

PROJECT REPORT

1 Introduction

In this project, I studied on frequent pattern analysis techniques for Market Basket Analysis. The idea is coming from a Featured Prediction Competition. In 2018, Instacart, a grocery ordering and delivery app shared their transaction and product label data-sets and they started a competition for making a market basket analysis on these data sets. I implemented FP-Growth algorithm for this concept. The FP-Growth algorithm implementation is presented in the Implementation (4) section. Here, I am going to explain the analyses that I had done over some frequent pattern analysis algorithms.

2 Frequent Itemset Mining Algorithm Analysis

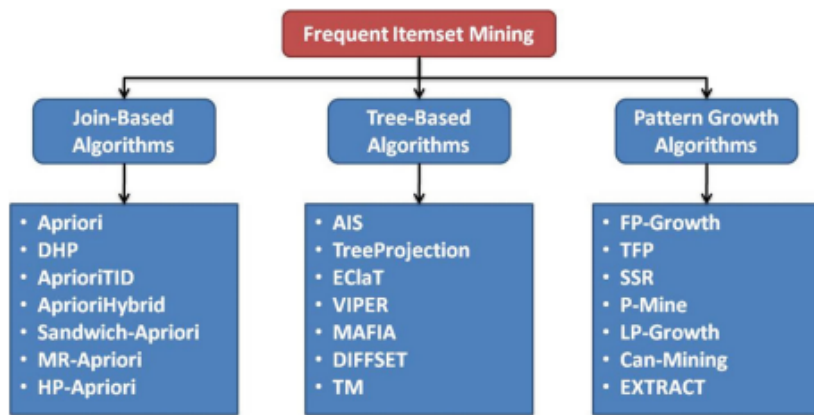


Figura 1: Frequent Itemset Mining

There are several frequent pattern set mining algorithms. But some of them are more popular. I searched every algorithms in the figure 1 and I only found several algorithm implementations for these methods. These are:

apriori algorithm, eclat and fp-growth.

2.1 Comparing My FP-Growth Algorithm with Python Implementations

I compared my FP-growth algorithm with an another python implementation and created some graphs according to results. The tests include 2 steps with following graphs and explanations.

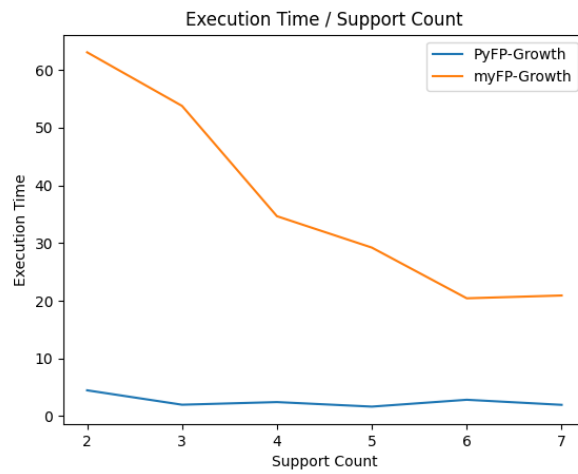


Figura 2: Execution Time / Support Count

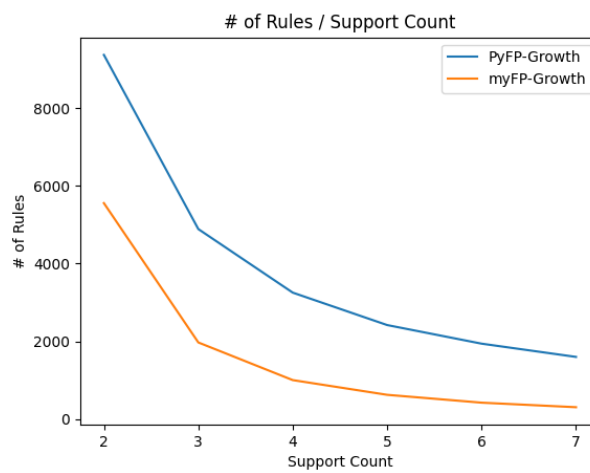


Figura 3: Number of Rules / Support Count

As it can be shown in the figures, my implementation is obviously very inefficient compared to other libraries. I think the problem is the recursive approach for frequent pattern mining. Mining frequent items with tree traversing could be much more efficient. Also number of rules that are found by my algorithm slightly less than the python implementation.

