****

**Final Project Report**

by

Platinum Justice Array

**Team Members:** Jimmy Griffiths and Walter Proulx

December 8, 2017

Table of Contents

[1. Introduction 1](#_Toc500523959)

[2. Background 2](#_Toc500523960)

[2.1 Explaining Data Definitions 2](#_Toc500523961)

[2.2 Explaining Apriori 3](#_Toc500523962)

[3. Data Architecture 5](#_Toc500523963)

[3.1 Data Structure 5](#_Toc500523964)

[3.1.1 Items Data Structure (Linked List) 5](#_Toc500523965)

[3.1.2 Items Translation (Dynamic Array) 6](#_Toc500523966)

[3.1.3 Items Array and Transactions Array 6](#_Toc500523967)

[3.1.4 1st and 2nd Subset Array 7](#_Toc500523968)

[3.2 Data Format 9](#_Toc500523969)

[4. Experiment Detail 10](#_Toc500523970)

[4.1 Experiment Platform Detail 10](#_Toc500523971)

[4.2 Experiment Detail 10](#_Toc500523972)

[4.3 Experiment Result 10](#_Toc500523973)

[5. Discussion and Conclusion 12](#_Toc500523974)

[6. Reference 14](#_Toc500523975)

# Introduction

In a technologic age where massive amounts of data is being retrieved, it is no longer practical for a human being to evaluate the data by hand. There is simply too much data for one person to read, even without interpreting what the data means. The procedure of finding meaning in data is now a much more complicated process involving multiple steps from selecting the data that is valid to a user, preprocessing the data or splitting it up into groups, transforming the data to a tangible form, data mining this tangible form to find patterns, and then interpreting what these patterns mean [1]. When all of these steps are performed, the hope is that whoever is reading the data can then optimize or learn from the results. The magnitude in which this knowledge can help varies, but has shown to have the potential to be an extremely powerful tool.

One essential part of this whole procedure is association mining, which falls under the multi-dimensional data mining process. Data mining involves many different fields which helps prepare the data for interpretation. These fields can involve artificial intelligence, machine learning, pattern recognition, expert systems statistics, visualization, and database management systems [1]. Association mining heavily relies on the pattern recognition field as it is looking for repetitions in the data. From these repetitions, the certain associations can be made. For example, if a transaction typically involves items 1 and 2, there is a strong association that item 3 will be involved as well.

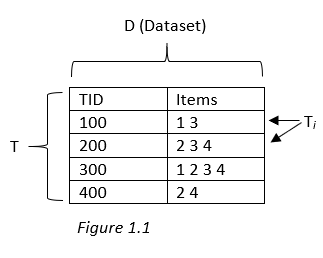
Generating these associations can be hard with larger data sets which require an algorithm to evaluate due to sheer volumes of data that can now be produced. The Apriori algorithm is one way to generate these associations [2]. This algorithm can take repetition, pruning the loosely associated data, giving a user what appears to be the most important data for making associations.

Association mining has a very practical application that can help in various situations. Sports, retail, and even banks can use these associations to make predictions that can help a business. Advertisements can be personalized through association mining. The potential benefits of associations mining (as well as the disadvantages) are unknown and can prove to be a great help or hindrance to society depending on the situation.

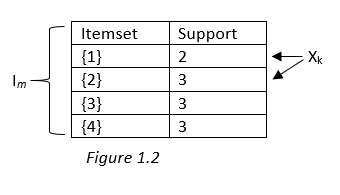
# Background

## Explaining Data Definitions

When dealing with Apriori, there are several important pieces of information that we need. We need to know how many items, or *i* there are total. Like, for example in a grocery store, or an electronics store. In a grocery store, *i* could represent a pound of apples, a watermelon, a liter of water, etc. In an electronics store, *i* could be a T.V., an ethernet cord, a keyboard, etc. We store all of those items in a set *I*. Now, when a customer makes a purchase or a transaction, we store that in a set *Ti*. So, if a customer buys a T.V. and a keyboard, we have a *T­1*= {T.V., keyboard}. Each one of those transactions we store in a set called *T*. So *T* = {*T1*, *T2*…*Tn*}. Now, we will store multiple sets of those sets in another set, *D*. From all of this data, we can make associations about what people buy, in the electronics store, and predict what people will buy together, or not together. Figure 1.1 shows all of this. There are several *Ti*sets of items, which are a subset of a greater *T* set, which is then a subset of a greater *D* set.



