ESE 589 Frequent Pattern Growth Tree Algorithm

Fedrik Durao 103302962, Nehul Oswal 112820228

Abstract:

The Frequent Pattern Growth Tree algorithm has been implemented and tested in this project. The goal was to not only successfully implement the intended behavior of the algorithm but to do so efficiently, as to achieve low latency and high resourcefulness, while also being able to observe how the algorithm's results change depending on the data it is being used on. The algorithm behavior and design of this implementation of the algorithm will be discussed in detail. Involving the data structures chosen and structure of the code itself. The code had been validated on simpler data tables, prior to being used on the experimental databases.

Introduction:

The Frequent Pattern Growth Tree algorithm aims to scan a database, and return sets of items or elements in the database that are seen together "frequently". What is considered "frequent" can be varied, as in that the patterns returned can be above some minimal frequency determined by the user. The algorithm starts by counting how many times a specific item has been found in the database, reordering the datasets based off the items' count. Then a tree is formed without items whose count is below the "frequency" minimum, and creates a record of each items position in the tree. Next the frequent associations are mined. In descending count order, a subtree will be produced for the item, containing other items in its path from the prior tree. Items may appear in the subtrees with a count lower than the minimum, and will be removed. The subtrees are created recursively until there is only one 1 branch in the subtree, and the items associated with each subtree will be returned as a frequent pattern.

Design of Software:

Three data structures were used to implement this algorithm. As expected, Data Trees were used to represent the top level tree (created from pruned database) and sub trees of items. Dictionaries or Hash Tables were used to create the headers which recorded the total count of an item in the tree, and the links of all same items within the corresponding tree. Dictionaries were optimal to refer to a specific item quickly, rather than searching for a given item in an array or list. Lists or arrays were also used for intermediate logic throughout the program.

This implementation follows a simple and intuitive program flow beginning with reading and parsing the database given. This is done by the readData( ) function, and then the main header is created by create_header_table( ) using the parsed database, and creating a record of items with their total count. The data is then sorted by sort_data( ) which will sort every itemset based off the header count. Then the create_tree( ) method creates the main tree from the sorted datasets, and stores same item links into the corresponding header slot. Below are the detailed descriptions of methods mentioned earlier.

1. **readData( ):** This function reads every row from the database file as a list and stores every list in a dictionary with row number as key value. As dictionary are inherently unordered therefore ordered dictionary is used so that the list is stored in the same order as they are read. Also, this function counts the occurrence of every item and stores it in a dictionary with item name as key with its count as the value in a sorted fashion.
2. **Create_header_table( ):** This function creates a header table as mentioned in the FP growth algorithm using the dictionaries created in the readData( ) function. The data structure used for creating header table is dictionary of list with the item name as the key value and its corresponding value is a list which contains its count and a reference to its first occurrence.
3. **Sort_data( ):** This function is used to sort every tuple in descending order according to the count of items in the entire database. This function uses the header table to access the count of every item in the database and returns a dictionary of every tuple sorted in descending order.
4. **Class Tree_Node:** is a class defined with its member variables as item name, item count, reference to parent node, list of its children and a reference to its node link.
5. **Create_tree( ):** This function is used to create the main FP tree. This function is called for every tuple in the database. Initially for every root the null root is passed, and the root is updated as we further down the tuple. This function checks if current item is in the children list of the current root. If it is present, then just the count of the children is incremented by 1 and for the tuple value the current child will be the root. If the current tuple is not present, then a new node is created with a reference to the current root as its parent and its count as 1. In addition to this, this function checks if there is a reference to this item value in the header table, if there is no reference this indicates that this is the first occurrence of this item value and the header table is updated with the reference of the item value and if there is a reference we travel to its neighbor node until we find a node link which is empty and which is then updated with the reference of the current item value.

Once the main Tree has been created, the generated_patterns_recur( ) function is called for each initial item in the header. All of the frequent patterns are generated from this function and the other functions within it. First all links made to the current item from the header are iterated and stored into a list of nodes. Then for every node, all parents leading to a specific node are stored into a list a single list. These lists are produced recursively by recur_getParentList( ), and are put into another list. From the lists of parent sets, and same node links, a new subheader is formed by create_sub_header( ), and a new subtree is formed by create_subtree( ) The resulting node links from the sub tree are stored in the sub header. Afterwards the sub tree is then "pruned" based off the iceberg condition, meaning that the nodes (or items) that do not meet the iceberg condition are removed. This is done by the pruneTree( ) function. Additionaly, the sub tree is then scanned by recur_moreThan1path( ) to determine if there are more than 1 path in the sub tree (after the root). This is important because the generated_patterns_recur( ) function is recursive, and a sub tree having only 1 path will be the means to end recursion. If there were more than 1 path after the root node of the sub tree, then generated_patterns_recur( ) is called for each subheader item, concatenating the results of each recursion. If for any recursion, there is only 1 path in the newly created sub tree, then gen_base_fp( ) will generate a base set of frequent patterns will be created from all subsets of items or nodes in the sub tree, in. Every returned

frequent pattern set is also concatenated with the item or element that those patterns were found for. Below are the detailed descriptions of the functions used in generated_patterns_recur( ).