2.2 Comparing Other Frequent Pattern Mining Algorithms

All algorithms are tested with several cases and compared with following graphs. It should be noted that, it was very hard to find suitable parameters for testing all techniques and comparing them each other.

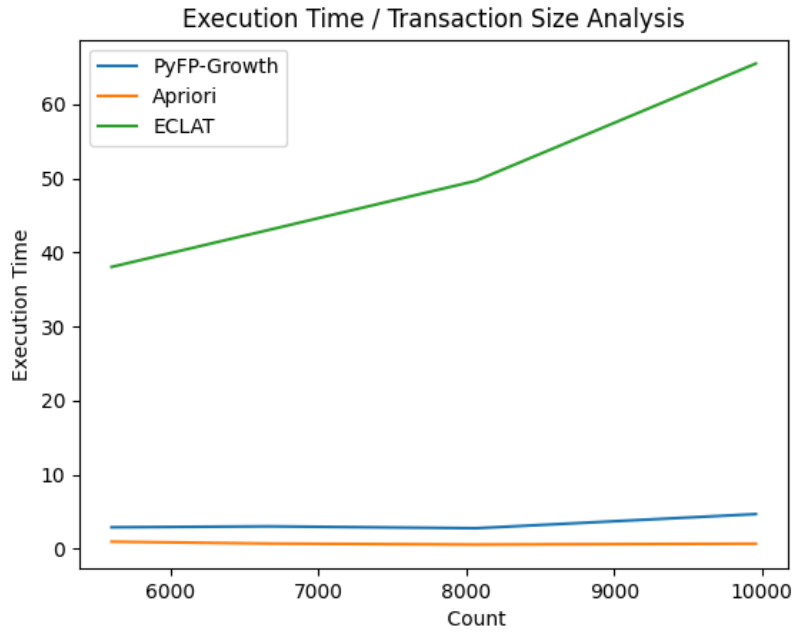


Figure 4: Execution Time / Transaction Size Graph

In the first graph, I tested three different algorithms with different size of transaction files and same values for rest of the parameters such as minimum support count. The ECLAT algorithm obviously has the worst efficiency. Since we know that the main disadvantage of ECLAT is that it requires more memory space and processing time, this is an expected result.

In order to compare FP-Growth and Apriori with larger datasets efficiently, in the following graphs, I compared only these 2 algorithms.



Figure 5: Execution Time / Transaction Size Graph

In this graph, we can see that FP-Growth algorithm execution time takes much more time. In order to make a good summary, I also counted the number of association rules that these functions returned.

