A High Performance Frequent Itemset Mining Algorithm Using Confidence Frequent Pattern Tree

Kun-Ming Yu, Bin-Chang Wu
Department of Computer Science and Information Engineering
Chung Hua University, Hsin-Chu, Taiwan, R.O.C.
yu, aquavit@pdlab.csie.chu.edu.tw

Abstract

Various processing methods for association data mining are presently being looked into. Most of them focus on data structure and computation improvement. The data structures usually have a high degree of data compression ratio and can express the original information from the database with integrity. There is also no need to obtain information from the database again. However, not many studies concentrate on using known frequent item sets to increase system performance. In order to avoid repeating the calculation of known frequent items to speed up the data mining process, a new tree structure to store all known frequent item sets and a header table to create a frequent item linking list are proposed. The experimental results showed that the proposed procedure performs better compared with existing data mining procedures.

1. Introduction

Data mining is one of the most useful techniques to discover the meaningful information from database. Many data mining algorithms iterate calculating known frequent items or item sets that will consume a large amount of computing time and memory space. If the known information can be used in the mining process, then performance will improve.

FP-Tree [4] is often used for data structure. However, it has to scan the database twice to build the structure. The first scan identifies all the frequent items, the second goes through the results of the first scan to build the data structure. What is more, a part of the item sets corresponds with the support and these item sets are mined again. This study will look into using these known frequent item sets.

In Section 2, some techniques to avoid excessive temporary data mining duplicating frequent item sets are discussed. In Section 3, a data structure and algorithms for data mining are proposed. Experimental results to evaluate the performance of the algorithm are presented in Section 4. Finally, the feature demonstrated in Section 5.

2. Related works

The FP-Tree is a data structure that is used to optimize the compression of a database. First, the FP-Tree scans the database to get frequencies of all items. Then, these frequencies are sorted in descending order. The purpose of sorting is to put the higher frequencies close to the tree root when building tree structure and thus significantly reducing the degree of tree nodes. Then the database is scanned again sorting all transaction data. The basis for the order of sorting is the frequency of items; infrequent items are removed. According to [1, 2, 3, 5, 6, 7, 10] representing data transactions by using regular tree structures make data compression efficient.

There are two mining approaches are based on the FP-Tree. These can be categorized as the bottom-up approaches [1, 2, 3, 4, 5] and the top-down approaches [5, 10]. The bottom-up approach sorts all frequent k-item sets in ascending order. Then search from the tree nodes to the root to build a candidate sub tree. The sub tree of (k+1)-item sets does not include the k-item sets that are not frequent. The top-down approach sorts all k-item sets in descending order and iterates using the virtual root instead of the k-item set tree structure.

Besides using the compressed structure of the FP-Tree, transaction mapping [6, 7] is also an efficient approach for mining data. This approach first builds an FP-Tree. After building the FP-Tree, an interval list for each item can be done by traveling the FP-Tree. In the mining phase, the interval lists can determine two items are frequent or not by intersection produce. Producing all the patterns is time consuming. The lexicographic tree [6, 7, 8] technique can make it easier.

These approaches perform well in data mining. But, they process items repeatedly. That consumes too many computing resources. In this paper, a new approach to avoid that is presented. In this study, the FP-
Tree structure was improved to store the known frequent item sets and data mining was done using the transaction mapping technique. Unlike the normal transaction mapping, the lexicographic tree for known frequent item sets was built dynamically. This avoids building unnecessary frequent items into the lexicographic tree. The transaction lists were intersected to verify whether an item set was frequent or not. The next section gives the details.

3. Confidence Frequent Pattern Grubbing

In order to avoid wasted a lot of computing resources when verifying a few frequent items. A new algorithm which has three phases (Figure 2) is proposed. The first phase is building a Confidence Frequent Pattern Tree (CFP-Tree). After creating a CFP-Tree, the depth first search is done to build a Confidence Frequent Pattern Header (CFP-Header) and the transaction list is made at the same time. After constructing all structures, a Confidence Frequent Pattern Grubbing procedure is used to mine data. In section 3.1, definitions for the algorithm details are shown in Section 3.2 and a sample shown in Section 3.3.

