An Efficient Graph-Based Approach to Mining Association Rules for Large Databases

Abstract: The task of Data Mining is to find the useful information within the incredible sets of data. One of important research areas of Data Mining is Mining Association Rules. If we can find these relations by mining association rules, we can provide better selling strategy to gain more customers' attentions. However, in some applications, the large itemsets may not always correlate with each other. In this paper, we propose a new graph-based algorithm to discover the association rules. It represents the large itemsets as a graph, which constructs a graph based on \( L_2 \). Then, by dividing the items to several groups, the association rule can be mined efficiently. We conduct several experiments using different synthetic transaction databases. The simulation results show that the GAR algorithm outperforms the FP-Growth algorithm in the execution time for all transaction databases.

Keywords: Association rules; Data mining; Graph-based mining; Large itemsets; Knowledge discovery.

Reference to this paper should be made as follows: ‘An Efficient Graph-Based Approach to Mining Association Rules for Large Databases’, *Int. J. Intelligent Information and Database Systems (IJIDS)*, Vol. x, No. x, pp.xxx-xxx.

Biographical Notes:

1 Introduction

In recent years, with the computers and information industries growing more and more rapidly, varies data around us become more complexity and huge. Commercial behavior, scientific statistics, natural phenomena and DNA projects are some examples to produce lots of data every day. In the past, we may put these data in drawers or databases to extract some information efficiently. However, as the amount of data grows, it becomes very difficult to determine the useful data. Generally speaking, there are four kinds of problems that we have met (Agrawal and Srikant, 1995; Han et al., 2007; Liu et al., 2006; Srikant and Agrawal, 1996; Zaki, 1998). First, there is too much information for us to digest. Second, it is difficult to recognize the reality of the information. Third, it is hard to guarantee the security of the information. Finally, we can not deal with different forms of information easily because of much data. In order to solve these four problems,
people start to think how to find useful knowledge without overwhelming by information flooding. Although recent database management systems can provide quick insertion, deletion, query and statistic, they can not detect the relations and rules between data. So, there comes a research named data mining which is used to help making decisions and information retrieval.

There are many methods for data mining, such as association rule mining, sequential pattern mining, attribute oriented induction, data classification, data clustering, pattern-based similarity search, and data cube (Lee, 2007). Mining association rules is one of techniques used to discover patterns from massive data. The type of data being processed by this technique is usually transaction data. This technology could be used for market analysis, pattern association. Mining association rules mean that the presence of one set of items implies the presence of another set of items (Han et al., 2007; Nanopoulos et al., 2007; Wei et al., 2006).

For example, people who purchased hammers also purchased nails.

The tasks of finding association rules in large databases will bring lots of benefits for many applications, such as bioinformatics, market-basket analysis, customer relationship management (Kamath, 2001), etc. There are several Apriori-based algorithms, such as AprioriAll (Agrawal and Srikant, 1995) and GSP (Srikant and Agrawal, 1996), have been proposed. However, their search space is extremely large. For example, with m attributes, there are \( O(m^k) \) potentially large itemsets of length \( k \). With millions of objects in the database, the problem of I/O minimization becomes paramount. Moreover, these algorithms are iterative in nature. Therefore, they need to scan the whole database many times, which is obviously very expensive in I/O cost (Zaki, 1998). Unlike Apriori, the FP-Growth algorithm (Han and Fu, 2004) is not Apriori-like restricted generation-and-test but restricted test only.

However, the large itemsets may not always correlate with each other. This property appears in several real-word cases, e.g., in bioinformatic, e-commerce applications, and bibliographic analysis. For example, the click streams that people visit the portal websites could be divided into different groups. For another example, for women and men purchasing goods in the different floors of the department store, it shows that the large itemsets of mining transactions will be separated into different groups. In such cases, the items will be partitioned into smaller groups consisting of items with significant correlations between them, whereas there will be few correlations between items of different groups (Nanopoulos et al., 2007). Yen and Chen have proposed the DLG algorithm (Yen and Chen, 2001), which uses the large 2-itemsets to construct an association graph. After the DLG algorithm, an algorithm based on a complete sub-graph is put forward (Liu et al., 2006). However, both algorithms follow a simplistic implementation of the graph. This simplistic approach is possible only for domains with a small size. To avoid the aforementioned inadequacies, Nanopoulos et al. (2007) do not follow the direct approach to partition the domain into groups. They do not find the support of all candidate 2-itemsets. Nevertheless, they scan the dataset twice to get the large 2-itemsets. Therefore, the graph will contain fewer edges than that constructed by the DLG algorithm, and it will fit in main memory. However, if it is possible to have data with partitions that are not clearly separated, the approach will consider this case as a noise. Moreover, it will scan the dataset at least three times.

