ESE589 Course Project II

FP Growth Algorithm  
November 14, 2019  
Professor Alex Doboli  
Kenneth Ng (110211727)  
Kurt Butler (109587387)

# Abstract

In this project, we implement the Frequent Pattern Growth (FP growth) algorithm using the Python programming language. We discuss our implementation of the algorithm, and we evaluate its performance using the benchmark data sets supplied by the UCI Machine Learning Repository [2]. The implementation of the FP Growth algorithm was successful for 16 test cases from the machine learning repository. We consider the operation of the algorithm as it represents causal relationships underlying frequent item sets, and visualization of those relationships in the binary case. The structure of the report is as follows: the first section covers some background on the FP Growth algorithm. The second section describes the software design. The following section uses a sample dataset to prove the functionality of our program. Once performed on the sample set, the 16 benchmark cases were evaluated. Some observations on the datasets were made based on these results, and we propose some considerations for further study.

Contents

[Abstract 1](#_Toc24745007)

[Introduction 3](#_Toc24745008)

[Design of the Software 4](#_Toc24745009)

[Table 1. Given Dataset [1] 4](#_Toc24745010)

[Table 2. Frequency Table [1] 4](#_Toc24745011)

[Table 3. Filtered Dataset [1] 4](#_Toc24745012)

[Figure 1. FP Growth Tree 5](#_Toc24745013)

[Table 4. Conditional Pattern Base [1] 5](#_Toc24745014)

[Table 5. Frequent Patterns [1] 6](#_Toc24745015)

[Algorithm Implementation 7](#_Toc24745016)

[Figure 2. Decoded Dataset 7](#_Toc24745017)

[Figures 3, 4. Ordered Itemset Ordered Table 7](#_Toc24745018)

[Figure 5. Frequent Pattern Base for a\_2 7](#_Toc24745019)

[Figure 6. Frequent Pattern Generated for {a\_2, c\_2} 8](#_Toc24745020)

[Figure 7. Pseudocode for Frequent Pattern Generation 8](#_Toc24745021)

[Experiments 9](#_Toc24745022)

[Table 6: UCI Benchmark Evaluation 9](#_Toc24745023)

[Case Study: Wine Data Set 9](#_Toc24745024)

[Plot 1: Relative Frequency of Items 10](#_Toc24745025)

[Plot 2: High Support Relationships 11](#_Toc24745026)

[Plot 3: Confidence Diagram 11](#_Toc24745027)

[Discussion 12](#_Toc24745028)

[Plot 4: Execution Time vs Min-Sup Ratio 12](#_Toc24745029)

[Plot 5: Norm Execution Time vs No. of Unique Items 12](#_Toc24745030)

[Plot 6: No. Unique Items vs No. Attributes 13](#_Toc24745031)

[Conclusion 13](#_Toc24745032)

[References 13](#_Toc24745033)

# Introduction

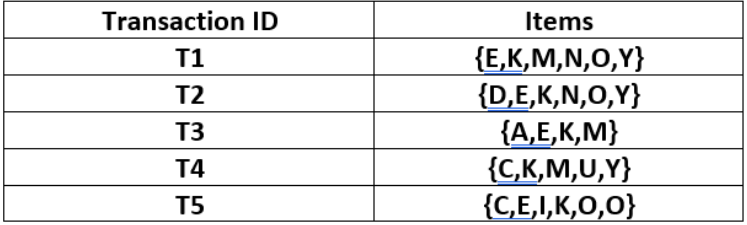
Companies in marketing and consumer analysis care about finding patterns in the buying habits of their customers. Knowledge of such patterns enables the seller to make better product recommendations that improve the experience and satisfaction of the customer. The goal of frequent pattern analysis is to discover such patterns in data that can be used to make predictions about the customer based on partial data.

In this project, we study the FP growth algorithm. FP growth belongs to a class of algorithms intended to recognize and find frequent associations between items in a data set. When items frequently co-occur, we organize them into a *frequent item* set. The presence of one item of a frequent data set greatly increases the probability that a customer will purchase other items in the frequent data set. As the size of a frequent item set increases, so does its predictive power; the occurrence of more items in a frequent item set usually increases the conditional probability of observing any one item in the customer’s basket, as more items are added. In symbols, is usually observed, where is the event of seeing the item of the item set in a basket.

