Project #1 report

1. **The data structure and the implement of the project.**

   This project is written in Java.

   The first step is to use "readData" method to scan the whole input file and read the data into the memory. It's stored as a <List<List<String>>. It also scans the database the second time to get the frequency of each item.

   The second step is frequent set mining, which is implemented in "build" method. This step is done using both projected data base(PDB) algorithm and FPtree to store the data base in each step of projection. My code implements this PDB algorithm recursively shown below. Here pet means projected enumerate tree.

```
pet( database, tree node){
    Scan the projected database to get the candidates(contains items and count);
    and append them to the node.children
    and also store the name and the support in a global List.
    for each child : node.children{
        Get project the database( pdb) specified by each child;
        Recursively call pet(pdb, child);
}
```

The data base is still stored as <List<List<String>>, but the projection of the data base is done with the help of FPtree as follows. In the pet function above, first an FPtree is created based on the database and the headers of the FPtree are exactly those candidates. Then in order to get the projected database corresponding to child, just go from the header with the same name as the child and access all the leaves with the same name as the header. At each leaf, before going to the next leaf, it traverses back to the FPtree root to get a list of items, which is a transaction in the projected database. The amount of this transaction is the same as the count stored in the leaf. In this way, only the lexicographically greater items can be added to the new projected database (because the tree was created based on lexicographic order), and the irrelevant items can be automatically eliminated because it cannot be accessible by the headers.

The third step is association rule generation, which is implemented in ARbuild. This algorithm follows the same one one the book, the Apriori-algorithm.

```
For each frequent item set FIS:
        With k being the size of the consequence set size
        Candidates of size 1 sets H1 is created:
        For k = 1 : FIS.size()
                For each candidates calculated the support
```

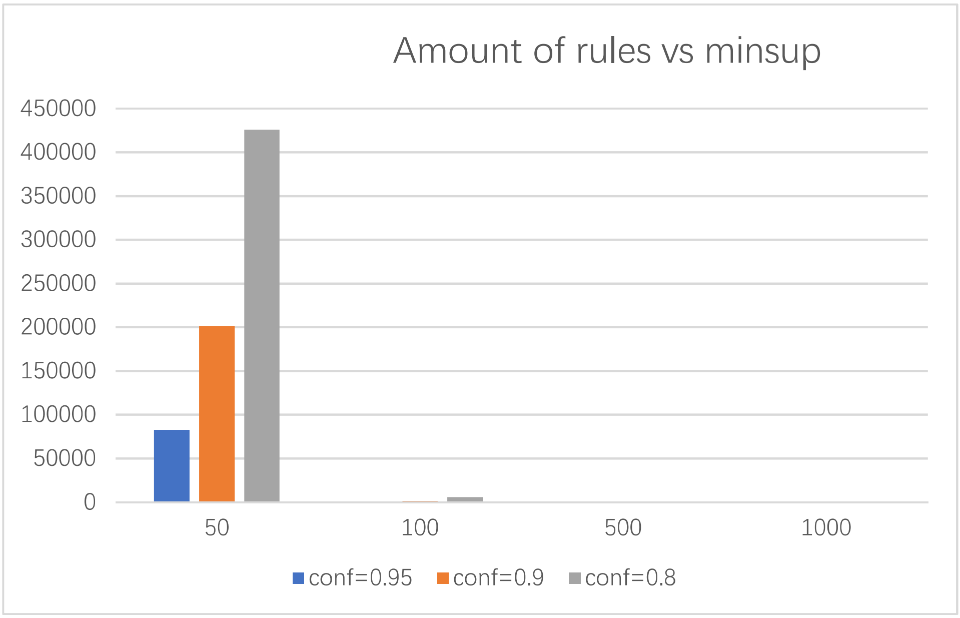
```
            Decided whether to prune
            Generate candidate k+1 sets
        End for
    End for
```