In this paper, to avoid the disadvantages, we propose a new graph-based algorithm to discover the association rules. It represents the large itemsets as a graph,
which constructs a graph based on $L_2$. Then, by dividing the items to several
groups, the association rule can be mined efficiently. Finally, we conduct several
experiments using different synthetic datasets. These simulation results show that
the proposed algorithm outperforms the FP-Growth algorithm in all datasets. The
main reason is that our algorithm could generate the densely GFP-Tree than the
FP-Growth algorithm.

The rest of the paper is organized as follow: Section 2 gives a survey of several
well-known data mining algorithms for association rules. In Section 3, we present
a new graph-based algorithm for mining association rules. In Section 4, we give
a comparison of the performance of the FP-Growth algorithm and our proposed
algorithm. In Section 5, we give a summary and point out some future research
directions.

2 Background

In this section, we give a survey of some well-known data mining algorithms
for association rules. The formal definitions of the problem of mining association
rules are proposed by Agrawal and Srikant (Agrawal and Srikant, 1995; Srikant and
Agrawal, 1996). Agrawal and Srikant proposed the Apriori algorithm (Agrawal and
Srikant, 1994) for mining association rules. This algorithm generates the candidate
itemsets to be counted in an iteration by using only the itemsets found large in the
previous iteration. This reduces the number of candidate itemsets. However, for
each candidate itemset, it needs to count its appearances in all transactions. In this
algorithm, each iteration requires one pass over the database and it wastes a lot
of time. Han proposed the FP-Growth algorithm (Han and Fu, 2004) for mining
association rules. Unlike Apriori, the FP-Growth algorithm is not Apriori-like
restricted generation-and-test but restricted test only. First, a data structure, called
the FP-Tree, is constructed, which is an extended prefix-tree structure storing cru-
cial, quantitative information about large itemsets. Only $L_1$ will have nodes in the
tree, and the tree nodes are arranged in such a way that more frequently occurring
nodes will have better chances of sharing nodes than less frequently occurring ones.
Second, an FP-Tree-based pattern fragment growth mining method, is developed,
which starts from $L_1$, examines only its conditional pattern base, constructs its
conditional FP-Tree, and performs mining recursively with such a tree. Based on
the algorithm, for the sample shown in Table 1 with the minimum support = 20%.
Figure 1 shows the FP-Tree.

3 The Graph-Based Approach

As described before, the large itemsets may be divided into some independent
groups. In this section, we present the GAR algorithm. It represents the large
itemsets as a graph, which constructs a graph based on $L_2$. Based on the property
of the graph, we partition the graph into different sub-graphs, which results in a
case that the processing time of mining association rules can be reduced.
Table 1  A transaction database as example

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ABDE</td>
</tr>
<tr>
<td>2</td>
<td>AJL</td>
</tr>
<tr>
<td>3</td>
<td>DEFKL</td>
</tr>
<tr>
<td>4</td>
<td>BCDJ</td>
</tr>
<tr>
<td>5</td>
<td>CKL</td>
</tr>
<tr>
<td>6</td>
<td>ABCD</td>
</tr>
<tr>
<td>7</td>
<td>DEI</td>
</tr>
<tr>
<td>8</td>
<td>BCDFHG</td>
</tr>
<tr>
<td>9</td>
<td>JK</td>
</tr>
<tr>
<td>10</td>
<td>FGHIJ</td>
</tr>
</tbody>
</table>

Figure 1  The FP-Tree
3.1 The Graphic Representation

We give a simple example to illustrate how to denote large itemsets by using the graphic representation. Take Table 1 as an example. In the first iteration, we scan all the transactions to count the number of occurrences for 1-itemsets and 2-itemsets. Assume that the minimum transaction support is two, the set of large 1-itemsets, $L_1$, and 2-itemsets, $L_2$, can then be determined. Then, we let each item as a node in the graph, and draw a line from node $A$ to node $B$, if itemset $\{A, B\}$ is a large 2-itemset. The graphic representation is shown as Figure 2.

