Data structures:

Each transaction is stored as an ArrayList of Integers. The database of transactions is therefore an ArrayList of ArrayLists of Integers. All projected databases are also stored in this manner.

Frequent itemsets are represented as space-delimited strings; those strings are hashed to their corresponding frequencies via a HashMap. These strings provide the path of the recursive mining algorithm.

Projection Algorithm Implementation:

The algorithm recursively calls generateFrequentItemsets() in a depth-first manner. At each step, the two arguments passed to this function are the projected database, and the current itemset (represented as a String so that it can be hashed). Initially it is passed the entire database and an empty string.

The function first scans the projected database for any items that are frequent; the frequency of items is tracked with a HashMap. Then it constructs a new database which is a copy of the original, but only keeps items which have a frequency greater or equal to minsup. This process filters out any itemsets that are infrequent. At the same time, any item that has a frequency greater than minsup is added to an ArrayList of candidates. The candidates are sorted in lexicographic order so that they will be expanded in order.

The function proceeds to iterate over and project the database onto all of the candidates (this is done in a depth-first manner to avoid large space complexity). The projection is done by iterating over the original database, and simultaneously constructing a new one. Any row that contains the item being projected onto is sliced at that item's index. This is possible to do because all transactions are sorted in lexicographic order. Any row not containing the item is ignored.

Since the algorithm can only see the item it's currently projecting on, it needs to keep track of the path it took to get to this candidate. Therefore, the candidate is added to the current itemset (it is appended to the string in a space-delimited manner) to construct a new itemset. Also, this candidate was frequent, which means the newly-constructed itemset is frequent. The itemset string is hashed to the frequency of the candidate. With this updated itemset, and the projected database, the function is called recursively. The itemset grows with the depth of the algorithm, and provides the algorithm its current path.

Algorithm Run Times:

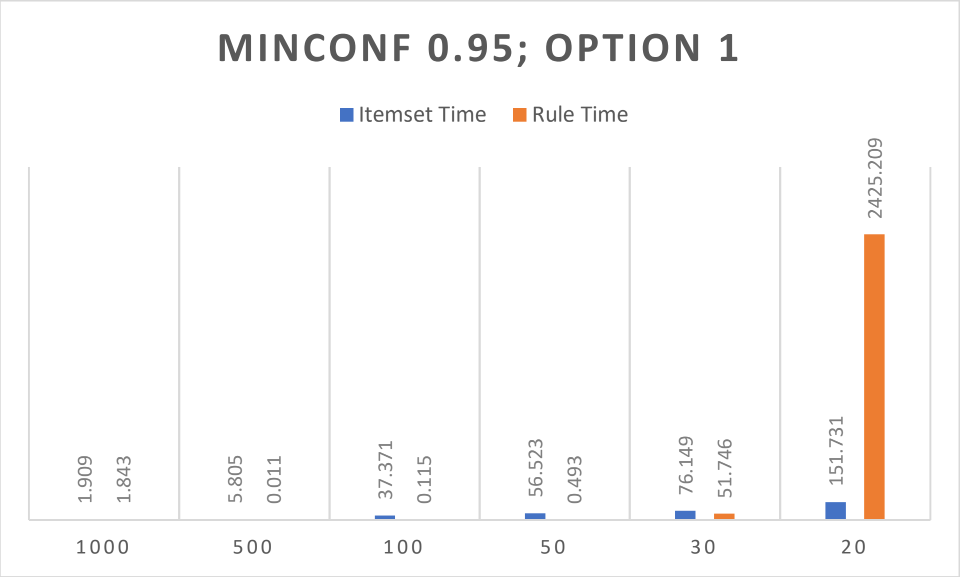
A few notes about my data:

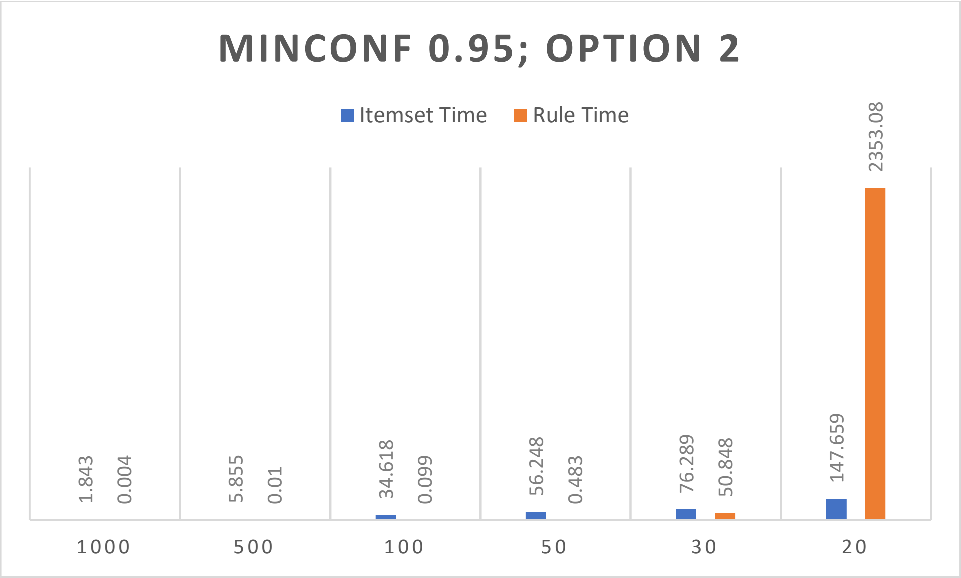
> *Though I've run the program for virtually all Minsup/Options/Minconf combinations at one point or another for testing purposes, unfortunately I underestimated how much time it would take to run and record all combinations. As a result, some values are not filled in: in particular, I was unable to generate association rules for some combinations of Minsup 15 and 20. This is because I understood from the instructions that I wouldn't have to generate these, but after reading the discussion forum, I found that I was mistaken (and left with not enough time to generate them).*

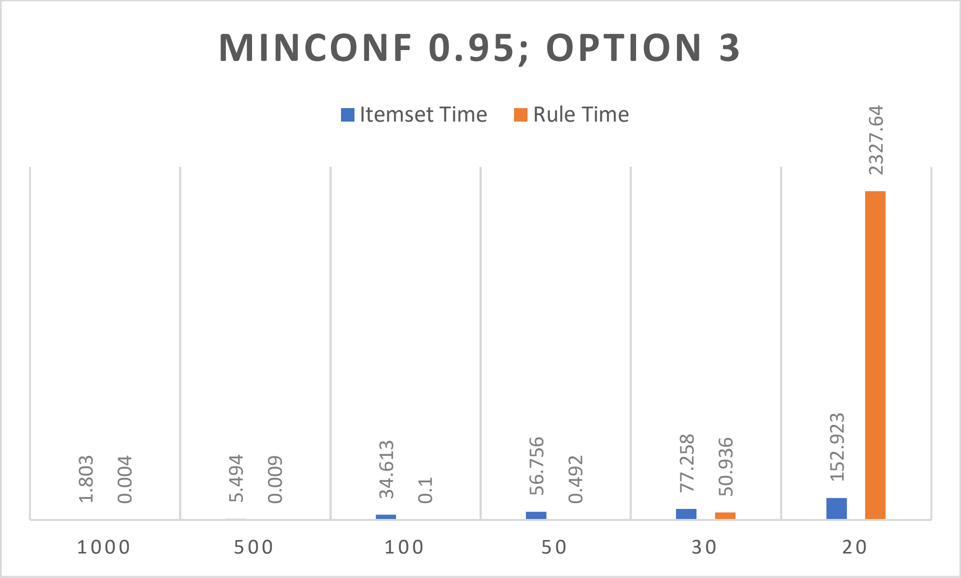
Also, it should be noted that some of these times ended up quite inconsistent. Though all tests were run on CSE Lab machines, I had to switch machines a few times. I found that some machines performed better than others. On the run I eventually recorded, generating frequent itemsets for minsup 15 took a very long time. However, I've run this many times before and gotten times less than 20 minutes. As a result, I'm not sure how reliable much of this data is anyways.