1. **recur_getParentList( ):** Checks if the current node's parent is the root node. If not, then it will iterate recursively, changing the node from its current self to its parent. If the current node's parent is the root node, then the current node is appended to the list. After a returned recursion, the current node from that recursion is also appended. Resulting in a list from highest parents to lowest parents.
2. **create_sub_header( ):** Uses the list of parents, and for every distinct parent item, a new key is made in the header dictionary, with a value of the count of the corresponding child node found in "theNodeList". The "theNodeList" is the list of same item nodes from the start of generated_patterns_recur( ). If a parent item is not distinct, the count of the corresponding child node is added to the count at the dictionary for that parent's key. Every header key is than given a blank link not pointing to any nodes yet (sub tree needs to be made first).
3. **create_subtree( ):** For every node in the parent list, checks if the node is found in root's children. If not, then the node is added as a child to the root. If the header does not have a link for that node's key, then that node is added to the header. If the header does have a link for that node's key, then that node is linked to the end of the link chain. Earlier if the node is found in the root's children set, then the child of the root has its count updated. Every iteration, the root changes to its child.
4. **pruneTree( ):** Checks to see if the current node in the sub tree has a count above the iceberg (if it is not the root). If not, then this node is removed from its parent's child set and leaves the recursion. If not, then if the node has no children, then that recursion ends. If the current node has children then recursion will iterate through each of its children as the next current node. This ensures that every node is checked against the iceberg condition.
5. **recur_moreThan1path( ):** Recursively iterates through every node in the sub tree. If any node has more than 1 child, a "True" is immediately returned. A "False" is returned otherwise.
6. **gen_base_fp( ):** This function is called if there is only every child of the root has only 1 branch. For every direct child the root of the subtree, all_subsets( ) is called. This function returns every subset of the nodes in the branch.

Other small portions of the code have not been discussed. These portions are mostly just band-aid logic to cover for conditions that these functions missed.

Validation:

Before taking measurements of the algorithm for different databases, the algorithm was tested on smaller more readable databases made by hand. Similar to the one used in the textbook and class examples. As changes were made to the simple database, the Frequent Pattern Growth Tree was done by hand and then compared to the results for the algorithm for each three parts. Many potential errors were eliminated while changing the small database and debugging the code. But there were indeed a few more errors afterwards when working with the large legitimate databases. These errors were eliminated after using the larger databases. Most of those late errors

were related to the items not being discrete among different dimensions. This was mitigated by appending a dimension letter to every item when the database is first scanned. Without a large database, it was difficult to visualize all potential problems since the algorithm could not be verified exhaustively when testing the algorithm on the small database.

Below is our results of Frequent Pattern Growth Tree based on a simple database, modeled after class examples.

```
1;2;3;4
I1;I2;I5
I2;I4
I2;I3
I1;I2;I4
I1;I3
I2;I3
I1;I3
I1;I2;I3;I5
I1;I2;I3

```

Figure 1: Original Database

```
Patterns:  []
Patterns:  [[4, 'I2', 'I1']]
Patterns:  [[4, 'I2', 'I3'], [2, 'I2', 'I1', 'I3'], [4, 'I1', 'I3']]
Patterns:  [[2, 'I2', 'I5'], [2, 'I2', 'I1', 'I5'], [2, 'I1', 'I5']]
Patterns:  [[2, 'I2', 'I4']]
```

Figure 2: Original Database results with pruning

```
1;2;3;4
I1;I2;I5
I2;I4
I2;I3
I1;I2;I4
I1;I3
I2;I3
I1;I3
I1;I2;I3;I5
I1;I2;I3
I2;I3;I4
I1;I4
I1;I4
I2;I4
I1;I4
I2;I4
```