So, how do we do all that? Let us make a new set called *Im* that has sets of *k* length. Now, we will have several sets *X*, where each *X* is of length *k*, and contains sets of items from *T*. Therefore, *X* is a subset of *T*. Then, we will compare *X*, to *T*, and see how many times *X* appears in *T*, and by extension *D*. This is the support for *X*. If the support for *X* passes the minimum support threshold given by the user or programmer, then it will stay. Otherwise, it will be pruned. Now looking at Figure 1.2, we can see this with a solid example.



We are seeing how many times 1, 2, 3, and 4 by themselves appear in the data given in Figure 1.1. 1 appears 2 times, 2, 3, and 4, appear 3 times. If the minimum support threshold is defined as 2, then all sets pass, and are furthermore known as *frequent itemsets*. If the minimum support threshold is defined as 3, then only 2, 3 and 4 are *frequent itemsets*.

Now, with all that information, we can begin to create an association rule, *X Y*. Where *Y* is another itemset. Firstly, both *X* and *Y* must be *frequent itemsets*, meaning they both pass the minimum support threshold. Then, *X Y* = Meaning that *X* and *Y* must have no element in common. Then we need to know the *confidence* of *X Y*. The *confidence* is the support of / the support of *X*. Like support, we will have a *minimum confidence threshold*. If we pass the minimum confidence threshold, along with everything else being true, then we know we have an association rule that *X Y*.

## Explaining Apriori

Apriori() {

1) L1 = {frequent 1-itemsets};

2) for (k = 2; Lk−1 ! = ∅; k++) {

// generate new candidate itemsets

3) Ck = apriori\_gen(Lk−1);

// calculate the support for all k-candidate-itemsets

4) for (t = 0; t < the number of total transactions; t++) {

// candidates contain in t

5) Ct = subset(Ck, t);

6) for (c ∈ Ct)

7) c.count++;

}

8) Lk = {c ∈ Ck | c.count ≥ minimum support};

}

9) return Lk;

}

where :

Lk is a set of frequent k- itemsets that satisfy the minimum support threshold. Each member of this set has two fields: itemsets and support count.

Ck is a set of k- candidate-itemsets that is potentially frequent. Each member of this set has the same two fields of the Lk.

Deciphering the important, repeating patterns in massive datasets can be hard and far too much work for a human to perform. Apriori is a step forward to efficiently translating data from massive random collections to counted repetitions. (The counting of repetitions, in this case, is called calculating support.) To make meaning out of the repetitions, the Apriori algorithm will prune any repetitions that do not meet a baseline expectation (or minimum support). For example, if a subset is only repeated twice, it might not be reported as a meaningful repetition. The point of pruning data, in this case, is to find strong connections between data which can then be translated into a greater meaning.

For this algorithm to find all meaningful repetitions, two actions must happen: calculating all possible subsets, and pruning of subsets that do not meet the minimum support.

To generate all possible subsets efficiently, the algorithm starts with all subsets consisting of 1 item and works its way up until it reaches the largest subset it can create. The *apriori\_gen()* function accomplishes this task of generating all possible 1 subsets (and 2 subsets, etc, until n is reached) by looking at similar elements. If a subset of three elements is to be created, it will look for two subsets of two with one similar element ( {1,3}, {1,4} for example) and generate a new subset ( resulting {1,3,4} ). The algorithm will keep going until it has found all subsets that can be made.

After the subsets have been created, it goes through each of these subsets and calculates the support (repetitions) of each subset. Once all the supports have been calculated, it takes out (prunes) all subsets that do not meet the minimum support.

The algorithm will repeat this process until every subset has been systematically generated. At the end of the algorithm, every subset that made it past pruning is returned giving the user all relations that are meaningful.