As observed in (Agrawal and Srikant, 1994), any subset of a large itemset must be a large itemset by itself. That is, $\{A, B, C\} \in L_3$ implies $\{A, B\} \in L_2$, $\{A, C\} \in L_2$, and $\{B, C\} \in L_2$. Based on this observation, if $\{A, B, C\} \in L_3$, the degree of node $A$ must be greater than 2 or equal to 2. So, we can discard the nodes whose degree is less than 2. For example, the degree of node $E$ is 1, and the degree of node $K$ is 1. These nodes can not appear in $L_3$. After pruning, the graph is shown as Figure 3.

Next, we use the cut-point property of the graph to divide a graph into subgraphs. A connected undirected graph is biconnected, if there are no nodes whose removal disconnects the rest of the graph. If a graph is not biconnected, the nodes whose removal would disconnect the graph are known as cut points (Weiss, 1993). The depth-first search provides a linear-time algorithm to find all cut points in a connected graph. Doing all the computation takes $O(|L_1| + |L_2|)$ time. First, starting at any node, we perform a depth-first search and number the nodes as they are visited. For each node $V$, we call this preorder number $V.Num$. Then, for every node $V$ in the depth-first search spanning tree, we compute the lowest-numbered node, which we call $V.Low$, that is reachable from node $V$ by taking zero or more tree edges and then possibly one back edge (Weiss, 1993). The FindCutPoint Procedure is shown as follows:
Figure 3  The pruned graph

![Pruned Graph Diagram]

Procedure FindCutPoint(v);
begin
  v.visited=true;
  v.low=v.num=counter++;
for all node w adjacent to v do
  if (w.visited=false) then
    begin
      w.parent=v;
      FindCutPoint(w);
      if (w.low ≥ v.num)
        add v to cutpointlist
      v.low=min(v.low,w.low);
    end;
  else
    if (v.parent ≠ w) then
      v.low=min(v.low,w.num);
end;

For example, the graph in Figure 3 is not biconnected: node D and node F are articulation points. The removal of node D would disconnect nodes A, B and C, from the rest of the graph. After grouping, the graph is shown as Figure 4.

3.2 The Algorithm

The GAR Algorithm is shown as follows. We use the same example, which is given in Table 1 with the minimum support = 20%, to go through the algorithm. First, we call procedure GenerateL1L2 to generate L1 and L2 by scanning database once. We count 1-itemsets in an array. In the same time, we count 2-itemsets in
a sparse matrix, in which only the lower triangular part is used. Then, we check whether the minimum support requirement is met. In this example, the large 2-itemsets are $AB, AD, BC, BD, CD, DE, DF, FG, FH, GH$ and $KL$.

**Procedure GAR:**

```
begin
GenerateL1L2();
ConstructGraph();
CutGraph();
ConstructGFP-Forest();
forall GFP-Tree $gt \in$ GFP-Forest do
    Trace GFP-Tree $gt$;
end;
```

**Procedure GenerateL1L2:**

```
begin
forall transactions $t \in D$ do
    begin
     count 1-itemsets in an array;
     forall 2-subsets $x$ of $t$
        count 2-itemsets in a matrix;
    end;
    $L_1 :=$ large 1-itemsets;
    $L_2 :=$ large 2-itemsets;
end;
```

Next, based on large 2-itemsets, we call procedure ConstructGraph, as follows, to construct the graph. We increase the degree of the nodes which are the items of large 2-itemsets and assign the different group number to the different graphs. The result of the example is shown as Figure 2. Then, we call procedure CutGraph, as follows, to divide the graphs into sub-graphs. For each large 1-itemset, we check the degree of node. If the degree of the node is less than 2, the node (item) would be pruned. For example, the degree of node $E$ is 1. The node $E$ will be pruned from the graph. Then, we call the procedure FindCutPoint to find all cut points.
For example, the node $D$ is the cut-point. So, we can separate the graph into sub-graphs. The result is shown in Figure 4.