A common alternative to FP growth is the Apriori algorithm, which generates association rules by continuously searching the data set. The advantage of FP growth over apriori is the generation of conditional FP trees which reduce time complexity from (as in apriori) to .

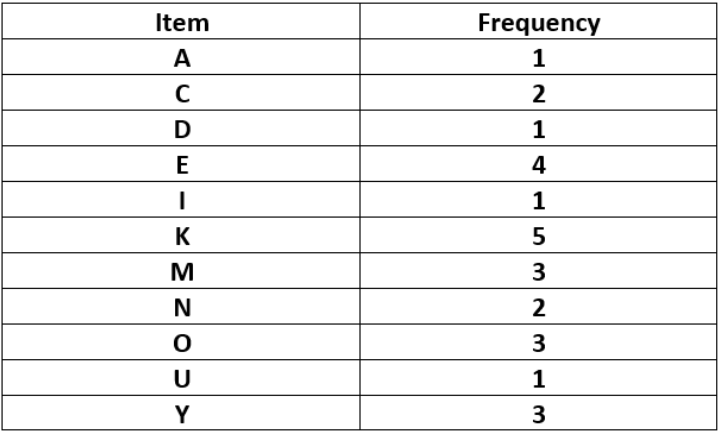
# Design of the Software

The FP growth algorithm identifies item sets that have a high correlation between each item. To visualize the algorithm, the following example will be taken from Alind Gupta’s article on FP Growth [1]. The following dataset is used in their example.



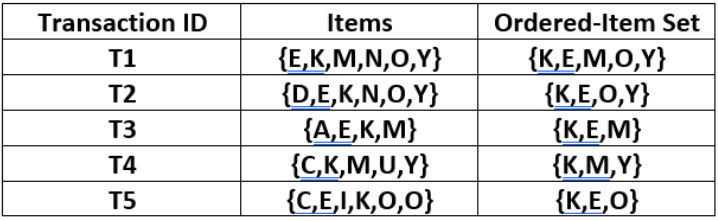
### Table 1. Given Dataset [1]

With the dataset and minimum support as the threshold, the frequencies of each element within each column are taken and stored in a hash table. The item name is the key and the frequency is the data. The resulting frequency table is described in Table 2.



### Table 2. Frequency Table [1]

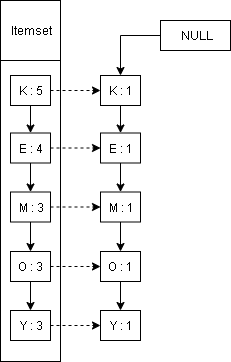
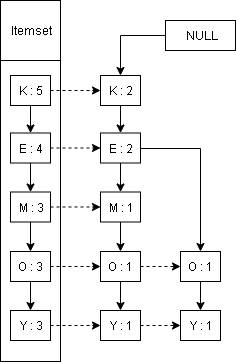
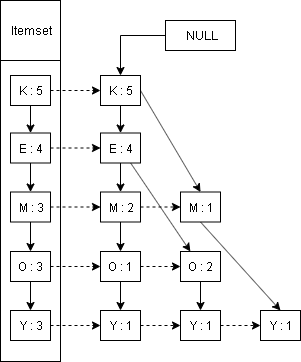
Each frequency in the hash table compared to the threshold and filtered as the first frequent pattern. The dataset is then filtered to only include the elements in the hash table and ordered by frequency in descending order. In the example, the minimum support is 3 and the items that meet this condition are {K, E, M, O, Y} with ordered frequencies {5, 4, 3, 3, 3}. The resulting filtered dataset labelled “ordered-item set” is shown in Table 3.



### Table 3. Filtered Dataset [1]

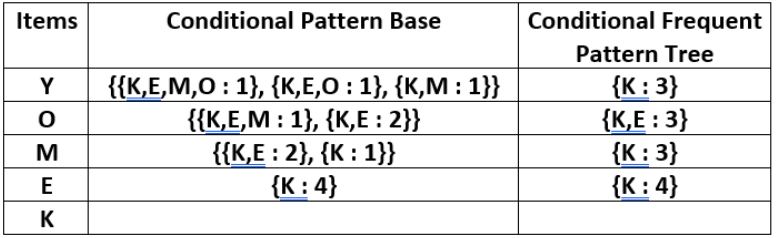
Using the filtered dataset, we build two structures: a tree for the union of all transactions, and a hash table for the current items containing the nodes with the total count of each item. Each node in the tree contains the name for the element, the frequency, and references to neighbors. The initialized Frequent Pattern (FP) Growth tree and hash table are described below in Figure 1.