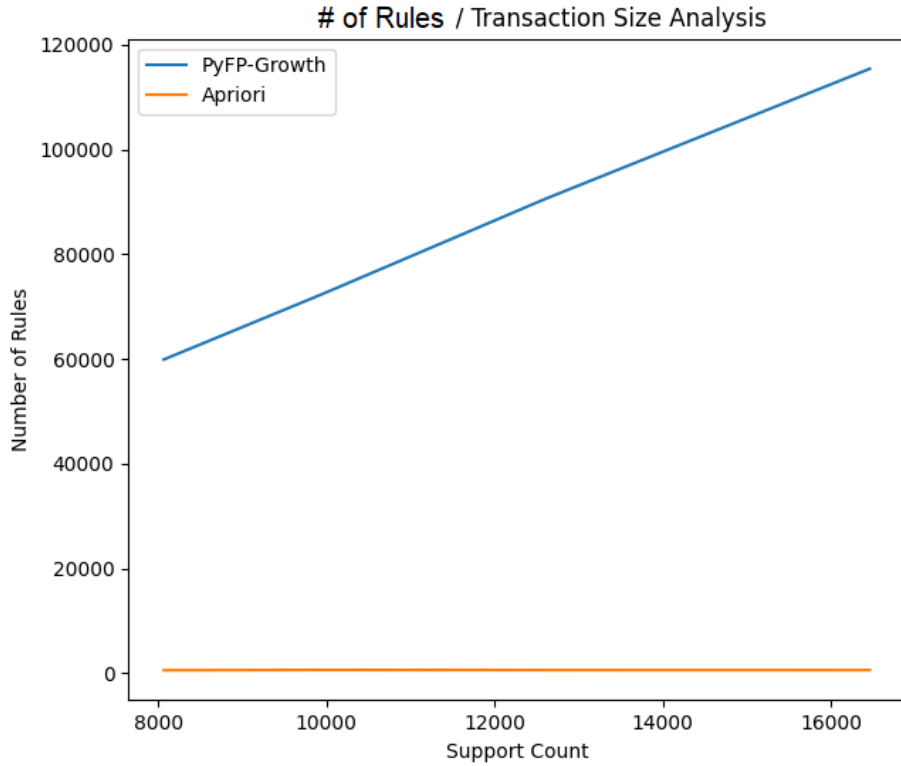


Figura 6: Number of Rules / Transaction Size Graph

We can see that FP-Growth algorithm execution takes long time and it also creates more rules from the dataset. On the other hand, the apriori algorithm finds nearly no rules and that's why the execution does not take much time.

Here, when I tried to increase the transaction size, the apriori algorithm crashed. The terminal simply killed the process. I think, the problem is that apriori algorithm uses system resources too much. But with less transaction size, it also cannot find many rules. I also decreased support count and created many graphs in order to see if apriori can find more rules. But it still failed to find more rules.

The summary for these graph is, when the size of the candidate items is extremely large, FP-Growth algorithm best to use. But also, my dataset is not appropriate for the apriori algorithm.

In order to figure out what is wrong with my analyses, I used another

apriori algorithm implementation and get different results as in the following graphs.



Figure 7: Execution Time / Transaction Size Graph

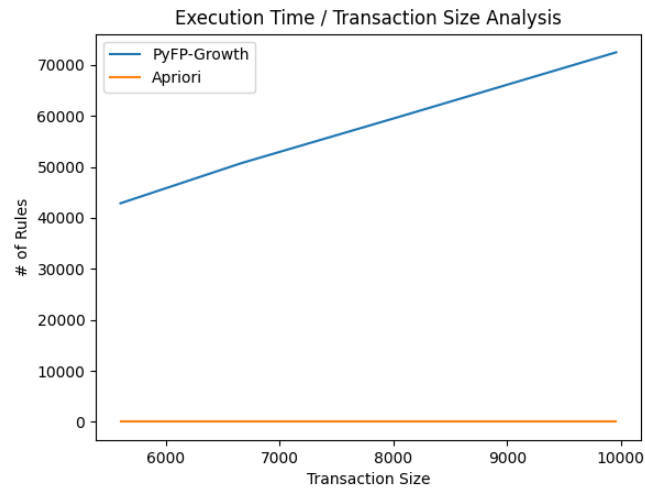


Figure 8: Number of Rules / Transaction Size Graph

Here we can see that, the apriori algorithm still cannot find rules in my Instacart dataset. On the other hand, the execution time is much more slower than FP-Growth algorithm. I guess, in the first tests, the apriori algorithm was so fast because it somehow passed many steps when it cannot find any rules.

I should also note that, I decremented support count very much to find rules with apriori algorithm. That's why, the rule number results of the FP-Growth algorithm is actually equal to maximum number of rules that FP-Growth can find.

3 The Article

I read the article "An improvement of FP-Growth association rule mining algorithm based on adjacency table", Ming Yin, Wenjie Wang, Yang Liu, and Dan Jiang. In the article there is an improved implementation of FP-Growth algorithm by using adjacency table.

The algorithm firstly scans the database and creates header table and fp-tree as in the regular algorithm. The header table includes support counts and list of node links. Until this point, there is no difference with the regular algorithm.

After that, it creates a graph using the created fp-tree and header table. Here, the items in each item sets considered related to each other. The weight of edge is considered to be frequency. Then, it crafts an adjacency table using the graph.