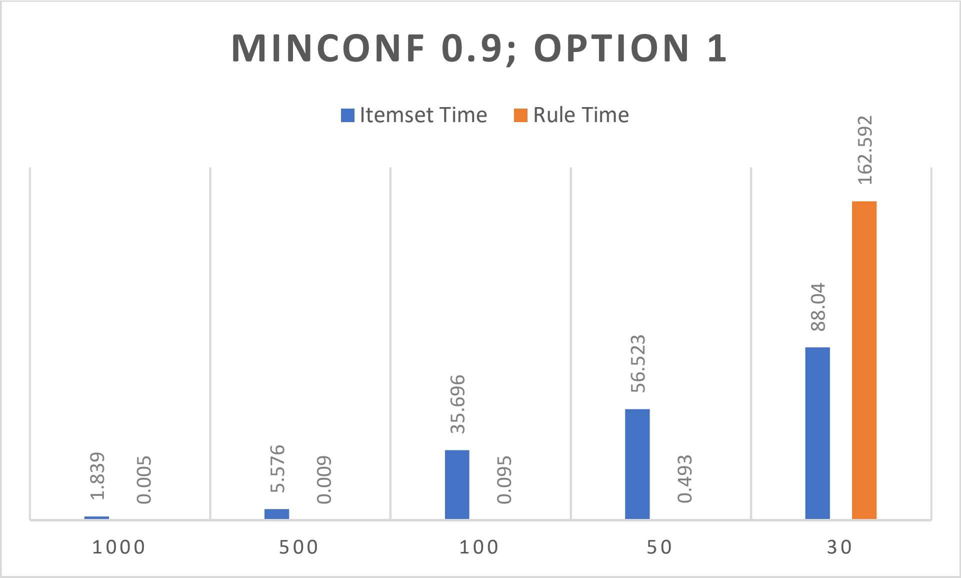
*Finally, I wasn't exactly clear on how many bar charts to make. Though the instructions say to make two bar charts for each minconf and options combination (so 3 * 3 * 2 = 18), this doesn't make much sense to me. The second bar chart is supposed to be the number of rules and itemsets generated for different minsups. However, it doesn't matter what option is chosen; the number of rules and itemsets will stay the same. As a result, I generated 12 charts total. 9 for the first bar chart, and then 3 charts for the different minconf values.*

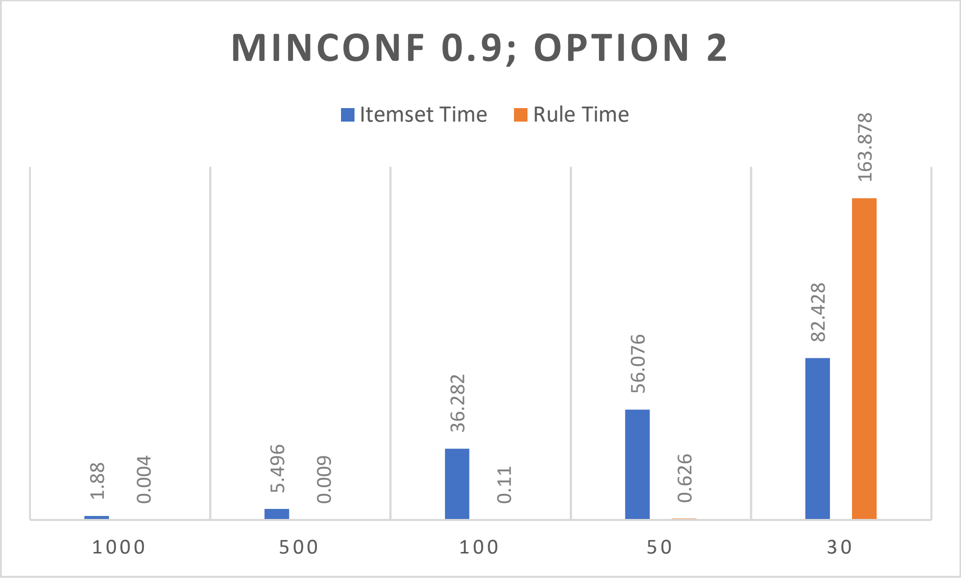
In the bar charts, the x axis represents minsup.