# Data Architecture

## Data Structure

**ItemsList (Linked List)**

int itemNumber  
int length

insert  
delete

**Items Translation (Dynamic Array)**

int itemNumber

**Items Array (Dynamic Array)**

bool itemExists

searchIndex

**Transactions Array (2D Dynamic Array)**

Items Array

initializeItems  
findItem

**1st Subset Array - SubList (Linked List)**

int[ ] itemIndex  
int support

insert  
prune  
initializeItemset

**2nd Subset Array – SubList (Linked List)**

int[ ] itemIndex  
int support

insert  
prune  
initializeItemset

### 3.1.1 Items Data Structure (Linked List)

The Items data structure will be comprised of integers referencing the item numbers in all transactions. The first step to creating our proceeding data structures is retrieving this list of the items that exist in all transaction, the length of the linked list when finished, and the number of transactions that occur. This linked list will not allow repeats to occur when inserting. Not allowing duplication makes it easy to scan through a transactions file and find all the numbers that exist within the file. Every number can be inputted, and if that number already exists then it will not be added to the linked list. For efficiency, these numbers will be inserted sequentially. This will help the construction of the next data structure.

Using a linked list for this particular task is necessary. Arrays are not possible as the number of items to be inserted is unknown. A queue and stack could be used for this task, but sequentially searching through these data structures would be less efficient than a linked list ordering the number. If the average case of a queue and stack require the whole data structure to be searched for duplicates, it makes more sense to use a sorted linked list which has a chance of needing only to search a part of the linked list, while the worst case requires the whole list to be searched.

The advantage to using a linked list is the ability to sort the item numbers for transplanting the data into a dynamic array, as well as the ability to have an easily expandable data structure. Unfortunately, the disadvantage of sequentially searching through the list is still there. Luckily with the data sets that are presented, there will only be a total of 1000 items to search through, which, with today’s technology, takes a very small amount of time. With datasets this small, other search algorithms do not offer significantly faster search times. This makes the need for using a more complex data structure unnecessary.

### 3.1.2 Items Translation (Dynamic Array)

Once the Items data structure has been generated, the Items Translation dynamic array can be made. The dynamic array of integers (item numbers) can be made with the length of the linked list. The array will be filled up with the contents of the Items data structure. The purpose of the Items Translation data structure is to associate an index of the array with a specific item. This array will now act as the translation between an index and an item. For the purposes of this algorithm, it is faster to just reference an index rather than searching through all the items whether it be sequential search or binary search.

The advantage to using a dynamic array which will already be sorted is the ability to use different search algorithms when loading up the Transactions Array. Each item in the data set is still an item number, and therefore it must be translated with this data structure. Using binary search for this task might help to slightly reduce translation times. There is also the advantage of all other data structures using indexes rather than search algorithms. Even fast, memory-intensive search algorithms cannot outspeed a reference to a specific index.

As mentioned above, this whole process is reliant on referencing indexes, making all other data structures currently available infeasible. Due to this constraint, a lot of memory is required. Arrays are massive sequential chunks of memory rather than split up references like a linked list, which could prove to be problematic on devices with little memory available. Sacrificing the amount of available memory will boost speed but be dangerous for large datasets as there might not be enough memory available.

### 3.1.3 Items Array and Transactions Array

Now that the Items Translation array has been set up, the Transactions Array (TA) can be initialized. The TA will have a length of a number of transactions *t*. Each index of this array will reference the data structure Items Array. Items Array consists of a bool array with the length of the Items (and Items Translation) data structure. Each index is now associated with an item number (which can be translated with the Items Translation data structure), therefore a Boolean value of 0 and 1 are needed only to indicate whether or not an item does not or does exist. The TA is now a 2-dimensional array with the y-axis referencing a transaction, and the x-axis referencing the item based on the index.

Since all items are now translated to indexes with a Boolean value, the need for search algorithms has been eliminated. In terms of performance, this is a huge boost. In terms of memory, this is a huge downfall as the Transactions Array is massive. A huge, sequential block of data is required to store this array. Devices that have this memory available will do fine, but in other environments, this will cause substantial problems for the program. If the data can be stored, there is the benefit of not having to access the data file more than twice: once for finding items and file length, the other for translating the transactions and storing them. Again, this is a performance boost, but it comes at the cost of memory.

In terms of structure, a dynamic array is the only data structure that works in this situation. Using any other data structures such as link lists or binary search trees still requires searching, which is not superior in search time. These structures could help reduce the amount of memory used, as they would only be storing the value of the item as opposed to all items for each transaction with a Boolean value. Structuring the process this way has forced a focus on efficiency and less on memory. It is an unfortunate but necessary tradeoff with these data structures.