### Figure 1. FP Growth Tree

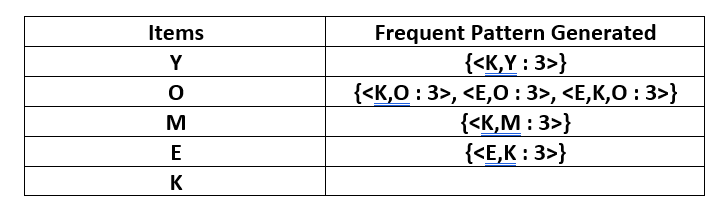
(a) Insert T1 (b) Insert T2 (c) Finished Insertion

With the completed initial tree, the conditional pattern base and conditional frequent pattern tree are constructed. Using the hash table, the paths to each node with the same item in the itemset is generated by accessing the parent creating the conditional pattern tree. From the conditional pattern tree, the union of all the items in the conditional pattern base is taken and compared to the threshold. For example, the union of the conditional pattern base for item Y is {K, E, M, O} with support counts {3, 2, 2, 2}. Only item {K} is above the minimum support, 3, and is preserved in the conditional frequent pattern tree.



### Table 4. Conditional Pattern Base [1]

The conditional frequent pattern tree is constructed and if there is more than one path, the process is repeated. Once there is only one path, the combinations of each frequent pattern and the item are outputted. For example, item O has the frequent pattern tree {K, E: 3}. The resulting frequent patterns are {{K, O} {E, O} {E, K, O}}: 3 as seen in Table 5.

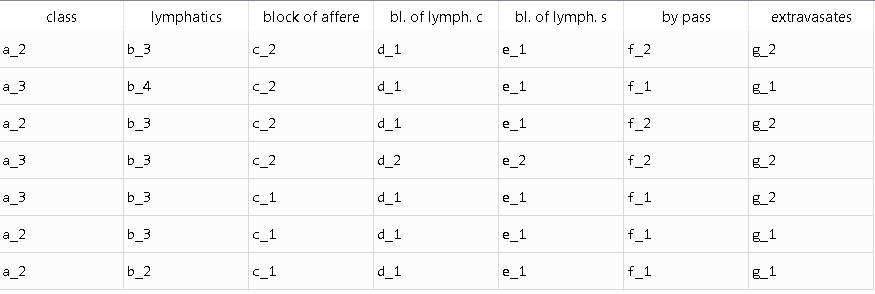


### Table 5. Frequent Patterns [1]

The benchmark structures are slightly different than the example because the example uses lists and the benchmarks are matrices. However, the example could be a subset of the benchmark. When the items are initially filtered using the minimum support, the resulting dataset will have lists like the dataset used in the example.

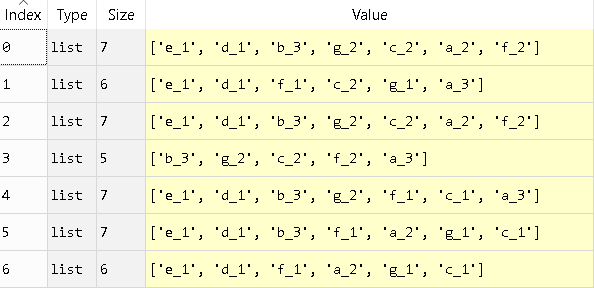
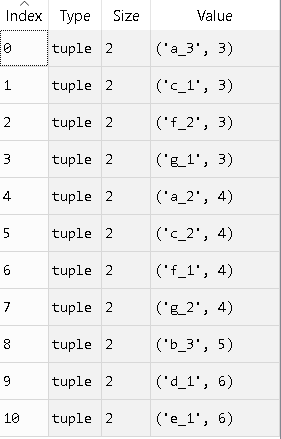
# Algorithm Implementation

Using a small sample to test the algorithm, Figure 2 shows the dataset once the elements have been decoded. The dataset was decoded to give each category a unique identifier. The events for the FP Growth algorithm can be recorded in either the console or an output file.



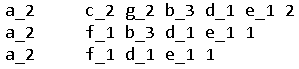
### Figure 2. Decoded Dataset

With the new dataset, the dataset is filtered with the minimum support set to 2, shown in Figure 3, and ranked according to the frequency and by name as shown in Figure 4.