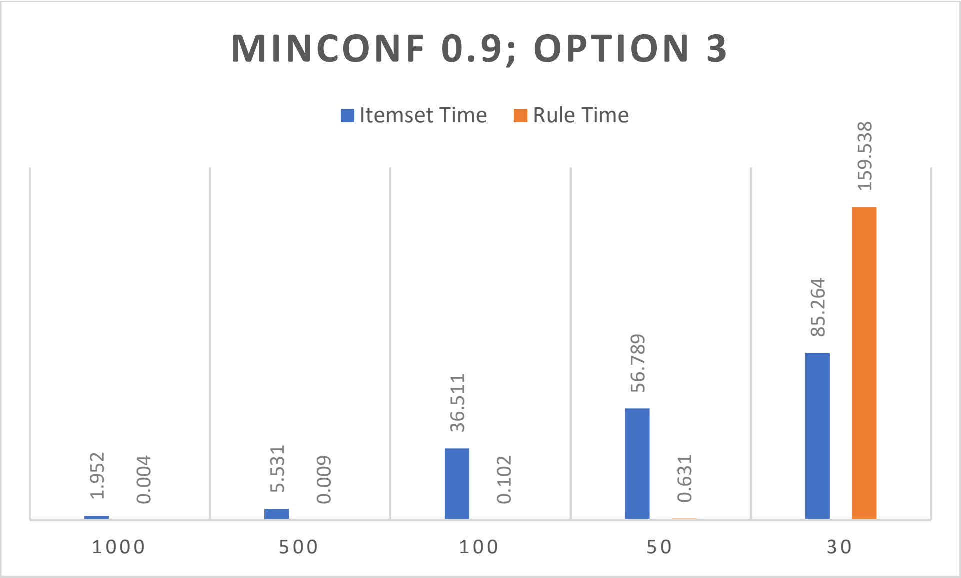


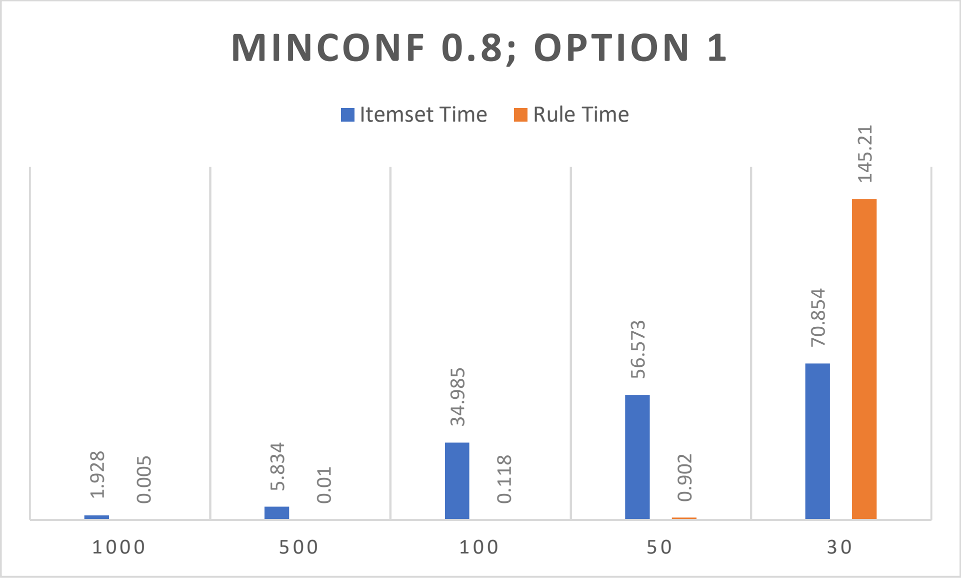

MINCONF 0.95; OPTION 2

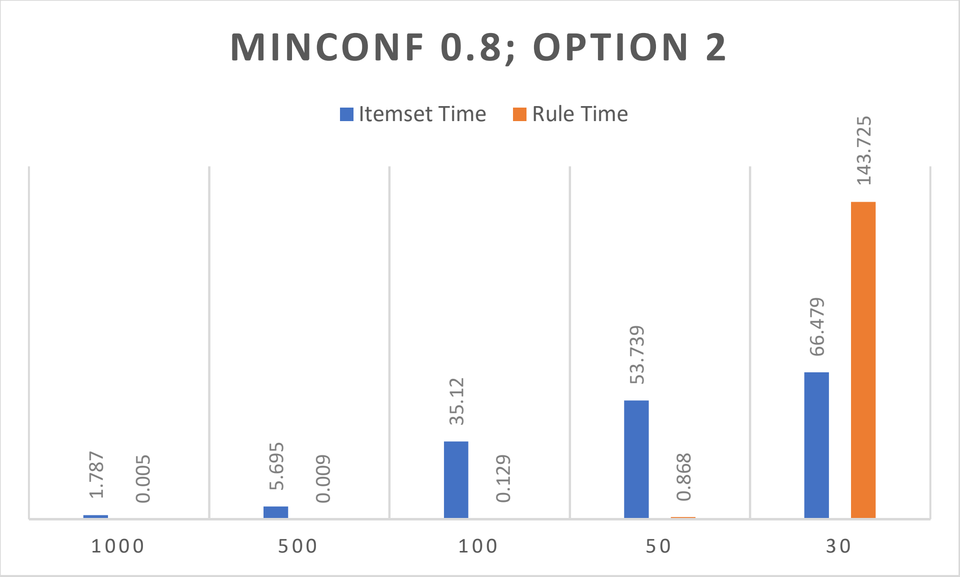

MINCONF 0.95; OPTION 3