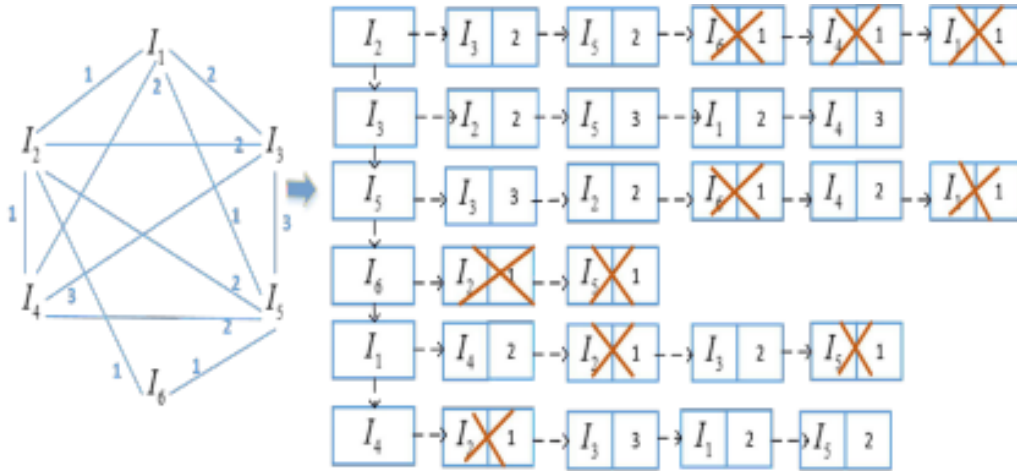


Figure 9: The graph and adjacency table

Using the adjacency table, the items below the support are removed. At the end, the algorithm analyses the table to find frequent sets. The frequent sets are, basically subsets of the possible combinations of nodes.

The article link: https://www.researchgate.net/publication/326944335_An_improvement_of_FP-Growth_association_rule_mining_algorithm_based_on_adjacency_table

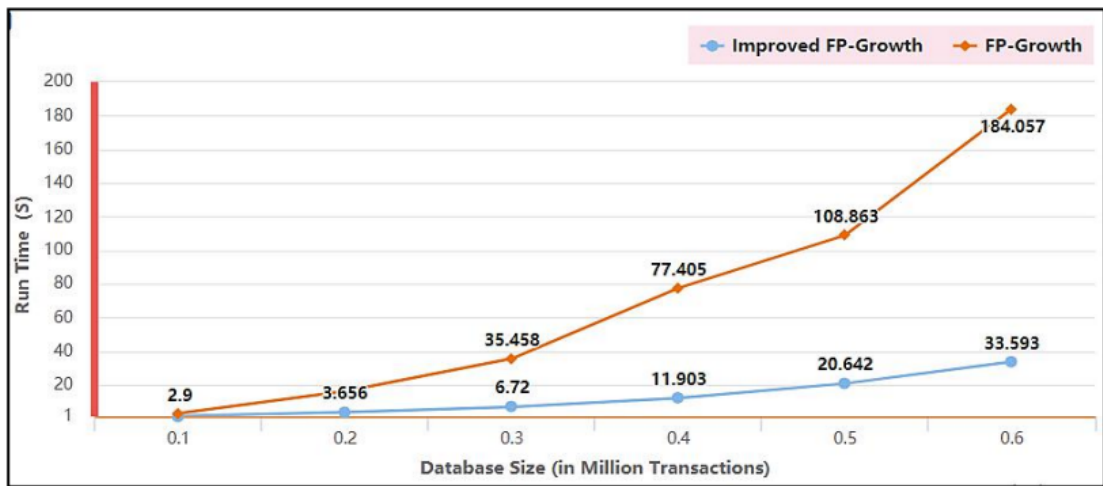


Figura 10: Performance comparison.

4 Implementation

The project separated into 3 files which are "main.py", "FPTree.py", "FPGrowth.py".

The "FPTree.py" file includes main algorithms. The "FPGrowth.py" file includes the `fpgrowth()` method which starts the frequent pattern mining process. Inside this method, firstly `ConstructTree()` and then `MineTree()` algorithms are called from the "FPTree.py" file.