### 3.1.4 1st and 2nd Subset Array

The last data structures necessary are the 1st and 2nd Subset Arrays. Both of these data structures will have a linked list referencing a node containing a support integer, and an integer array. The integer array will hold the item numbers in index form. (This is as opposed to storing the items as item number forms. Since there is an Items Translation data structure, items can be stored by index reference and translated back if needed. Referencing by index is much more efficient, therefore this translation is necessary.) The dynamic integer array is necessary as subsets will start out with 1-itemsets, and increment to 2-itemsets and so on requiring a dynamic array. Since subset sizes will always be known, the length of each dynamic array does not need to be calculated. Creating two identical data structures is necessary, as later itemsets require the previous itemset as a reference for making the new one. These data structures will need the ability to insert new subsets as well as the ability to prune subsets that do not meet the minimum support threshold. Additionally, the data structure needs the ability to initialize its subsets given a similar data structure.

Since these subset arrays are constantly changing, a linked list seemed like the most logical choice. It is unknown how many subsets can be generated from a dataset, making arrays impractical. Additionally, the pruning process makes deletion essential, which would also be harder to perform in an array. Other data structures such as binary search trees are unnecessary as there is no need to search these structures. Each element in the linked list will have to be handled individually, so there is no benefit from being able to easily search subsets. The dynamic arrays stored in this linked list do have the ability to be generated as the length is always known. There is not a particular benefit to using a dynamic array. A linked list could also achieve this task as there will never be a need to reference a specific index. Both a linked list and array will have to go through sequentially. The only nice thing about using an array over a linked list is an array will be conceptually easier to keep track of specific elements and to debug. Linked lists store values with pointers, making it hard to figure out what data a program is pointing to. (It is harder to debug a memory reference than an index number.) Other alternatives could have been used, but for the sake of this program, arrays are being used for conceptual simplicity.

|  |
| --- |
| **ItemsList (Linked List)** |
| # \*mHead: Node  # \*mTail: Node  # mLength:  int  Node  # mItemNumber: int  # \*mNext: Node |
| + ItemsList()  + ~ItemsList()  + getItem(index: int): int  + getLength(): int  + setItem(index: int, item: int)  + clear()  + display()  + insert(item: int): bool  + isEmpty(): bool  + isExist(item: int): bool  + removeAt(index: int): int  + operator[](index: int): int |

|  |
| --- |
| **SubList (Linked List)** |
| # \*mHead: Node  # \*mTail: Node  # mLength:  int  Node  # mItemSubset: int[]  # mSupport: int  # \*mNext: Node |
| + SubList()  + ~SubList()  + getItem(listIndex: int, subsetIndex: int): int  + getLength(): int  + getSubset(listIndex : int) : int\*  + getSupport(): int  + setItem(listIndex: int, subsetIndex: int, item: int)  + setItemSubset(listIndex: int, subset: int\*)  + setSupport(listIndex: int, support: int)  + clear()  + display(sublistLength : int)  + incrementSupport(listIndex: int)  + insert(itemSubset: int[], subsetLength: int): bool  + isEmpty(): bool  + isExist(itemSubset: int[]): bool  + outputToFile(subsetLength : int, out : &ofstream, itemTranslation : int[])  + removeAt(listIndex: int): int\*  + operator[](listIndex: int): int[] |

## Data Format

The expected input for the data will come in forms of sets of transactions, with items being represented by integers. Furthermore, the items in each transaction are already sorted. The text files will have names such as Tx.NyK.DzK.txt. Tx is the average length of the transaction, Ny is the number of different items, in thousands, hence the K. And Dz is the total number of transactions in the file, in thousands. Therefore, T5.N1K.D5K means that the file contains 5000 transactions, with an average transaction length of 5, and the range of items is 0-999.

Some lines for the file of T5.N1K.D5K could look like…

1 410 780

47 89 700 810 943

189 231 441 678

The output of the data we find will be in a text file simply known as Output.txt. We will take each set that is frequent of each length, and display the support of each set. For example, an output text file could look like

{1} support: 7

{2} support: 4

{1, 2} support: 3

And so on. This would display all Ik and support for each.