Figure 3: Expanded Database

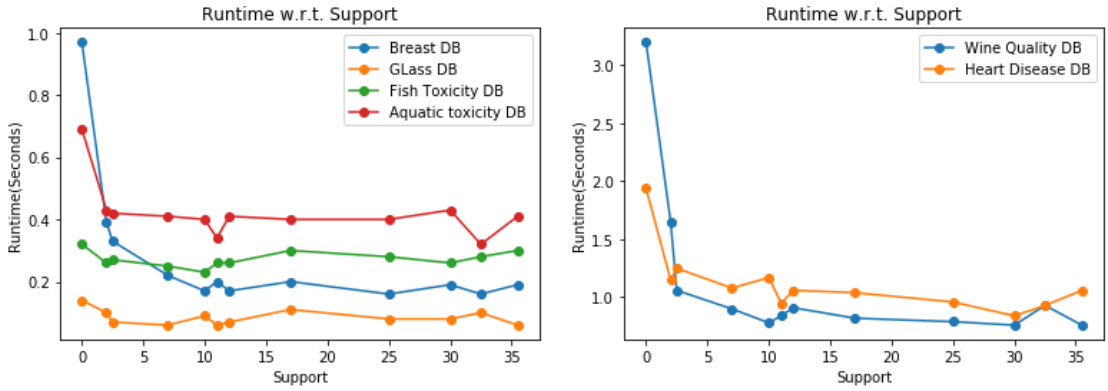
```
Patterns:  []
Patterns:  [[4, 'I2', 'I1']]
Patterns:  [[5, 'I2', 'I4'], [4, 'I1', 'I4']]
Patterns:  [[5, 'I2', 'I3'], [2, 'I2', 'I1', 'I3'], [4, 'I1', 'I3']]
Patterns:  [[2, 'I2', 'I5'], [2, 'I2', 'I1', 'I5'], [2, 'I1', 'I5']]
```

Figure 4: Expanded Database results with pruning

Results and Analysis:

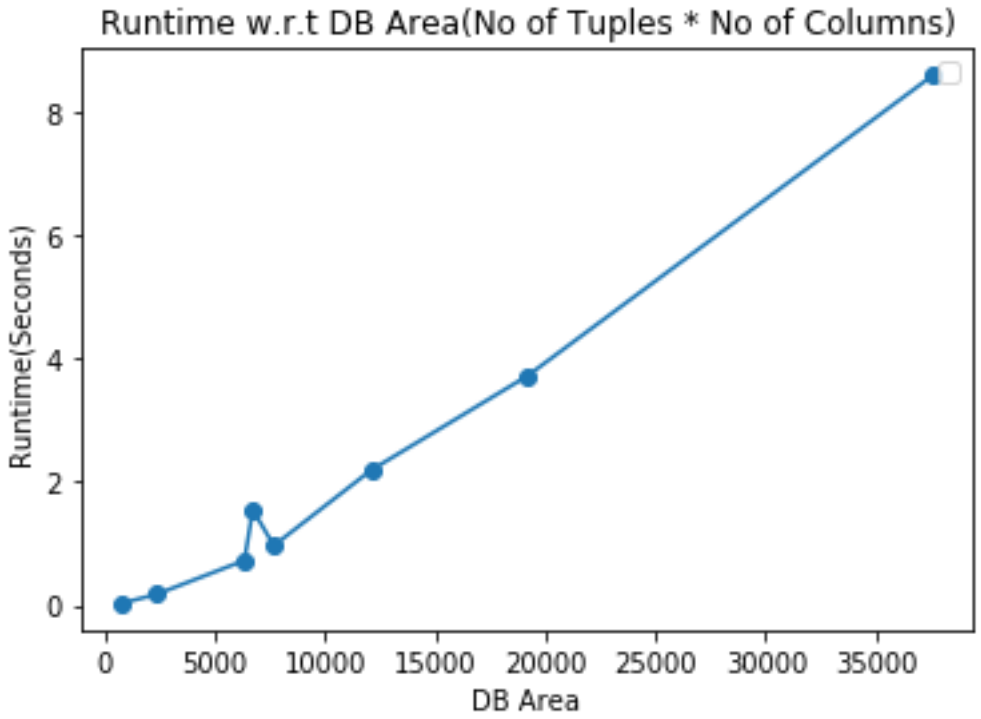
Three relationships were depicted from graphing the results of Frequent Pattern Growth Tree on the six databases: Wine Quality, Aquatic Toxicity, Fish Toxicity, Breast Cancer, Heart Disease, and Glass. The relationships observed are execution time v. support, time v. database size, and patterns v. support. These results were produced on a laptop with 12GB of RAM and an i7 processor.