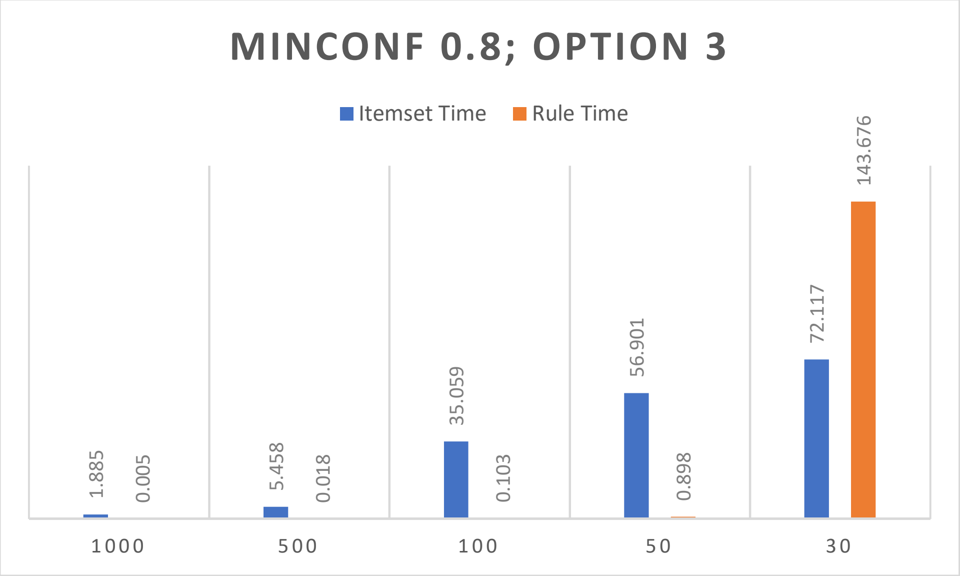

MINCONF 0.9; OPTION 1
Itemset Time
Rule Time
1.839
0.005
5.576
0.009
35.696
0.095
56.523
0.493
88.04
162.592
1000
500
100
50
30

