Recursion Pruning for the Apriori Algorithm

Christian Borgelt
Department of Knowledge Processing and Language Engineering
School of Computer Science, Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany
Email: borgelt@iws.cs.uni-magdeburg.de

Abstract

Implementations of the well-known Apriori algorithm for finding frequent item sets and associations rules usually rely on a doubly recursive scheme to count the subsets of a given transaction. This process can be accelerated if the recursion is restricted to those parts of the tree structure that hold the item set counters whose values are to be determined in the current pass (i.e., contain a path to the currently deepest level). In the implementation described here this is achieved by marking the active parts every time a new level is added.

1. Introduction

The implementation of the Apriori algorithm described in [2] uses a prefix tree to store the counters for the different item sets. This tree is grown top-down level by level, pruning those branches that cannot contain a frequent item set. This tree also makes counting efficient, because it becomes a simple doubly recursive procedure: To process a transaction for a node of the tree, (1) go to the child corresponding to the first item in the transaction and process the rest of the transaction recursively for that child and (2) discard the first item of the transaction and process it recursively for the node itself (of course, the second recursion is more easily implemented as a simple loop through the transaction). In a node on the currently added level, however, we increment a counter instead of proceeding to a child node. In this way on the current level all counters for item sets that are part of a transaction are properly incremented.

2. Recursion Pruning

Since the goal of the recursive counting is to determine the values of the counters in the currently deepest level of the tree (the one added in the current pass through the data), one can restrict the recursion to those nodes of the tree that have a descendant on the currently deepest level. Visiting other nodes is not necessary, since no changes are made to these nodes or any of their descendants — only the nodes in the currently deepest level of the tree are changed.

To implement this idea, which I got aware of at FIMI 2003, either from the presentation by F. Bodon [1] or from a subsequent discussion with B. Goethals, I added markers to each node of the prefix tree, which indicate whether the node has a descendant on the currently deepest level. Fortunately only one bit is necessary for such a marker, which could be incorporated into an already existing field, so that the memory usage is unaffected.

These markers are updated each time a new level is added to the tree, using a recursive traversal, which marks all nodes that have only marked children. New nodes are, of course, unmarked, and nodes on the previously deepest level that did not receive any children are marked to seed the recursion. Note that the recursion can exploit the markers set in previous passes, because a node that did not have a descendant on the deepest level in the previous pass cannot acquire a descendant on the currently deepest level.

Of course, the other pruning methods for the counting process described in [2] are applied as well.

3. Experimental Results

I ran experiments on the same five data sets I already used in [2], relying on the same machine and operating system, though updated to a newer version (an AMD Athlon XP 2000+ machine with 756 MB main memory running S.u.S.E. Linux 9.1 and gcc version 3.3.3). Strangely enough, however, the new versions of the operating system or the compiler lead to longer(!) execution times for an identical program, an effect that seems to be a nasty recurring feature of the S.u.S.E. Linux distribution. Therefore the experiments were repeated with the old program version to get comparable results.

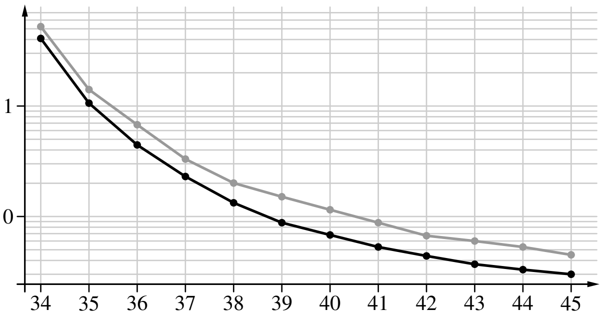

Figure 1. Results on BMS-Webview-1

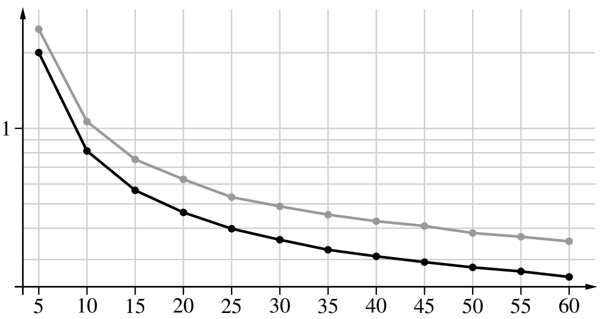

Figure 2. Results on T10I4D100K

The results for these data sets are shown in Figures 1 to 5. The horizontal axis shows the minimal support of an item set (number of transactions), the vertical axis the decimal logarithm of the execution time in seconds. Each diagram shows as grey and black lines the time without and with recursion pruning, respectively.

As can be seen from these figures, recursion pruning can lead to significant improvements on some data set. (Note that the vertical scale is logarithmic, so that the 20-40% reduction, which results for webview1, for example, appears to be smaller than it actually is.) For census, chess, and mushroom, however, the gains are negligible.

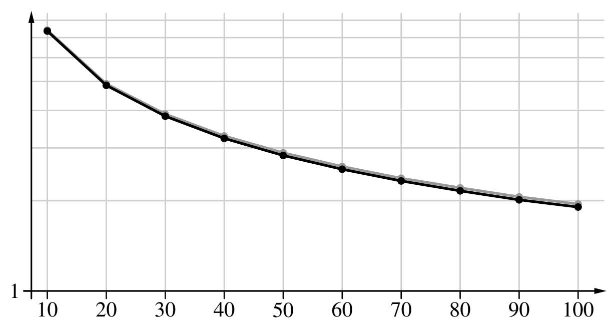

Figure 3. Results on census

Figure 4. Results on chess

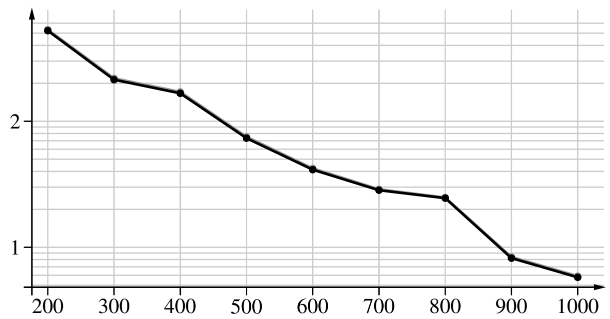

Figure 5. Results on mushroom

4. Programs

The implementation of the Apriori algorithm described in this paper (Windows™ and Linux™ executables as well as the source code) can be downloaded free of charge at

http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html

The special program version submitted to the workshop uses the default parameter setting of this program.

References

[1] F. Bodon. A Fast Apriori Implementation. *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003. http://www.ceur-ws.org/Vol-90/

[2] C. Borgelt. Efficient Implementations of Apriori and Eclat. *Proc. 1st IEEE ICDM Workshop on Frequent Item Set Mining Implementations (FIMI 2003, Melbourne, FL)*. CEUR Workshop Proceedings 90, Aachen, Germany 2003. http://www.ceur-ws.org/Vol-90/