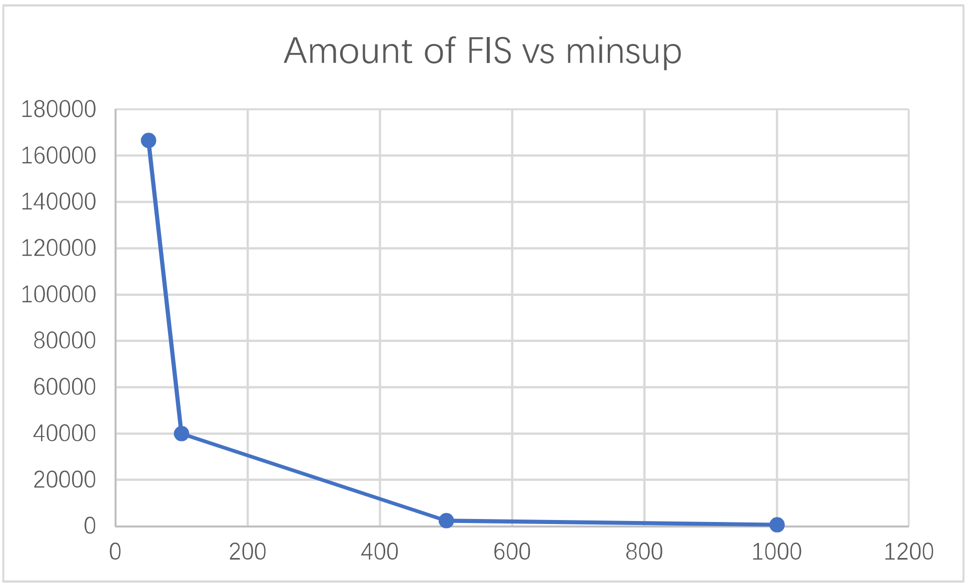
First step traverses the database twice thus has time complexity O(2N), the second step builds a tree with the same number of nodes as the number of the generated frequent item sets. So it's equivalent to traversing this tree as far as the time complexity is concerned. Plus at each node, it also traverses the projected database and builds the FPtree. So the time complexity is around O(M)*O(M) = O(M^2). M is the number of frequent item set. The last step has time complexity O(M)*O(K), where K is the number of association rules.

2. **Experiment results:**

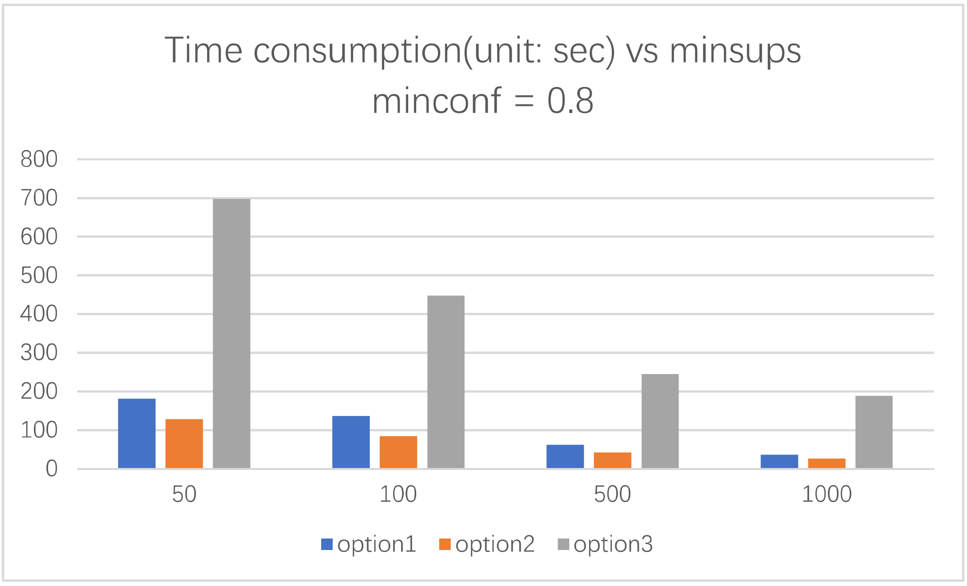
   Sorry the memory on my PC is too small for minsups of 15, 20, 30 and I wasn't able to get access to the computer in the CS department. So I have only the data below.