3.1. Definitions of Terms

Given a set of items \( I = \{I_1, I_2, ..., I_n\} \) in a database \( DB \). Each transaction \( T \) is a subset of \( I \) (\( T \subseteq I \)). An array \( F \) is used to determine the frequency of each item. If a frequency of item is equal or large then the minimum support \( Sup \). This item is frequent. The CFP-Tree is constructed in a descending order of frequency. Each node of CFP-Tree has the same structure \( Node \). There are three properties in a \( Node \) which are \( Node.child \), \( Node.freq \) and \( Node.strat \). The \( Node.child \) stores the sub \( Node \) structure of tree. The \( Node.freq \) stores the frequency and \( Node.start \) considers the first transaction of each section. The CFP-Tree has only a root node in the beginning. A header table \( H \) appends each \( Node \) to the linking list of particular items. A lexicographic tree \( L \) is dynamically built to assist and verify items are frequent or not.

3.2. Confidence Frequent Pattern Algorithm

The CFP-Tree procedure scans the database fully and analyzes the frequencies of all items at first. Second, each transaction is sorted in descending order according to the frequency of items. Each frequent item of current transaction is inserted into the tree. If any item does not exist in the child of node, it creates a new item node and assigns one to the frequency. Otherwise the frequency of child node adds one. When an item is inserted into the tree, a frequent item node can be combined with its parent node if and only if the parent is not root. This new node has the same properties as the current frequent item node and can be put on the level of the parent node. The frequency of the parent node will decrease the current frequency of the new node. After combining, the frequent item node is deleted. The Figure 3 shows the details.

---

**Figure 2. The procedure of algorithm**

3.1. Definitions of Terms

Given a set of items \( I = \{I_1, I_2, ..., I_n\} \) in a database \( DB \). Each transaction \( T \) is a subset of \( I \) (\( T \subseteq I \)). An array \( F \) is used to determine the frequency of each item. If a frequency of item is equal or large then the minimum support \( Sup \). This item is frequent. The CFP-Tree is constructed in a descending order of frequency. Each node of CFP-Tree has the same structure \( Node \). There are three properties in a \( Node \) which are \( Node.child \), \( Node.freq \) and \( Node.strat \). The \( Node.child \) stores the sub \( Node \) structure of tree. The \( Node.freq \) stores the frequency and \( Node.start \) considers the first transaction of each section. The CFP-Tree has only a root node in the beginning. A header table \( H \) appends each \( Node \) to the linking list of particular items. A lexicographic tree \( L \) is dynamically built to assist and verify items are frequent or not.

---

**Figure 3. The CFP-Tree procedure**

All known frequent items are combined in construction. It requires additional data structures and procedures to verify all items. There are two important phases for mining. The first one is using depth first search to build a linking header for each item. The header table is an index structure enabling one to reach an item quickly. The second is assigning a new transaction list for each item. A tree node is comprised of one or many items in CFP-Tree. Each item of this node is
appended to the header table. If sub nodes remain in this node, then the procedure is processed recursively. On the other hand, the next transaction number follows that of the current transaction number. The detailed is described in Figure 4.

**Input:** a CFP-Tree $CT$ and initial value 1
**Output:** a CFP-Header
**Procedure:** The CFP-Header creates as follows.

create a linking structure $H$
call CFP-Header($CT$, 1 )
Procedure CFP-Header($T$, $S$ )

For ( each $C$ in $T$.child )
  For ( each item $i$ in $\{C\}$ )
    append $C$ to $H[i]$
    $C$.start := $S$
    CFP-Header($C$, $S$ )

$S := S + C.freq$

**Figure 4. The CFP-Header procedure**

**Input:** a frequency array $F$ and header $H$
**Output:** association rules
**Procedure:** The CFP-Grub mines as follows.