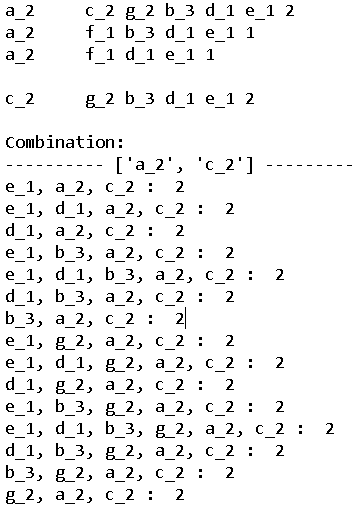
### Figures 3, 4. Ordered Itemset Ordered Table

With the ordered itemset, the FP Growth tree and conditional pattern base is generated. The following Figure 5 shows the frequent pattern base for item ‘a\_2’. Following the item ‘a\_2’ are the paths to different ‘a\_2’.



### Figure 5. Frequent Pattern Base for a\_2

The union of the frequent pattern base for ‘a\_2’ are {c\_2: 2, f\_1: 2, g\_2: 2, b\_3: 3, d\_1: 4, e\_1: 4} and all items meet the minimum support. Therefore, the conditional pattern base is the set for the conditional frequent pattern tree. The paths are still greater than one, so the next frequent pattern set is {a\_2, c\_2}. The threshold is still met, and the conditional frequent pattern tree has one path. The list of frequent patterns is then generated for every combination shown in Figure 6. The pseudocode for the implementation of the FP Growth algorithm is shown in Figure 7.



### Figure 6. Frequent Pattern Generated for {a\_2, c\_2}

##

# inputs: tree – frequent pattern tree or subtree

# a – current frequent pattern

# min\_sup – threshold

# outputs: result – frequent pattern

##

procedure FPGrowth(tree, a, min\_sup)

result = []

if tree has one path

while tree.head is not none

result.extend(result) #make copy of result

for each set in result copy

append tree.name

else

for all keys in tree.hash #tree.hash is hash table

if tree.hash [key] >= min\_sup

build condpatbase list #cond freq pattern base

if condpatbase is not null

a.append(key)

create condpattree #cond freq pattern tree

FPGrowth(condpattree, a, min\_sup)

a.pop()

### Figure 7. Pseudocode for Frequent Pattern Generation

# Experiments

Using the UCI Machine Learning Repository [2], we consider 16 test data sets to evaluate our implementation. Because each data set is attributed to different contributors, we will refer to each data set using its name within the UCI database. For each benchmark, we consider the computer memory usage and execution time of the program operating on each data set, as a function of data set and choice of *Min Sup*.

When we convert benchmark data into an item set, we are interested in the number of unique traits or items. For categorical attributes, each possible value contributes to the number of possible items. For numerical data, we discretize values by binning them into two Boolean categories: High and Low. Each Boolean then contributes two items per attribute. Thus, the number of item sets (***No. Items***) in this model corresponds to the number of unique nodes in the FP tree. In the table below, ‘Cat’ is shorthand for ‘Categorical.’