The tree construction starts with "`ConstructTree()`" algorithm. This method firstly calls "`CreateHeaderTable()`" and then it creates a dictionary while deleting the items below support.

```
# - Constructs FP-Tree or Conditional FP-Tree.
# To construct conditional FP-Tree, a frequency list must be provided.
# otherwise, custom frequency '1' will be used.
# - HeaderTable = Product Name: [Freq, [Node List]]
def ConstructTree(self, dataset, support, freq=None):
    if (freq == None):
        freq = [1 for i in range(len(dataset))]

    # Create the header table
    headerTable = self.CreateHeaderTable(dataset, freq)

    # Delete the items below support
    headerTable = dict((item, sup) for item, sup in headerTable.items() if sup[0] >= support)

    if (len(headerTable) == 0):
        return None, None

    #print(headerTable)

    # Create test file
    #self.print_dict("test.txt" , headerTable)

    # Sorted transactions
    st = self.SortTransactions(dataset, headerTable)
    #print(st)

    #self.test_transactions(st, headerTable)

    tree, ht = self.BuildTree(st, headerTable, freq)

    return tree, ht
```

Figura 11: Construction of FPTree

```

# - Creates header table according to given dataset
# - Dataset should be 2D list
def CreateHeaderTable(self, dataset, freq):
    headerTable = {}

    for i, trans in enumerate(dataset):
        for product in trans:
            if product in headerTable:
                headerTable[product][0] += freq[i]
            else:
                headerTable[product] = [freq[i], []]
    return headerTable

```

Figura 12: Header Table Creation

After that, "SortTransaction()" method is called to sort transactions according to header table. This method returns 2D list of sorted transactions. Transactions are sorted descending according to support count.

```

# Sorts transactions according to frequencies (Freq) in the header table
# - HeaderTable = Product Name: [Freq, [Node List]]
def SortTransactions(self, dataset, headerTable):
    sheaderTable = [item[0] for item in sorted(list(headerTable.items()), key=lambda p:p[1][0], reverse=True)]

    newList = []
    for i in range(len(dataset)):
        newTrans = [x for a in sheaderTable for x in dataset[i] if x == a]
        newList.append(newTrans) if len(newTrans) > 0 else None

    return newList

```

Figura 13: Sorting Transactions

At the end, the "BuildTree()" method is called to start building the FP-Tree.

In this method, firstly the root of the tree created. After that, the items in each transaction (the transaction is sorted before according to descending support count) are processed and a branch is created for each transaction. If the product's (an item in the transaction) node already exists in the path, then the count (frequency) variable of the node is incremented.

For each newly created node of an item, the node is added header table of the item. That's how, for each item in the header table, a node-link list created. At the end, a new header table with node-link list and fptree is returned.

```
# - Constructs FP-Tree or Conditional FP-Tree according to transactions list.
# To construct conditional FP-Tree, a frequency list must be provided.
def BuildTree(self, transactions, headerTable, freq):
    # Create Tree
    tree = FPNode(None, None, None)

    for i, trans in enumerate(transactions):
        currentNode = tree

        for product in trans:
            if product in currentNode.children:
                currentNode.children[product].increment(freq[i])
            else:
                newNode = FPNode(product, freq[i], currentNode)
                currentNode.children[product] = newNode

                headerTable[product][1].append(newNode)

            currentNode = currentNode.children[product]

    return tree, headerTable
```

Figura 14: BuildTree Method

After creating the FP-Tree, in order to create frequent pattern sets, the header table is the only need.

The frequent pattern set mining starts with MineTree() method. This method calls MineTreeRec() method with some specific arguments. The "freqItemList" arguments will contain the frequent item sets. And the "pre"list is used to store prefix path of the recursively searched item.

```

def MineTree(self, headerTable, support):
    freqItemList = {}
    self.MineTreeRec(headerTable, support, list(), freqItemList)

    return freqItemList

def MineTreeRec(self, headerTable, support, pre, freqItemList):
    sortedProductList = [item[0] for item in sorted(list(headerTable.items()), key=lambda p:p[1][0])]

    # (ItemName, [Freq, [Node_List]])
    for ItemName in sortedProductList:
        newFreq = pre.copy()
        newFreq.append(ItemName)

        hashableFreq = tuple(newFreq)
        if ((hashableFreq not in freqItemList) or (freqItemList[hashableFreq] > headerTable[ItemName][0])):
            freqItemList[hashableFreq] = headerTable[ItemName][0]

        nodes = headerTable[ItemName][1]

        # Create conditional pattern base
        condBase, freq = self.CondPatternBase(nodes)

        # Create conditional fp-tree
        tree, newHeader = self.ConstructTree(condBase, support, freq)

        if newHeader != None:
            self.MineTreeRec(newHeader, support, newFreq, freqItemList)

