Naju G. Mancheril
naju@cmu.edu

This document can be found online at `http://www.andrew.cmu.edu/user/ngm/15-721/summaries/`

Fast Algorithms for Mining Association Rules

Agrawal, Srikant

This is a very long and complicated paper about taking a set of transactions (what the paper calls *basket data*) and finding association rules in them. For example, a marketing firm might want to ask "What percentage of people who bought X also bought Y?" Another question might be "What two items are most popular amoung people between ages 18 and 25."

The naive solution would be to do an exhaustive search across all possible subsets of items and count how many satisfy the predicate conditions we are looking for. This approach, although it would be efficient space-wise (only store the combinations we need) would waste a lot of time (creating all possible combinations). This paper presents a few algorithms that start with a *seed* itemset (one that already satisfies the boolean predicates we wish to evaluate) and grow them into itemsets of maximal size.

Not all predicates operations can be handled this way. The two that this paper looks at are *confidence* and *support*.

- Given itemsets X and Y, an *association rule* $X \Longrightarrow Y$ has confidence c if $c\%$ of transactions in our transaction database that contain X also contain Y.
- Given itemsets X and Y, an association rule $X \Longrightarrow Y$ has support s if $s\%$ of transactions in our transaction database that contain $X \cup Y$.

Their algorithm tries to find all association rules that have some minimal support and some minimal confidence. They do this by first finding all association rules that have minimum support (thereby cutting down the space of association rules to check). They then examine these itemsets for those that have minimum support by applying Bayes Rule:

$$\text{The association rule } X \Longrightarrow Y \text{ has confidence } = \frac{support(Y)}{support(X)}$$

Questions:

Q1. What is a *large* itemset? Explain how Apriori improves on the performance of AIS in generating large subsets, and give an intuitive argument about why this idea will generate all valid large subsets.

A large itemset $\{i_1, i_2, \ldots, i_k\}$ is one that has minimal support. That is, at least $s\%$ of all transactions in our transaction database were transactions on items $\{i_1, i_2, \ldots, i_k\}$. AIS finds all such itemsets largely by exhaustive search (checking all possible combinations of items).

Apriori has a candidate generation step that generates k-itemsets by joins on $(k-1)$-itemsets. There is an additional *prune* step that removes k-itemsets that have a $(k-1)$-itemset that does not have minimal support. Why does this work?

It seems that if a transaction T contains the k-itemset in question, then it must contain every subset of that itemset as well. This is why the pruning step does not remove any large k-itemset. Furthermore, since we are going with the assumption that the itemsets can be lexiographically ordered, it seems that we should be able to inductively show that all large k-itemsets can be constructed through a join by taking their largest two elements and creating $(k-1)$-itemsets from them.

Q2. Apriori requires many set operations. What kind of data structures does it use for verifying membership and discovering subsets? How are these data structures modified in AprioriTID? In your opinion, are these good/efficient choices?

Apriori uses a *hash-tree*, a tree where each node is a hash table, to store the itemsets. The tree is traversed at depth d by applying a hash function to item d in our itemset. The result of the hash function tells us which child pointer to take. The leaves store lists of itemset that were discovered by our algorithm. All nodes are initially leaf nodes, but when the number of sets in a leaf node grows large, it is converted into an interior node.

Furthermore, to see whether an itemset I is contained within a transaction T, we use a bitmap that contains all items in T. $I \subseteq items(T)$ iff

$$bitmap(I) \ \& \ bitmap(items(T)) == bitmap(I)$$

AprioriTID tries to reduce the number of database reads. I guess this is important if we have a lot of memory and disk I/O is really slow. It could also be useful if we don't want to lock the database with a read lock for too long. It instead stores candidate itemsets with the transactions they were found in: $\langle TID, \{X_k\}\rangle$.

To make this more space efficient, they assign an ID number to the itemset instead, storing $\langle TID, ID\rangle$. Unfortuneately, since they don't want to keep looking up which items are in the set identified by ID, they store *generators* and *extensions*. That is, the sets that were joined to make this itemset (generators) and the itemsets that were created by a join on this itemset (extensions).

I'm not sure whether this is a good choice. The authors think that this hash-ID structure cuts down on the amount of scratch space required by this algorithm, but it only seems to pay off if the original items we were tracking are very heavy. But what database stores so many heavyweight objects? This is especially true in the "market" conditions they described where every item is simply a barcode. It seems like this itemset-to-ID conversion (if we want unique IDs) quickly create numbers that many bits long.

Q3. List two pros and two cons of this paper.

Pros:

- The authors spare no expense describing their data structures. To often, it is assumed that the reader will sit down and figure out many of the data structure details on their own.
- The authors compare their algorithm against the competition using both real and synthesized data. This is important in showing that the algorithm behaves well under the data skew found in real-world databases.

Cons:

- The authors use very confusing notation, constantly switching their dialogue among sets, itemsets, subsets, candidate sets, large sets, etc. For someone who has not seen this information before, it would have been invaluable to see a definition list in the front of the paper that formally defines each of these terms.
- It might not have been a bad idea to separate the data structure descriptions from the algorithm itself. The authors could have instread given the running bounds of their data structures and simply used them to analyze the algorithm's overall performance. Then, they could describe the data structures at liesure in an appendix. This would make the paper considerably less confusing since the reader can focus on their algorithm details without being distracted by hashing and tree traversals.