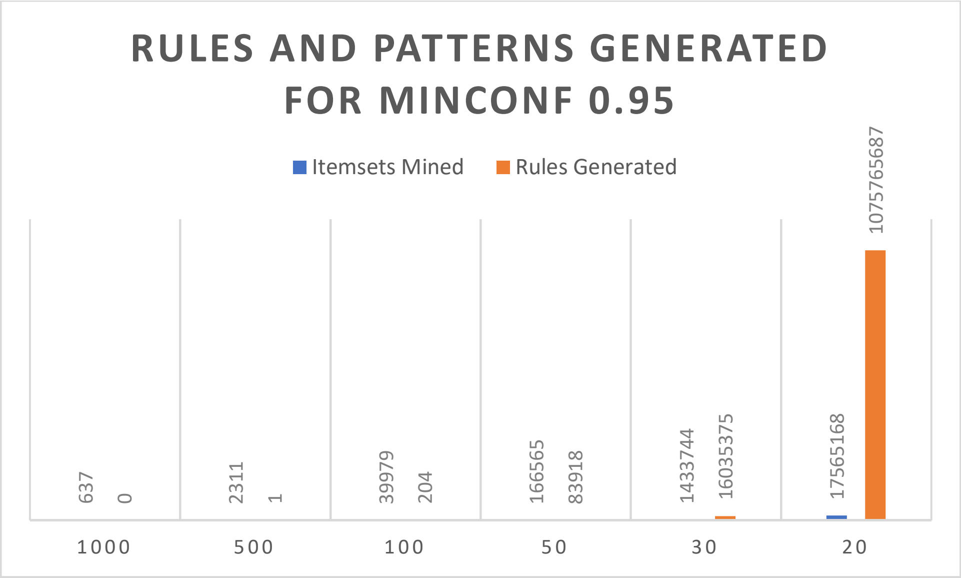

MINCONF 0.9; OPTION 2
Itemset Time
Rule Time
1.88
0.004
5.496
0.009
36.282
0.11
56.076
0.626
82.428
163.878
1000
500
100
50
30

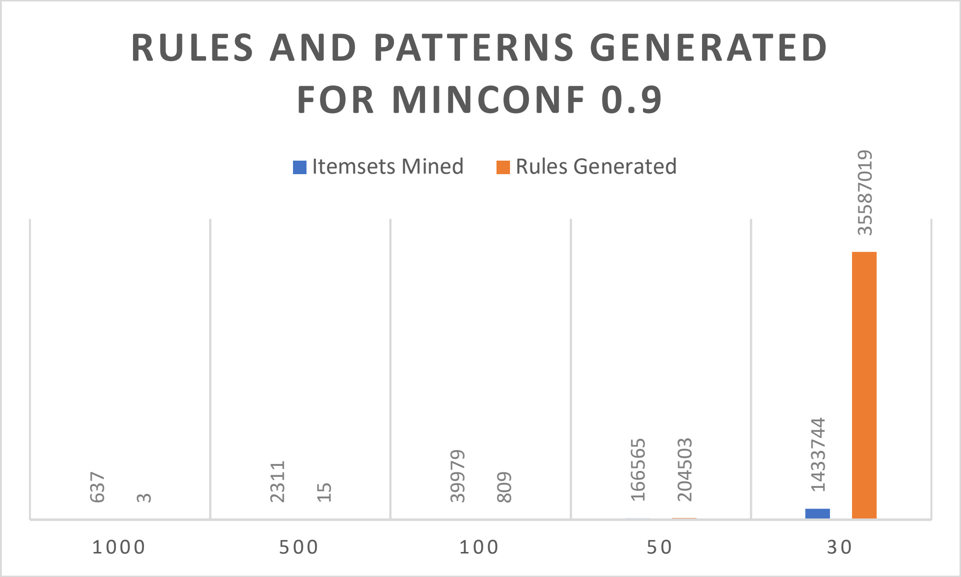

MINCONF 0.9; OPTION 3
Itemset Time
Rule Time
1.952
0.004
5.531
0.009
36.511
0.102
56.789
0.631
85.264
159.538
1000
500
100
50
30

