sorted in ascending order too. In count support step, if the subset gotten from the transaction is greater than the largest itemset in the candidate set list, than this transaction will be discarded.
2.2.3 Overall architecture of code

We use the same implementation of assignment 4.

1st - Provide a interface to prompt user to input:
   1. data file
   2. min support
   3. min confidence

2nd - Transfer data file to:
   1. file "input.txt", in which each item is replaced as integer and
      stored in a sorted order
   2. file "map.txt", in which the string-to-integer relation of item
      is stored

3rd - Implement Apriori algorithm to find the frequent patterns.
      The steps are exactly the same as described as above.

2.3 Apriori with hash tree

In pratical, we find that the most time consuming steps are: join to create candidate, add support. In Add support step, the algorithm will scan all transactions to find if the existance each $k$ pattern, then add the support number according to the scan result.

Since the candidate set can be very huge. For $k$-itemset, if the candidate size is $l$, and there are $n$ transactions of average length $m$. Then the worst time complexity is:

$$O(n.l.\frac{m^k}{k})$$

Now we store all candidate in a hash tree structure, then find a pattern will be:

$$O(n.log(l).\frac{m^k}{k})$$

2.3.1 Algorithm

It is still Apriori algorithm. In candidate set generation step, we will put all patterns in a hash tree structure. There are two kinds of nodes: interior node will contain a hash table, and leaf node will contain the $k$-itemset as well as its count.

First create a node as leaf node, and the first node is root node. New itemset are stored in this node. If there are more than two itemsets stored in the node, then this node will be labeled as an interior node which will store a hash table. Hash those itemsets stored in this node to new nodes.
To find a pattern, just follow hash function until to the leaf node. And the compare the itemset with the subset of transactions. Figure 1 [6] gives an example of hash function.

Figure 1: A transaction database as running sample

2.3.2 Design / Implementation issues

A new c++ class: HashTree class was implemented to store the frequent pattern. The key step is how to split a leaf node to an interior node as well as a set of hash nodes. The size of hash table will choose half of item numbers. There is also a linked list to link all leaf nodes together in a sorted order. This will benefit the join step for next level candidate generation.

2.3.3 Overall architecture of code

The different steps is:

3rd - Implement Apriori algorithm to find the frequent patterns.

- Generate candidate. Join $k-1$ itemset and creat $k$ items, and then put it in the hash tree.
- Scan transactions and count support: For each $k$ subset of the transaction, follow the hash function to the leaf node. If
there is a same item set of the subset, then add the support; otherwise go on the next sub set.

- Continue the steps until all transactions are scanned.

2.4 FP-tree algorithm

As we learn from the lecture that the association rules mining is an active research topic. “Frequent pattern mining plays an essential role in association mining.” [4] Among those algorithms mining frequent patterns, Apriori-like approach is broadly adopted [4]. But the drawback of the Apriori-like approach is that it requires a lot of join operation and need to generate a lot candidate itemsets to find the frequent patterns. However, ”the candidate generation could be costly, especially when long patterns exist in the data set.” [4]

To get an idea about how the Apriori-like approach may be inefficient, Han pointed out that, e.g. “if there are 104 frequent 1-itemsets, the Apriori algorithm will need to generate more than 107 length-2 candidates and accumulate and test their occurrence frequencies.” [4] And for a pattern of length 100, the generated candidates may be close to 2100 1030. [4] Also the Apriori approach will “repeatly scan the dataset and check a large set of candidates by pattern matching” [4].

Based on those observations, Han, etc. [4], proposed a new algorithm, FP-tree (Frequent Pattern Tree) algorithm to mining frequent patterns from large data set. It turned out to be more efficient than other approaches, especially when the minimum support rate is very small, i.e. a lot frequent patterns will be mined.

2.4.1 Data-Structure

To use FP-tree algorithm to find frequent patterns, the main data structure is the FP-tree the construction of which will be described later, the crucial point of this data structure is to make most use of the common shared frequent items and put them in the toppest of the tree, i.e., the more frequently occurred a item, the most likely will it be in the beginning of the tree.

Associated with each FP-tree, there is a header table each entry of which will map a linked list to the item in the tree such that traversing the node with same item name will be readily achieved.