**Procedure** ConstructGraph;  
begin  
forall 2-itemsets $\{x, y\} \in L_2$ do  
begin  
x.degree++;  
y.degree++;  
x.groupnumber = y.groupnumber  
end;  
end;  

**Procedure** CutGraph;  
begin  
forall 1-itemsets $x \in L_1$ do  
if $x.degree < 2$ then;  
prune($x$);  
forall group $g \in graphs$ do  
begin  
FindCutPoint($v$);  
end;  
end;  

Then, we call the procedure ConstructGFP-Forest, as follows, to construct the GFP-Forest using the similar structure of the FP-tree. Based on sub-groups, we build the rearrangement arrays and transform the database $D$ to database $D'$, as shown in Table 2. Unlike FP-Growth algorithm, which rearrange the items based on the support values of large 1-itemsets, we build the order based on the degree of the nodes. For example, the original items are $\{ABDE\}$ for $TID = 1$. Because node $E$ is not in the graph, item $E$ will be pruned. Then, recasting the remaining items so that each record is ordered according to the ordering of degree of the nodes, the items are transformed to $\{DBA\}$ for $TID = 1$. Next, the transformed itemsets have been inserted into the GFP-Forest, as shown in 5. Finally, we trace the GFP-Forest and get the large itemsets, $\{A,B,D\}$, $\{B,C,D\}$ and $\{F,G,H\}$. The total number of nodes in the GFP-Forest is 10, whereas the total number of nodes in the FP-tree is 27, which represents a reduction ratio of 2.7.

**Procedure** ConstructGFP-Forest;  
begin  
Build rearrangement arrays;  
forall transactions $t \in D$ do  
begin  
prune the items which are not in graph;  
Recast the items;  
Add to GFP-Forest;  
end;  
end;  


Table 2 The projected transaction database

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
<th>sub-transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ABDE</td>
<td>DBA(1)</td>
</tr>
<tr>
<td>2</td>
<td>AJL</td>
<td>A(1)</td>
</tr>
<tr>
<td>3</td>
<td>DEFKL</td>
<td>D(1), F(1)</td>
</tr>
<tr>
<td>4</td>
<td>BCDJ</td>
<td>DBC(1)</td>
</tr>
<tr>
<td>5</td>
<td>CKL</td>
<td>C(1)</td>
</tr>
<tr>
<td>6</td>
<td>ABCD</td>
<td>DBCA(1)</td>
</tr>
<tr>
<td>7</td>
<td>DEI</td>
<td>D(1)</td>
</tr>
<tr>
<td>8</td>
<td>BCDHG</td>
<td>DBC(1), FGH(2)</td>
</tr>
<tr>
<td>9</td>
<td>JK</td>
<td>(null)</td>
</tr>
<tr>
<td>10</td>
<td>FGHIJ</td>
<td>FGH(2)</td>
</tr>
</tbody>
</table>

Figure 5 The GFP-forest

4 Performance

In this Section, we study the performance of the proposed algorithm. In this paper, we only make a comparison of our proposed algorithm with the FP-growth algorithm. Our experiments were performed on a Pentium 4 personal computer with one CPU clock rate of 1.5 GHz, 512 MB of main memory, running Windows XP Professional, and coded in Java. The data was stored on a local 40G IDE 3.5” drive.

4.1 Generation of Synthetic Data

We generated synthetic transactions to evaluate the performance of the algorithms over a large range of data characteristics. The synthetic data is said to
Table 3  Parameters

| $|D|$ | Number of transactions |
|---|---|
| $|T|$ | Average size of transactions |
| $|MT|$ | Maximum size of the transactions |
| $|L|$ | Average size of maximal potentially large itemsets |
| $|ML|$ | Maximum size of the potentially large itemsets |
| $|L|$ | Number of maximal potentially large itemsets |
| $N$ | Number of items |

simulate a customer buying pattern in a retail environment. The parameters used in the generation of the synthetic data are shown in Table 3. The length of a transaction is determined by a Poisson distribution with mean $\mu$ equal to $|T|$. The transaction is repeatedly assigned items from a set of potentially maximal large itemsets $\mathcal{F}$, until the length of the transaction does not exceed the generated length (Agrawal and Srikant, 1994, 1995).

