Efficient Frequent Pattern Mining Based on a Condensed Tree Structure

Byung Joon Park¹, Sang Young Kim¹, and Lynn Choi²

¹ Department of Computer Science, Kwangwoon University, Seoul, Korea
{bjpark, sykim}@kw.ac.kr
² School of Electrical Engineering, Korea University, Seoul, Korea
lchoi@korea.ac.kr

Abstract. In this paper, we present an efficient tree structure and its associated algorithm for discovery of frequent patterns from a large data set. We demonstrate the effectiveness of our algorithm and performance improvement over the existing approach CATS which is one of the fastest frequent pattern mining algorithms known to date.

Keywords: frequent pattern discovery, tree structure

1 Introduction

Discovery of frequent patterns from a huge collection of data has been a topic of active research in the information processing community. Several tree structures have been devised to represent the input data set for efficient pattern discovery[1][2]. One of the fastest frequent pattern mining algorithms known to date is the CATS algorithm, which can efficiently represent the whole data set and allow mining with a single scan over the database[3]. This paper describes our work on improvement over the original CATS approach in terms of memory usage and processing time. The proposed tree structure allows insertion or deletion of transactions at any time like CATS, but usually results in more condensed types of tree enabling a more efficient pattern discovery process.

2 The Proposed Algorithm for a Conditional Condensed Tree

It is the process of constructing a conditional condensed CATS-tree that is the main focus of our work. Like CATS, the overall mining process proceeds in three phases:

Step 1: Convert the CATS tree generated from a database scan into a condensed tree with nodes having the frequency count less than the minimum support removed.

Step 2: Construct conditional condensed CATS-trees (i.e., as alpha-trees) for items in the header table with frequency counts greater than the minimum support.
**Step 3:** For each alpha-tree generated in step 2, item sets with at least minimum support are mined.

Our algorithm differs from CATS-FELINE in step 2. Instead of recursively constructing alpha trees for each frequent item set, our algorithm generates a single conditional condensed tree for each item using a pre-order traversal of the original CATS-tree. To illustrate the basic idea behind the algorithm, we will use the following database as an example (same as the sample database in [3]):

**Table 1: Sample database**

<table>
<thead>
<tr>
<th>TID</th>
<th>Original Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F, A, C, D, G, I, M, P</td>
</tr>
<tr>
<td>2</td>
<td>A, B, C, F, L, M, O</td>
</tr>
<tr>
<td>3</td>
<td>B, F, H, J, O</td>
</tr>
<tr>
<td>4</td>
<td>B, C, K, S, P</td>
</tr>
<tr>
<td>5</td>
<td>A, F, C, E, L, P, M, N</td>
</tr>
</tbody>
</table>

Given the above database, the original CATS-tree constructed from a database scan and its condensed one will look like the following (assuming minimum support of 3):

![Figure 1: CATS-tree and its condensed one](image)

This condensed tree, a header table containing all the frequency counts for each item, and the required minimum support will be the actual input to our algorithm called FPM(Frequent Patterns Merge). Given the above condensed tree, FPM starts building an alpha tree for each frequent item (*i.e.* alpha item) as follows:

![Figure 2: Successive rounds of constructing alpha tree for c](image)

1. If the root item of the condensed tree is frequent, either insert the root into the current alpha tree at proper position (if not present already) or update the existing count of the root by adding its new count.
2. Otherwise, either insert the root into the current alpha tree as a child of the root (if not present already) or update the existing count of the root by adding its new count.
3. Next, for each subtree of the condensed tree, do the same process recursively.

3 Comparison with CATS-FELINE

Given the same database used in Section 3, alpha trees constructed by CATS-FELINE and ours will be different as shown in Figure 3. The difference is due to the way how the infrequent items are dealt with. As can be seen in Figure 3, CATS-FELINE keeps many separate nodes (P:2 and P:1) for infrequent items such as P although they share the same alpha node. Hence, it needs more memory space for storing infrequent items. However, FPM results in a more condensed alpha tree in most cases since separate infrequent items are collapsed into single child nodes of the alpha item (root).

Figure 3: Different types of alpha tree

4 Conclusion

In this paper, we described an efficient tree structure for representing a conditional condensed tree (alpha tree) for a frequent item in transactions. We demonstrated the performance improvement of FPM over CATS-FELINE by comparing them using various minimum support values and different sizes of databases. A considerable performance improvement over CATS in terms of memory usage and processing time has been achieved by our proposed tree structure.

References