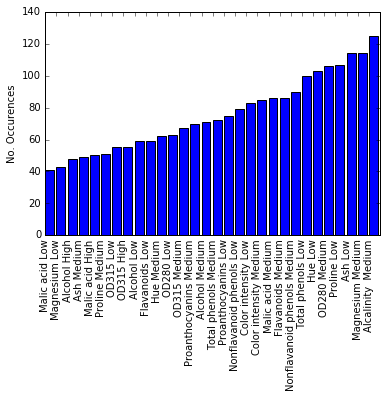
### Table 6: UCI Benchmark Evaluation

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **No.** | **Data Set Name** | **Attribute Types** | **No. Samples** |  | **No. Attrib.** | | **No. Items** | **Min Sup** | **Exe. Time (seconds)** | **Memory Usage (Bytes)** |
| 1 | Lymphography | Cat | 148 |  | | 18 | 63 | 30 | 3.187 | 121,245,696 |
| 2 | SPECT Heart | Cat | 267 |  | | 22 | 46 | 68 | 0.847 | 18,874,112 |
| 3 | SPECTF Heart | Cat | 267 |  | | 44 | 279 | 70 | 2.838 | 14,163,712 |
| 4 | Glass | Real | 214 |  | | 10 | 32 | 20 | 1.760 | 122,388,480 |
| 5 | Stock Portfolio | Real | 315 |  | | 12 | 53 | 100 | 6.091 | 124,039,168 |
| 6 | Soybean (Small) | Cat | 47 |  | | 35 | 151 | 30 | 6.115 | 29,589,248 |
| 7 | Echocardiogram | Cat, Int, Real | 132 |  | | 12 | 34 | 50 | 0.963 | 129,785,856 |
| 8 | Flare(Data 1) | Cat | 323 |  | | 10 | 40 | 100 | 2.390 | 23,473,920 |
| 9 | Flare(Data 2) | Cat | 1067 |  | | 10 | 48 | 320 | 10.738 | 26,652,416 |
| 10 | Seeds | Real | 210 |  | | 7 | 24 | 10 | 0.822 | 80,056,320 |
| 11 | Wine | Int, Real | 178 |  | | 13 | 24 | 40 | 1.624 | 63,008,768 |
| 12 | Fertility | Real | 100 |  | | 10 | 26 | 20 | 0.522 | 29,134,592 |
| 13 | CSM Dataset 2014 and 2015 | Int, Real | 217 |  | | 12 | 38 | 80 | 2.051 | 126,918,656 |
| 14 | Cloud (Data 1) | Real | 1024 |  | | 10 | 50 | 200 | 8.068 | 63,680,512 |
| 15 | Cloud (Data 2) | Real | 1024 |  | | 10 | 50 | 200 | 8.103 | 66,596,864 |
| 16 | Flags | Cat, Int | 194 |  | | 30 | 304 | 100 | 8.611 | 125,001,728 |

## Case Study: Wine Data Set

We will illustrate the usage of the algorithm for the Wine data set. When processed, we calculate the number of occurrences (relative frequency) for each item. Out of the 178 transactions, we have 24 “items.” Opting to label the items by their data set names, we show them in Plot 1. Since Wine is a numerical data set, we note that these results were achieved by partitioning each parameter into three subcategories (High, Low, and Medium) which subsequently become items in our interpretation of the algorithm.