```

Figura 15: Mining The FPTree

The MineTreeRec() method recursively creates Conditional Pattern Base list by calling "CondPatternBase()"method and Conditional trees by using ConstructTree() method. The The newly created header tables are again used recursively to create nested frequent pattern lists.

The "CondPatternBase()"method creates conditional pattern base lists by traversing each node to tree root.

```
# Creates conditional pattern base lists.
# For each node, finds the path to node's root.
def CondPatternBase(self, nodes):
    condBase = []
    freq = []

    for n in nodes:
        path = []
        self.ascendTree(n, path)

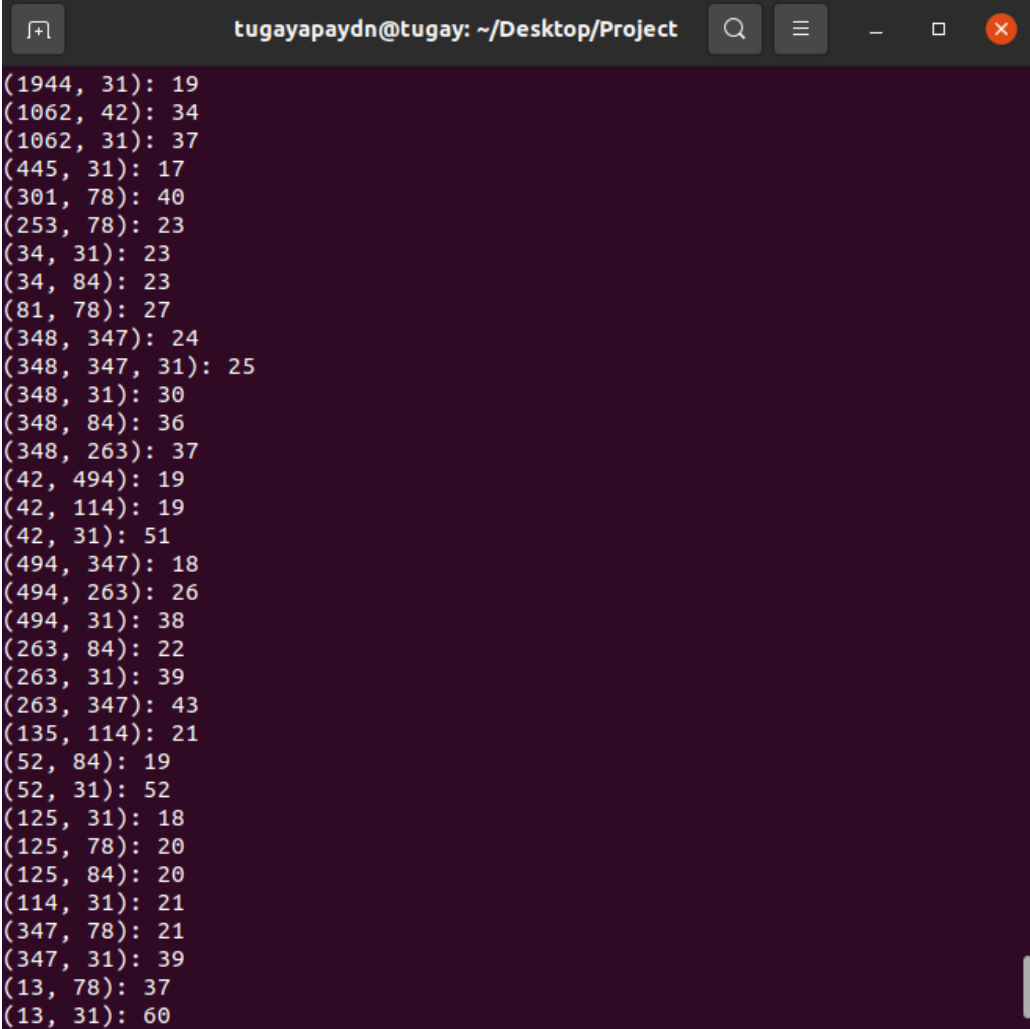
        if (len(path) > 1):
            condBase.append(path[1:])
            freq.append(n.freq)

    return condBase, freq

# Finds the path from a node to it's root
def ascendTree(self, node, path):
    if node.parent != None:
        path.append(node.product_id)
        self.ascendTree(node.parent, path)
```