Each node has at least three fields: \textit{item name}, which keeps the name of the item; \textit{count}, which keeps how many counts this node has; \textit{node link}, which is used to traversing those nodes with the same \textit{item name} in this FP-tree.
2.4.2 Construction of FP-tree

Before constructing a FP-tree from a data set, we need to do the following preprocessing: First, create a frequency list, F\textit{list}, of the items in descending order; second, sort each transaction in the data set according to the F\textit{list}. After that, we may construct the FP-tree. The following example is exactly from Han, et al. [4].

Suppose the data set is as in Figure 2 [4] and the minimum support = 3. We first get the F\textit{list} : \(< f : 4, c : 4, a : 3, b : 3, m : 3, p : 3 >\) and the sorted transactions as in the right column of Table 1. Please note that the sorted transactions have already removed those items which failed the minimum support. Then based on those sorted transactions, using the algorithm 1 in [4] to build the FP-tree and the tree is as in Figure 3 and 4:

<table>
<thead>
<tr>
<th>TID</th>
<th>Items Bought</th>
<th>(Ordered) Frequent Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>f, a, c, d, g, s, m, p</td>
<td>f, c, a, m, p</td>
</tr>
<tr>
<td>200</td>
<td>a, b, c, f, l, m, o</td>
<td>f, c, a, b, m</td>
</tr>
<tr>
<td>300</td>
<td>b, f, h, j, o</td>
<td>f, b</td>
</tr>
<tr>
<td>400</td>
<td>b, c, k, s, p</td>
<td>c, b, p</td>
</tr>
<tr>
<td>500</td>
<td>a, f, c, e, l, p, m, n</td>
<td>f, c, a, m, p</td>
</tr>
</tbody>
</table>

Figure 2: A transaction database as running sample

Figure 3: The algorithm to build the above FP-tree is given in algorithm 1 as in [4]

transactions as in the right column of Table 1. Please note that the sorted transactions have already removed those items which failed the minimum support. Then based on those sorted transactions, using the algorithm 1 in [4] to build the FP-tree and the tree is as in Figure 3 and 4:

2.4.3 A few concepts of FP-tree

Conditional pattern base of a given item, say p, is the tree formed such that the leaves of the tree are nodes with m as its item name. So, the conditional
Algorithm 1 (FP-tree construction)

**Input:** A transaction database $DB$ and a minimum support threshold $\xi$.

**Output:** Its frequent pattern tree, FP-tree.

**Method:** The FP-tree is constructed in the following steps.

1. Scan the transaction database $DB$ once. Collect the set of frequent items $F$ and their supports. Sort $F$ in support descending order as $L$, the list of frequent items.

2. Create the root of an FP-tree, $T$, and label it as “null”. For each transaction $Trans$ in $DB$ do the following.

   Select and sort the frequent items in $Trans$ according to the order of $L$. Let the sorted frequent item list in $Trans$ be $[p[P]]$, where $p$ is the first element and $P$ is the remaining list. Call \textit{insert\_tree}([p[P]],T).

   The function \textit{insert\_tree}([p[P]],T) is performed as follows. If $T$ has a child $N$ such that $N.item-name = p.item-name$, then increment $N$’s count by 1; else create a new node $N_1$, and let its count be 1, its parent link be linked to $T$, and its node-link be linked to the nodes with the same item-name via the node-link structure. If $P$ is nonempty, call \textit{insert\_tree}(P,N_1) recursively.

Figure 4: The algorithm to build the above FP-tree is given in algorithm 1 as in [4]
pattern base of item p is the tree: (¡fcam:2, ¡cb:1). We note that the count of each path is the same as the count of the leaf (p node). [4]

Conditional FP-tree of given conditional pattern base, as that from the previous conditional pattern base, is the FP-tree built using algorithm 1 by taking each path of the conditional pattern base as a transaction. We need be careful here that we may need to remove those items in the following means: 1) build a new F_list for those items in the base but keep the order of the new F_list as that of the older one; 2) based on this new F_list, we first prune those items which failed the minimum support. After these steps, we can build the conditional FP-tree for given conditional pattern base. E.g, for the previous one, the conditional FP-tree of p is ¡c:3 after pruning. [4]

Before we continue with the mining part of this algorithm, We would like to bring attention to you that, in the following algorithm 2, when it say that “recusively build conditional FP-tree”, it means not only the tree, but also the new header table corresponding to the new conditional tree!