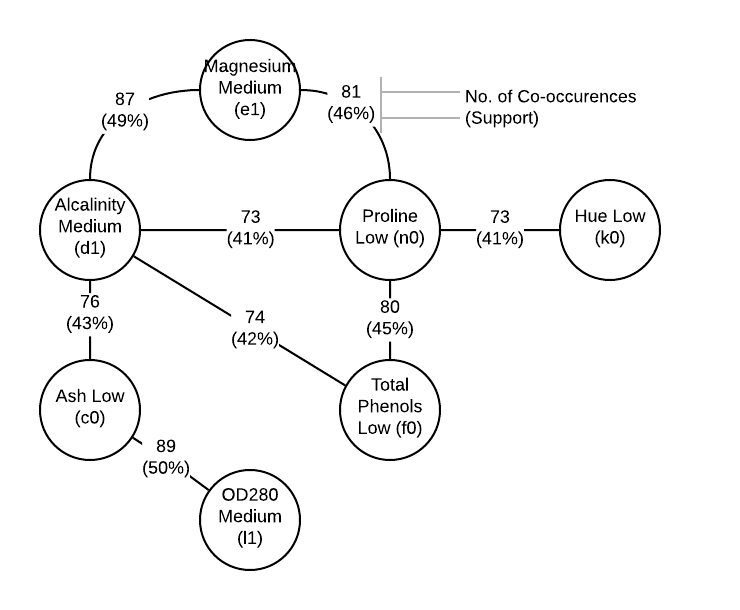
### Plot 1: Relative Frequency of Items



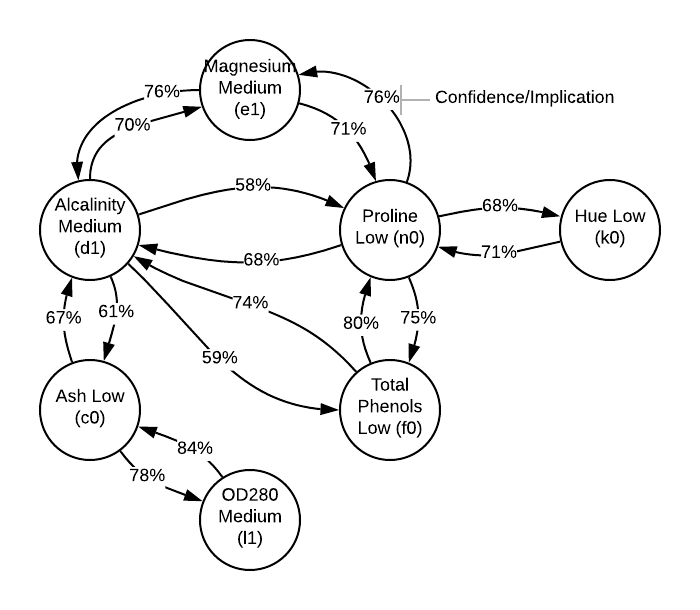
Given the FP growth algorithm, we can retrieve rules that predict frequent item sets. Selecting a threshold value of 73 (for ease of visualization), we generate many singleton items sets, which agree with Plot 1, and a few 2-item sets. Namely, for the most frequent seven items in Plot 1, we can calculate the support and confidence for relations that pass the threshold criteria. No three-item rules were produced at this threshold.

We calculate the *support* of a rule by the percentage of all transactions that appear in the intersection of the two sets in a binary rule, and we define *confidence* of by computing the conditional probability . A graph of nodes representing items and edges representing threshold-passing *binary* frequent items is shown in Plot 2. In Plot 3, we visualize the confidences that are induced by the supported relations in Plot 2. For convenience, both the FP-tree label and data set label appear on each node. For each arrow in the confidence plot, the percentage indicates what the probability of a transaction that has the item represented at the origin node also has the item of the end node (i.e. if and only if ).

### Plot 2: High Support Relationships



### Plot 3: Confidence Diagram



# Discussion

Firstly, we consider the dependence of execution time and memory usage against our selection of parameters, namely threshold or minimum support. We define the *min-sup ratio* to be Min-Sup over the No. of samples in the data set (or transactions). We observe that they are inversely related, in that low thresholds produce long execution times. For this reason, our selection of min-sup to enable efficient pruning is essential to the performance of the algorithm.

### Plot 4: Execution Time vs Min-Sup Ratio

After importing a data set and storing the transformed/discretized data as an itemset, we consider the number of unique items as a measure of data set complexity. We define the n*ormalized execution time* to be the execution time multiplied by the min-sup ratio and we use the number of unique items as measured, shown in Table 6.

### Plot 5: Norm Execution Time vs No. of Unique Items

The correlation between these quantities is visualized in Plot 5. We notice that generally in the data set, normalized time increases with the number of unique items in a data set. The threshold values heavily influence the execution time and a reasonable threshold must be chosen for different datasets, causing variation within the measurement. Thus, after examination we found this trend to be not-significant to analyzing algorithm performance.

Next, we consider the dependence of no unique items vs the number of attributes in the original data set. Note that in general, an attribute of the data only contributes roughly 5 items to a data set on average, as shown in Plot 6. For this reason, we compute complexity by as it is directly given by the data set.

### Plot 6: No. Unique Items vs No. Attributes

In general, we observed no useful or interpretable trend in memory usage as a function of the data set or threshold. All data sets had a multi-megabyte offset, due to the usage of Python libraries such as Pandas, but the additional memory usage saw no interesting trend.

In terms of data visualization, the support and confidence diagrams are effective for visualizing binary relationships, but higher complexity relations don’t admit such visually simple forms. In the binary case, we have two possible rules, and , and both can be visualized with arrows in a digraph. For example, a ternary relationship between items A, B and C requires *six* different cases of analysis, and arrows can’t be drawn to represent them. In general, we require numbers to describe a clique of items, and further work needs to be done to effectively visualize them.

# Conclusion

We implemented the FP Growth algorithm and observed its scalability in time and memory. The choice of threshold is important to achieving reasonable execution time, as is the number of unique items considered in the data set. Further revisions of this program would seek to reduce the memory usage by relying on less memory intensive library usage and would explore other representations of item-set data that may improve performance.

# References

1. Gupta, Alind. “ML | Frequent Pattern Growth Algorithm.” *GeeksforGeeks*,   
   [www.geeksforgeeks.org/ml-frequent-pattern-growth-algorithm](https://www.geeksforgeeks.org/ml-frequent-pattern-growth-algorithm/). Accessed November 10 2019.
2. D. Dua, C. Graff. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information and Computer Science.
3. J. Han, M. Kamber, J. Pei. (2012). *Data Mining: Concepts and Techniques* (3rd ed.). Waltham, MA: Morgan Kaufmann Publishers.