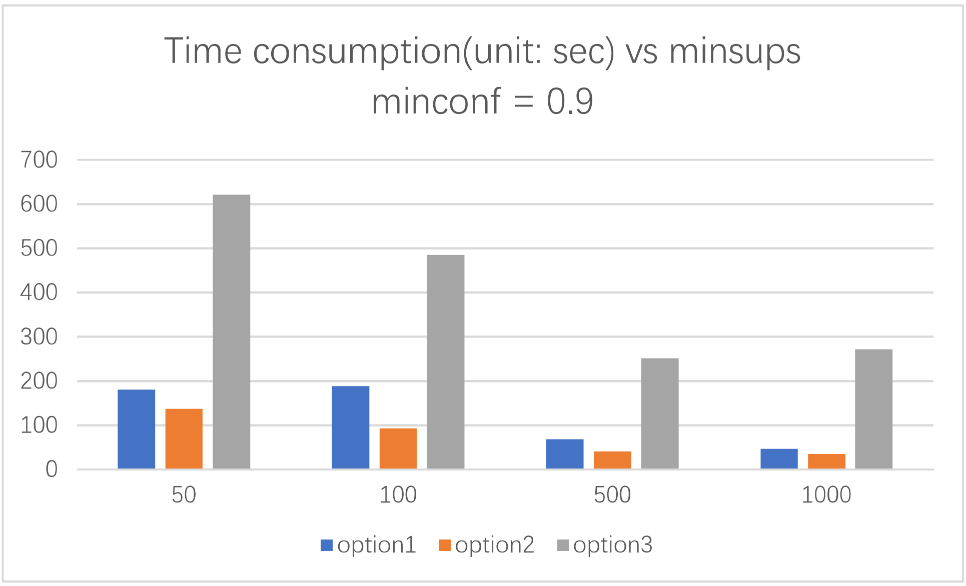
   2.1 The amount of frequent sets and rules and how they change as minsup decreases.

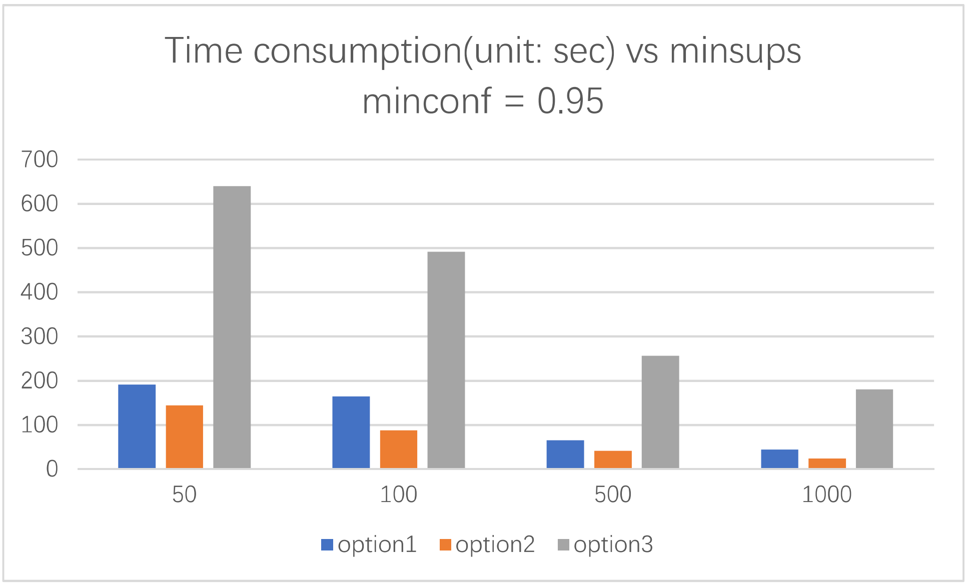




2.2 The time consumption for different options of order and how they change as minsup decreases.







3. **Analysis of the result.**

First as expected, the smaller the minsup and the smaller the minconf is, the larger amount of time is needed. And the asymptotic behavior looks like an inverse power law. The smaller the minconf is, the larger amount of frequent itemset and association rules are generated. This is easily understood. When the minsup is smaller, the amount of frequent item set and the association rules increases, and appears to be an exponential explosion behavior, which makes sense. Because the total number of all possible frequent sets and rules is exponentially dependent on the number of items.

Which value of the options parameter resulted in faster execution time and why?

About the ordering, we can see that option 2 generally requires smaller time than the other orderings, which makes sense. Option 2 is the increasing frequency order. So when doing this recursive projected database algorithm, the smaller frequency is first to be processed. They have less frequent super set, so this ordering tends to postpone computationally heavy part to the later. This prevent the recursive stack from going to deep, which would happen for option 3, the reverse version of option2. Thus accelerate the process.

But option 2 is not always the best. This is because for some data, for example, we can have 1 million transactions of just {e}. So item “e” has much bigger frequency and should be processed later. But actually {e} has only one item in the set, which doesn’t generate too much branches. So it should be calculated earlier. That’s why option 2 can sometimes behaves not as fast as, say, option 1.