2.4.4 Mining frequent patterns

After building the FP-tree, the frequent pattern can be mined recursively using the algorithm 2 as given in [4]. The details of the mining are clearly described in the section 3 of [4]. We only give the algorithm here which is exactly from [4] and some comments we learned when We read and implemented the algorithm. [4]:

α is the item in the header table associated with the given FP-tree. The FP-tree of β is actually that of α. When initially called, the FP-tree as for previous example is the FP-tree built from the total transactions which is normally should not be a single path tree. To make this algorithm more clear, we may add such statement between line (6) and (7): together, the new header table of Treeβ, because we need this table to traverse the node-link for later possible conditional tree build-up.

2.4.5 Design / Implementation issues

In the design/implementation phase, the first decision is how to choose/design the appropriate data structures to represent the concepts mentioned in the previous sections. It turned out that the STL in C++ provides us powerful tools to implement those concepts which can be tell from the codes submitted.

The most difficult part in implementing this algorithm is that part of algorithm 2, i.e., FP_growth() method which is basically a recursive method. The difficulty originates from the ambiguous of the description of this algorithm which has been clarified in the previous section just underneath algorithm 2.
Algorithm 2 (FP-growth: Mining frequent patterns with FP-tree by pattern fragment growth)

**Input:** FP-tree constructed based on Algorithm 1, using DB and a minimum support threshold \( \xi \).

**Output:** The complete set of frequent patterns.

**Method:** Call FP-growth (FP-tree, null).

Procedure FP-growth \((Tree, \alpha)\)  

\[
\text{(1) if } Tree \text{ contains a single path } P \\
\text{(2) then for each combination (denoted as } \beta) \text{ of the nodes in the path } P \text{ do} \\
\text{(3) generate pattern } \beta \cup \alpha \text{ with } support = \\
\text{minimum support of nodes in } \beta; \\
\text{(4) else for each } a_i \text{ in the header of } Tree \text{ do} \\
\text{(5) generate pattern } \beta = a_i \cup \alpha \text{ with} \\
\text{support = } a_i; \text{support;} \\
\text{(6) construct } \beta \text{'s conditional pattern base and then } \beta \text{'s conditional FP-tree } Tree_\beta; \\
\text{(7) if } Tree_\beta \neq \emptyset \\
\text{(8) then call FP-growth } (Tree_\beta, \beta) \quad \}
\]
2.4.6 Overall architecture of the code

The code basic follow the procedure described in [4], i.e., first build the FP-tree from the original data set, then call $FP.growth()$ method in which it is recursively called to mine the frequent patterns from the data set.

3 Evaluation

We will then run the three programs a set of times according to different minimum support. And then we try to draw some conclusions.

3.1 Experiment results

3.1.1 Apriori v.s. Hashtree

We test the correctness on some small datasets. Then we run the programs on the data set we described before. Below is the result. And we can see the graphically comparision in Figure 5. Noticed that when minimum support is less than 11, the time is much longer. The reason is that there are more large frequent pattern such as 4-itemset, 5-itemset generated. And join step and support counting step will consume much time.

Since the linear scale can not suit all results, we will us logarithm on the time scale.

In this experiment, we also noticed that the if $k \leq 6$, then join step dominates the time consumed; if $k \geq 7$, then the major time consuming part is support count step. The reason is the average transactions size of the dataset is greater then 25. So the time spent on generation of the $k$ subset, where $k \geq 7$ out of 30, is huge.

And in this experiment, we observed that Apriori algorithm is a little faster than hash tree algorithm, and it’s not what we expected before. The reason is the way how we implement apriori. In the implementation, both candidate sets and transaction are sorted. If a subset of the transaction is greater than biggest frequent in the candidate, and then this transaction can be discarded. And this situation occurs a big part in our experiment.