create a new lexicographic tree $L$
call CFP-Grub($F$, $H$ )
Procedure CFP-Grub($F'$, $H'$ )
create a frequencies array $F''$
create a linking structure $H''$
for ( each $i$ in $F'$ in descending order and $F'[i] \geq Sup$ )
  for ( each $C$ in $H'[i]$ )
    append $\{C\} - i$ to $L$
  append $\{C\} - i$ to $L$
  for ( each $C$ in $H'[i]$ )
    $F''$,$H'' := (\{C$.child $\cap \{H'\}\} - \{H'[i]\})$
    sort $F''$ in descending order
    push $i$ into $P$
    CFP-Grub($F''$, $H''$ )
    verify $P$ and $L$ by intersection
    pop $P$
    if $P$ is empty
    $L := empty$

**Figure 5. The CFP-Grub procedure**

At last, the mining procedure forms a dynamic lexicographic tree and uses intersection to verify items. In the CFP-Tree, each known frequent item set is combined. Only the power set without current verified item still needs to be verified. Each item without the current verified item is dynamically built into the lexicographic tree. This procedure stops when no more items can be mined and verify the last item and lexicographic tree by intersection. The details are given in Figure 5.

### 3.3. Exemplification

Table 1 shows an example of a database. If the support is three and the sorted frequent items are $f(4)$, $c(4)$, $a(3)$, $b(3)$, $m(3)$ and $p(3)$. Each transaction is gradually inserted into the tree. When inserting the last frequent items of transaction. The $f$, $c$ and $a$ are frequent and can be combined to one. A new node $fca$ has the same structure as the current node $a$ and is inserted into the child of the tree root. The transaction number starts from one. When traveling the CFP-Tree, the transaction number is passed into the next recursive procedure. Otherwise, the transaction number increases the current frequency of node. At the same time, each item of a node is appended to the header table. The complete CFP-Tree and CFP-Header are given in Figure 6. Each frequent item is processed in descending order. The item $f$ has the largest support and there are two nodes $fca[1,3]$ and $f[4,1]$. The power sets without $f$ are built into a lexicographic tree (Figure 7) and the header of all sub items is shown in Figure 8. Only the $m(3)$ is frequent in the power set of $f$ so that $m[1,2]$ and $m[3,1]$ are intersected the lexicographic tree. The frequent item set contains $f$ are $fm$, $fc$, $fcm$, $fca$, $fcam$, $fa$ and $fam$ at the finish.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items (Order)</th>
<th>(Order) frequent items</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>$f, a, c, d, g, i, m, p$</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>$b, c, k, p$</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 6. CFP-Header Example**

**Figure 7. The lexicographic tree of item f**

**Figure 8. The sub CFP-Header of item f**
4. Experiments

In order to evaluate the performance of the proposed algorithm, it was developed with Microsoft Visual C++ 8.0 and an IBM association data generator [9] to produce required data. All experiments are implemented using Windows XP SP2 operating system and performed on Pentium Xeon with 3.2GHz and 1 GB RAM.

4.1. Experimental Results

In Figure 9, T10I4N100K was used to compare the execution time. It has ten items per transaction (T10) on average, four of average length of maximal pattern (I4) and 100K of different items (N100K). The support was set at 0.0005. Especially, the execution time of the proposed algorithm dropped significantly when 500K. Because many known frequent item sets are reducing the repetition of calculation greatly.

Figure 9. Different transactions in T10I4N100K

In Figure 10, the total number of items is reduced to 10K (N10K); the supports were 0.005, 0.001, 0.0005 and 0.0001. It was found that the execution time for CFP-Grub had an average of 50% increase in execution time compared to the FP-Growth. When the mining data were of higher complexity, there are more items had to be mined, the CFP-Grub could cope with the number of known frequent items.

Figure 10. Different supports in T20I4D100KN10K

5. Conclusion

In this paper, we proposed an algorithm that uses known frequent item sets to reduce the number of iterations in association data mining. This includes three procedures, namely CFP-Tree, CFP-Header and CFP-Grub. From the experimental results it can be seen that using known frequent item sets avoids unnecessary iterating calculation.

Not many studies have been done in known frequent item sets. We implemented an algorithm to verify that using known frequent item sets can increase performance in data mining. However, building a CFP-Tree and a lexicographic tree takes a lot of time. In future, improvements will be made to avoid bottlenecks in order to get better performance in memory usage and execution time.

References

[4] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," Data Mining and Knowledge Discovery, 2004, Vol. 8, No. 1, pp. 53-87.