Figura 16: Conditional Pattern Base List Creation

At the end the outputs are as in the figures:



```
tugayapaydn@tugay: ~/Desktop/Project
(1944, 31): 19
(1062, 42): 34
(1062, 31): 37
(445, 31): 17
(301, 78): 40
(253, 78): 23
(34, 31): 23
(34, 84): 23
(81, 78): 27
(348, 347): 24
(348, 347, 31): 25
(348, 31): 30
(348, 84): 36
(348, 263): 37
(42, 494): 19
(42, 114): 19
(42, 31): 51
(494, 347): 18
(494, 263): 26
(494, 31): 38
(263, 84): 22
(263, 31): 39
(263, 347): 43
(135, 114): 21
(52, 84): 19
(52, 31): 52
(125, 31): 18
(125, 78): 20
(125, 84): 20
(114, 31): 21
(347, 78): 21
(347, 31): 39
(13, 78): 37
(13, 31): 60
```

Figura 17: Output Example

```
tugayapaydn@tugay: ~/Desktop/Project
NaturalChicken&SageBreakfastSausage,GrapefruitSparklingWater: 6
NaturalChicken&SageBreakfastSausage,CageFreeExtraLargeGradeAAEggs: 13
NaturalChicken&SageBreakfastSausage,CageFreeExtraLargeGradeAAEggs,PlasticWrap: 1
NaturalChicken&SageBreakfastSausage,CageFreeExtraLargeGradeAAEggs,OrganicLemon: 1
NaturalChicken&SageBreakfastSausage,CageFreeExtraLargeGradeAAEggs,OrganicWholeStrawberries: 3
NaturalChicken&SageBreakfastSausage,OrganicWholeStrawberries: 18
NaturalChicken&SageBreakfastSausage,OrganicWholeStrawberries,OrganicLemon: 2
NaturalChicken&SageBreakfastSausage,PlasticWrap: 20
NaturalChicken&SageBreakfastSausage,OrganicLemon: 20
NaturalChicken&SageBreakfastSausage,OrganicLemon,PlasticWrap: 1
VitaminDWholeMilk,PlasticWrap: 1
VitaminDWholeMilk,CageFreeExtraLargeGradeAAEggs: 3
VitaminDWholeMilk,OrganicLemon: 3
VitaminDWholeMilk,OrganicLemon,OrganicWholeStrawberries: 1
VitaminDWholeMilk,GrapefruitSparklingWater: 6
VitaminDWholeMilk,OrganicWholeStrawberries: 21
GrapefruitSparklingWater,OrganicLemon: 2
GrapefruitSparklingWater,OrganicLemon,OrganicWholeStrawberries: 1
GrapefruitSparklingWater,CageFreeExtraLargeGradeAAEggs: 10
GrapefruitSparklingWater,CageFreeExtraLargeGradeAAEggs,OrganicWholeStrawberries: 2
GrapefruitSparklingWater,PlasticWrap: 21
GrapefruitSparklingWater,OrganicWholeStrawberries: 39
CageFreeExtraLargeGradeAAEggs,OrganicLemon: 12
CageFreeExtraLargeGradeAAEggs,OrganicLemon,OrganicWholeStrawberries: 1
CageFreeExtraLargeGradeAAEggs,OrganicLemon,PlasticWrap: 4
CageFreeExtraLargeGradeAAEggs,PlasticWrap: 37
CageFreeExtraLargeGradeAAEggs,OrganicWholeStrawberries: 60
OrganicLemon,PlasticWrap: 37
OrganicLemon,OrganicWholeStrawberries: 55
PlasticWrap,OrganicWholeStrawberries: 1
```

Figura 18: Output Example 2