<table>
<thead>
<tr>
<th>Support rate</th>
<th># of Support</th>
<th>Time (sec, Apriori)</th>
<th>Time (Hash tree)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>7</td>
<td>21097</td>
<td>228684</td>
</tr>
<tr>
<td>1.2</td>
<td>9</td>
<td>18017</td>
<td>23810</td>
</tr>
<tr>
<td>1.5</td>
<td>11</td>
<td>1099</td>
<td>1715</td>
</tr>
<tr>
<td>2.0</td>
<td>14</td>
<td>5.0</td>
<td>4.0</td>
</tr>
<tr>
<td>2.5</td>
<td>18</td>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td>3.0</td>
<td>21</td>
<td>4.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Table 1: Run time of mining frequent patterns of variant support rate
dataset. Meanwhile, the hash tree algorithm have to scan all subset of a transaction.

Recall that the assumption of build hash tree is based on:

- The total number of candidates can be very huge. \[3\]
- One transaction may contain many candidates. \[3\]

But in our experiment dataset, the density of candidate in each transaction is low. But the transaction is very long. If we let minimum support small enough, there will be lots of long itemsets generated. But the candidate size is small. So to generate sub set from the transaction will be very long and dominate the time spent.

Hence in future work, we may improve it such as:

1. If support count is less than 7, then hash tree algorithm is applied.

2. Otherwise, candidate is stored in a linked list sorted. And we will try to read a block of transactions in the memory. And then scan all candidate sets to see if all items in a frequent itemset are in a transaction. Based on this to add count for each itemset.

So, we can see the parameter of data set is very important, too. We may need choose different algorithm according to different dataset.

![Figure 5: Run time of Apriori and Hash tree](image)

Figure 5: Run time of Apriori and Hash tree
3.2 Apriori v.s. FP tree

After implementing the FP-tree algorithm and test its correctness by using a small data set from which the results can be obtained manually, we run the program for the same data set of [4], but cutting the size of it due to the slow computing environment on torch. The data set is D2 from Han’s website [5]. The shown time is the CPU time of the process took to mining the frequent pattern. Since the congestion varies at different time intervals, the execution of the code may vary greatly.

In this experiment, we implement the Apriori algorithms a little different as above. We only want to compare the algorithms. So we will read all transactions into memory once the program begin to run. And no more I/O scan. Hence the algorithm will be faster as preieve experiment.

3.2.1 Experiment I

First, we run the FP-tree and Apriori algorithm on the data set (my-data.txt). The characteristics of this experiment is that there are generally more than 20 items in each transaction and the total items amounts to 4069. So we can expect long patterns and the great number of short size frequent itemsets.

The result are listed in Table 2

<table>
<thead>
<tr>
<th>Support rate</th>
<th># of Support</th>
<th>Time (seconds, FP-tree)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3</td>
<td>3</td>
<td>247.55</td>
</tr>
<tr>
<td>0.5</td>
<td>4</td>
<td>167.11</td>
</tr>
<tr>
<td>0.7</td>
<td>5</td>
<td>109.75</td>
</tr>
<tr>
<td>0.8</td>
<td>6</td>
<td>72.57</td>
</tr>
<tr>
<td>0.9</td>
<td>7</td>
<td>47.06</td>
</tr>
<tr>
<td>1.1</td>
<td>8</td>
<td>28.25</td>
</tr>
<tr>
<td>1.2</td>
<td>9</td>
<td>18.18</td>
</tr>
<tr>
<td>1.5</td>
<td>11</td>
<td>7.14</td>
</tr>
<tr>
<td>2.0</td>
<td>14</td>
<td>1.39</td>
</tr>
<tr>
<td>2.5</td>
<td>18</td>
<td>0.12</td>
</tr>
<tr>
<td>3.0</td>
<td>21</td>
<td>0.04</td>
</tr>
</tbody>
</table>

Table 2: Run time of mining frequent patterns of variant support rate

And this result can be graphically displayed as the following 2 figures:

In this experiment, when the threshold support number is too small, FP-tree algorithm encounters an inherent shortcoming from its recursion nature such that the computing resources is consumed up to cause its failure.

Its comparison with Apriori is shown in Figure 8. It’s obvious that FP-tree runs much faster, in a few order faster.
Figure 6: Run time of mining frequent pattern using FP-tree Algorithm v.s threshold support rate.