The length of an itemset in $\mathcal{F}$ is determined according to a Poisson distribution with mean $\mu$ equal to $|L|$. The size of each potentially large itemset is between 1 and $|ML|$. Items in the first itemset are chosen randomly from the set of items. To model the phenomenon that large itemsets often have common items, some fraction of items in subsequent itemsets are chosen from the previous itemset generated. We use an exponentially distributed random variable with mean equal to the correlation level to decide this fraction for each itemset. The remaining items are picked at random. In the datasets used in the experiments, the correlation level was set to 0.5. Each itemset in $\mathcal{F}$ has an associated weight that determines the probability that this itemset will be picked. The weight is picked from an exponential distribution with mean equal to 1. The weights are normalized such that the sum of all weights equals 1. For each transaction, we generate a random real number which is between 0 and 1 to determine the ID of the potentially large itemset. To model the phenomenon that all the items in a large itemset are not always bought together, we assign each itemset in $\mathcal{F}$ a corruption level $c$. When adding an itemset to a transaction, we keep dropping an item from the itemset as long as a uniformly distributed random number (between 0 and 1) is less than $c$. The corruption level for an itemset is fixed and is obtained from a normal distribution with mean $= 0.5$ and variance $= 0.1$. Each transaction is stored in a file system with the form of <$\text{transaction identifier, item}>$.

Some different data sets were used for performance comparison. Table 4 shows the names and parameter settings for each data set. For all data sets, $N$ was set to 1,000 and $|L|$ was set to 2,000.

4.2 Simulation Results

In our simulation result, the performance measures which we concern about are the CPU execution time of the computation and the sensitivity to parameters. When we choose four cases as the synthetic dataset and with varied minimum supports between 0.1% and 1.2%, the comparisons of the execution time of the
Table 4  Parameter values for synthetic datasets

<table>
<thead>
<tr>
<th>Case</th>
<th>Name</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T5.I2.D20K</td>
<td>5</td>
<td>10</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>T10.16.D50K</td>
<td>10</td>
<td>15</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>T5.I2.D200K</td>
<td>5</td>
<td>10</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>T10.16.500K</td>
<td>10</td>
<td>15</td>
<td>6</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 6  The relationship between the execution time (seconds) and the minimum support (Case 1)

FP-Growth and GAR algorithms are shown in Figure 6, Figure 7, Figure 8 and Figure 9. And the memory that are needed for the GAR algorithm is used to store the sparse matrix. In our experiments, N was set to 1,000. Hence, 500,000 integers (about 2MB) are needed to store the matrix.

In Figure 6, Figure 7, Figure 8 and Figure 9, we show that the execution time of the GAR algorithm is always less than that of the FP-Growth algorithm. Both of the GAR and FP-Growth algorithms scan the database only twice, so the execution time of disk I/O in both algorithms is the same. However, the GAR algorithm constructs the GFP-Forest structure is better than the FP-Tree structure of the FP-Growth algorithm in terms of the storage space.

The larger the minimum support value is, the smaller the number of valid itemsets is; thus the space of the storage is less, too. Moreover, the structure of the GFP-Forest is closer than that of the FP-Tree, so we can say that the larger the minimum support value is, the heavier the reduction in the space of the storage is. The comparisons of the storage space of the FP-Growth and GAR algorithms are shown in Figure 10, Figure 11, Figure 12 and Figure 13. Although the GAR algorithm takes more time to find the cut-points, the number of nodes is less than that of FP-Growth algorithm. Furthermore, it represents a reduction ratio of 2 with the minimum support = 1.2%. Therefore, the total processing time and storage space are less than those of FP-Growth algorithm.
Figure 7  The relationship between the execution time (seconds) and the minimum support (Case 2)

Figure 8  The relationship between the execution time (seconds) and the minimum support (Case 3)

Figure 9  The relationship between the execution time (seconds) and the minimum support (Case 4)
Figure 10   The relationship between the storage space (the number of nodes) and the minimum support (Case 1)

Figure 11   The relationship between the storage space (the number of nodes) and the minimum support (Case 2)

Figure 12   The relationship between the storage space (the number of nodes) and the minimum support (Case 3)
Figure 13 The relationship between the storage space (the number of nodes) and the minimum support (Case 4)

5 Conclusion

In this paper, we focus on the cases where the large itemsets may not correlate with each other and propose the GAR algorithm. The large 2-itemsets have been construct an association graph. Based on the cut-point property of the graph, we partition the items into different groups, which results in a case that the processing time of mining association rules can be reduced. We have presented the proposed graph-based algorithm for mining association rules and have conducted several experiments using different synthetic datasets. The simulation results have shown that our algorithm could provide better performance than the FP-Growth algorithm in terms of the processing time.

References


Han, J. and Fu, Y. (2004), ‘Mining Frequent Patterns Without Candidate Generation: A Frequent-Pattern Tree Approach’, Data Mining and Knowledge Discovery, Vol. 8, No. 1, pp. 53–87.