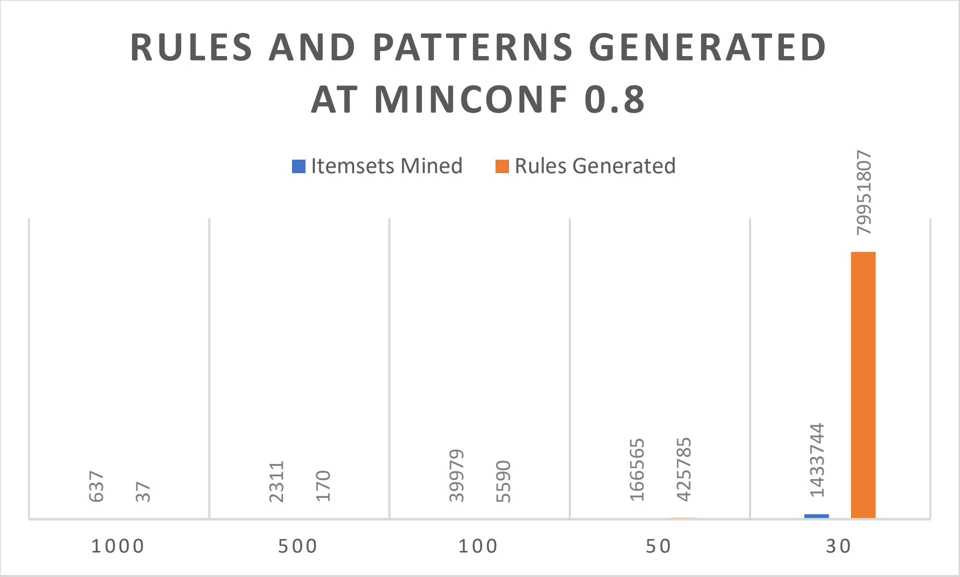

MINCONF 0.8; OPTION 1
Itemset Time
Rule Time
1.928
0.005
5.834
0.01
34.985
0.118
56.573
0.902
70.854
145.21
1000
500
100
50
30


MINCONF 0.8; OPTION 2
🟦 Itemset Time
🟧 Rule Time
1.787
0.005
5.695
0.009
35.12
0.129
53.739
0.868
66.479
143.725
1000
500
100
50
30


MINCONF 0.8; OPTION 3
🟦 Itemset Time
🟧 Rule Time
1.885
0.005
5.458
0.018
35.059
0.103
56.901
0.898
72.117
143.676
1000
500
100
50
30


RULES AND PATTERNS GENERATED FOR MINCONF 0.95
■ Itemsets Mined
■ Rules Generated
637
0
2311
1
39979
204
166565
83918
1433744
16035375
17565168
1075765687
1000
500
100
50
30
20


RULES AND PATTERNS GENERATED FOR MINCONF 0.9
■ Itemsets Mined
■ Rules Generated
637
3
2311
15
39979
809
166565
204503
1433744
35587019
1000
500
100
50
30



Data Analysis:

Though the data are not entirely complete, conclusions can still be drawn about the data. For high minsup values, there doesn't appear to me much correlation between the option chosen and the time taken. However, a pattern starts to emerge when minsup gets smaller (though it's not incredibly pronounced). In particular, option 2 seems to perform the best when minsup gets small. While we were told to look for this, it honestly wasn't obvious to me why this is happening. If I had implemented the algorithm via an FP tree, then this would make a lot of sense; the tree would be a lot more compact with option 2. However, in my algorithm there is no tree, any speed increase can only occur from the change in the way my algorithm expands the data.

My hypothesis is that if an item is common, it's likely to have a lot of supersets. When the lexicographic ordering is done by item frequency, then all of its potential supersets are expanded from one node in the enumeration tree. That means we can quickly determine all its candidates, and thus all of its immediate supersets. Alternatively, if the lexicographic ordering is not done based on item frequency, then a frequent item may reappear at many places in the enumeration tree. Nodes that would otherwise have no children, will end up having a frequent itemset as a child. Furthermore, to find the supersets of a frequent item, we'll have to find all instances of the itemset in the enumeration tree and count its frequency. In essence, **we're expanding this same item multiple times instead of once, leading to a less efficient algorithm.**