Figure 7: Run time of mining frequent pattern using FP-tree Algorithm v.s threshold support rate.
We will present the following experiments only in the comparison with Apriori as it here in Figure 8 so we can see clearly the difference of these two algorithms.

Figure 8: FP-tree vs. Apriori algorithm

3.2.2 Experiment II
In this experiment, we reduced the number of items in data set (mydata.txt) by limiting the number of items of each transaction to 10 items. So the total number of items becomes 2068 though the total number of transactions is still 638.

The comparison is presented in Figure 9. We can clearly see that again the FP-tree is faster in a matter of a few order than Apriori algorithm. This conclusion is independent of the data set.

We also notice that when the number of items becomes smaller, FP-tree can run smoothly even the threshold support rate is very small. This indicates that the number of items in the data set has an effect on this algorithm. This is because that when the item number become smaller, the tree depth reduced accordingly such that the recursion call may not go so deep as to cause the depletion of computing resourced.

3.2.3 Experiment III
The last experiment was done on the D2 data set (T25I20D100k_data.gz.txt), but constraining it such that each transaction has only 5 items, so there are
total 100000 transactions and 2826 items. So the number of transactions are far more than the previous two experiments. For this experiment, we want to see how FP-tree algorithm performs on large data set, i.e., huge number of transactions. It turned out FP-tree still can run smoothly for very small support rate, the main reason for this is the fact the minimum support number is still far larger than 1 even the support rate is small due to the huge number of transactions. (E.g., for minimum support rate 0.05, the minimum support number is \(51 >> 1\)) From those experiments, it is clear that what matter is the absolute support number not the support rate.

The result is shown in Figure 10

3.2.4 Evaluation

In the experiment, we notice that the mining of frequent patterns using FP-tree is more efficient than other means, such as Apriori, especially when the threshold support number is small, i.e., the size of the frequent itemset is small, which means more frequent patterns need to be mined. This is the good part of FP-tree algorithm. But at the same time, there is a potential limitation on FP-tree algorithm as we encounter in experiment I.

As we know previously that the \(FP\_growth()\) method is recursively called in the mining process. And conditional FP-trees will be generated in this recursive call. There comes the potential issue with this algorithm, if there are many items and long patterns may exist in the dataset such

![Figure 9: FP-tree vs. Apriori algorithm](image)
that the conditional tree built at each recursive call is such that only a few nodes/items will be excluded, so that the computing resources may be used up before the mining results can be generated. This may happen if the threshold support number is close to 1 and many items in the data set, and it is actually the situation in the data set (mydata.txt).

4 Conclusion

We can see that in this experiment, hash tree algorithm does not show its advantage to original apriori algorithm. And we noticed that parameter of data set is very important, too. We should choose different algorithm according to different dataset.

Theoretically and experimentally, it shows that FP-tree algorithm is a better means to mining frequent patterns from large data set. (By large, it mainly means that there are a lot of frequent patterns and long patterns in the data set.) And it works very fast due to its construction nature of this algorithm.

But, we also need to be aware of one of the potential drawbacks rooted from the recursion algorithm which when used in FP-tree to mine the frequent patterns may cause the failure of this algorithm, i.e., when the frequent

![Figure 10: FP-tree algorithm running on the T25I20D100k data set with constrain that each transaction has only 5 items.](image)

Figure 10: FP-tree algorithm running on the T25I20D100k data set with constrain that each transaction has only 5 items.
pattern size is close to 1 and the size of long pattern is such that during recursively building the conditional FP-trees, the computing resources may be used up if the recursive call tree is too deep.

Experiments show that the performance of the all three algorithms are mostly determined by the absolute minimum support number and the nature of the data set. We also can see, Apriori algorithm and hash tree based algorithm is easy to implement. And dataset can be seperated. So we can easily combine it to other means such as parallel computing and to speed the association rule mining.

5 Acknowledgement

We sincerely thank Dr. Qigang Gao for his direction and advice in the process of our project. We also thank Dr. Jiawei Han for the data set used in the project and his excellent paper from which we can understand and implement the FP-tree algorithm.
References


[6] Data Mining: Concepts and Techniques, Jiawei Han and Micheline Kamber, Department of Computer Science University of Illinois at Urbana-Champaign, www.cs.uiuc.edu/~hanj