1) Time v. Support



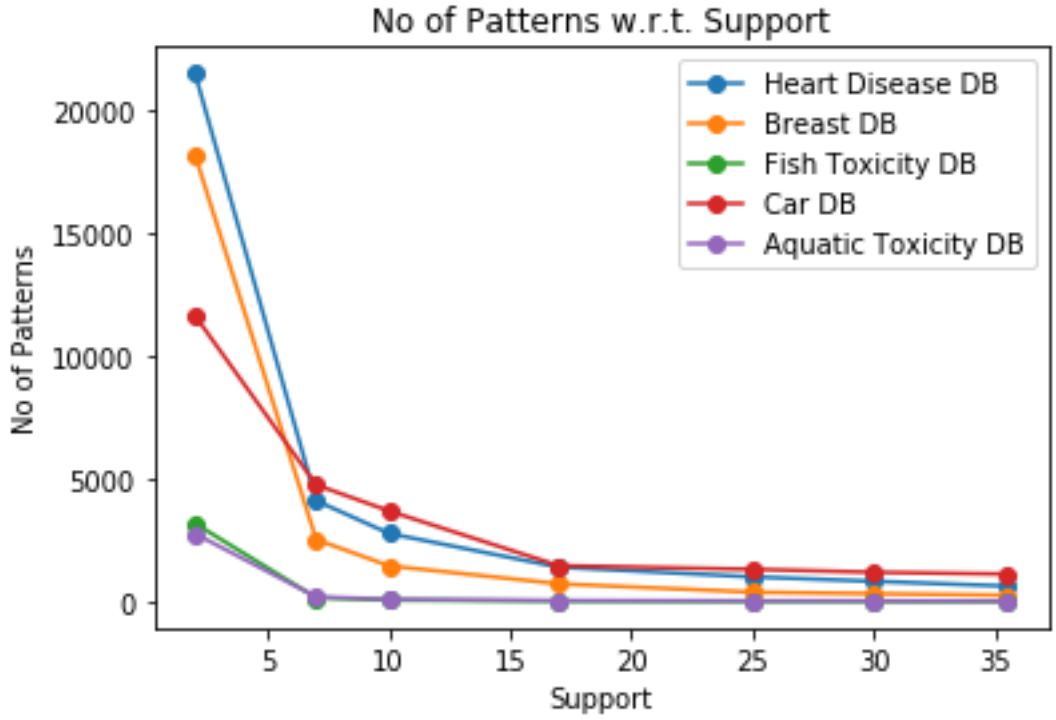
The results were expected to have runtime decrease as the support increases. This holds true for all instances in these graphs. As support increases, the graphs flatten out as expected. These flat regions were likely to occur since they can be attribute to many factors. The imbalance of frequent patterns can lead to this. Pruning may make the program faster, but there are still functions that will not change in performance relative with pruning. Scanning of the database, creation of main tree and sub trees (pruning takes place after creation) will still take their nominal runtime regardless of an iceberg condition. These constant runtimes may be what is seen in the flat regions, possibly being the bottleneck of the system.

2) Time v. Database size



As expected, the runtime increases for increasing number of data entries. The relationship is quite linear, suggesting that the database size has consistent effects on the run time. The spike could be attributed to some databases simply just having less items to prune/eliminate.

3) Number of Patterns v. Support



With increasing support, the number of resulting patterns should decrease, since more patterns will not qualify and will be eliminated. This is accurately illustrated from the graph. Since the resulting patterns do decrease with support, that suggests that the algorithm is pruning correctly. This graph agrees with the Time v Support graph in the sense that the change in the number of patterns being small suggests that not much time was saved by eliminating a few extra patterns. This correlation can be made for the graphs at large support values.

Conclusion:

Most challenges with the algorithm came from having to create new sub trees. Simple lapses in logic can utterly ruin the structure and integrity of the sub trees, resulting in incorrect patterns. Having to visualize where a node in a tree is and determine where it is supposed to be can be confusing at times. It is in situations like this that reliable and well tested insert and traverse functions be implemented for the Tree data structure.

Before implementing the Frequent Pattern Growth Tree algorithm, the value in producing frequent patterns was evident to the group. From the pattern sets; confidences, supports, and correlations can be made on the large datasets, with relatively low complexity. The results displayed relationships that one would expect to see. The simplicity of this algorithm made these relationships easy to achieve.

Many small things may have been improved with this implementation of the Frequent Pattern Growth Tree Algorithm. The fundamental flow and structure of the implementation is for the most part is fine. But there are many areas that could undergo optimizations. When recursively making sub trees and sub headers, a lot of the data was stored from the established dictionaries into lists, for better visualization of certain data when trying to manipulate the data. This increases memory and wastes time storing vales that have already been stored in another data type. Multithreading this implementation would be feasible and would definitely yield better runtimes. And making more reliable Tree methods and recursive loops would mitigate a lot of the overhead and band-aid logic that surely slows the system down. Pruning could not be optimized unfortunately, since during the creation of the sub trees, a nodes count will accumulate as more identical nodes are added. Meaning pruning must wait until no more accumulations take place and will strictly occur after creation of the sub tree. Preferably this would be done during creation if possible.