# Experiment Detail

# Experiment Platform Detail

Provide information on the platform[[1]](#footnote-1) where the experimentation was conducted. For example:

16GB DDR4

Intel Core i5-7500

3.40GHz

64-bit OS

# Discussion and Conclusion

Overall the data structures used for the Apriori algorithm greatly enhanced the speed at which the dataset could be traversed. By storing the dataset in a single two-dimensional array, supports could be calculated with great speed. The main thing stopping this program from running faster is the decision to use a linked list as part of storing subsets. Sequentially searching through a linked list has proved to be more detrimental than first determined, and it has forced several workarounds to occur, so the program could be optimized. As will be discussed later, this is the key factor that has stopped this program from processing the larger datasets as the time it takes to manage these larger data sets grows exponentially.

Adding to the number of distinct items can affect the program in two ways. The first way it can slow the program down is in the first phase in which all items are retrieved. It doesn’t take long to traverse the file, so the impact of adding more distinct items is minimal. Adding items unfortunately has a side effect of adding more distinct subsets. This was troublesome as the more subsets that needed to be generated also made for a longer linked list. By the nature of this program, it will sequentially search for subsets on the linked list. Even with minor optimizations (such as cutting off a search when a result is known to be non-existent), only so much can be done to make the search go faster. No number of shortcuts can jump over the underlying problem that a sequential search is still necessary. As mentioned before, the more subsets that are created, the more time it takes to traverse these subsets. Since there are nested for loops, this time doesn’t increase linearly, but exponentially. Out of all the things that can be changed in a transaction, this has the biggest impact.

Changing the average transaction length has little to no effect on this program. Due to the nature of how transactions are stored, making an average transaction length longer only creates a bigger array to take up more memory. Storing the values in an array will make the time grow in a linear rate. The transaction length will not have a huge impact even when calculating support factors, as the array indexes are being referenced, so no search algorithm is required. Since indexes are determining the speed as which individual transactions are searched, there is little one can engineer to make this an inefficient means for searching.

Giving more transactions to deal with will naturally make the program slower. Each subset must search through the whole array, so however many transactions were added will make every search through the array that much longer. The impact isn’t significant, but it will be noticeable. There is little that can be done about this, as there is no way to skip searching the last couple of transactions that occur. Compared to adding more distinct items, adding more transactions has a small impact.

Upping the minimum support threshold is one of the most obvious ways to make the program faster. During the pruning process more, subsets will be taken out. For this program, this is a huge boost in performance. When subset lists start getting to long, the performance tanks. By keeping the subsets in check and pruning the unnecessary ones, the program will run fast. The single operation that takes the most time is searching through these linked lists to get to the subsets. Once this can be evaded, such as increasing the minimum support threshold, then the program can run much faster.

In review, this program exceeded our expectations. The average or expected time for this program to run on a small dataset was somewhere around five minutes. We were able to push this time down from over 200 minutes to a sheer 48 seconds with a minimum support threshold of two. This was an increase in efficiency of over 25,000%. Even with some minor optimizations, that time initially went from 200 minutes to just over 7 minutes. The final result is very satisfying considering how a couple of minor tweaks to the program could increase efficiency by this much. The final time exceeded our expectations for speed for the smaller datasets.

That being said, there is most certainly some changes that can be made to this program to boost efficiency. Designing the process to mostly use arrays most certainly helped efficiency. It doesn’t appear that this process took an excessive amount of memory, especially as a standalone program. It could be optimized to not take up as much memory but for the first iteration of the program speed was prioritized over memory consumption. The Linked List did help conserve memory, but in a relatively small amount compared to the massive transaction array. Using Linked Lists provided efficient use of memory, but it ended up bottlenecking the speed of the program by a problematic amount. If the program were truly efficient in time it would have to find a way to avoid using a Linked List. If memory availability was low, then we would have to consider optimizing the way the transactions array holds the data file. It might be more efficient to only hold half the file depending on the size, which would cut a large part of the memory used. There are many other ways this program could be optimized, but the transactions array is the biggest problem in terms of memory.

If this project could be repeated, there would have to be a reconsidering of the SubList item. There was too much time spent going through these SubList objects because of the forced sequential search. Either there would have to be a way to only focus on the sub lists that are important (which would make the related functions much more complicated) or a dynamic array would have to replace the linked lists, and some sort of algorithm could be developed to generate a well predicted size for the dynamic array. Both options would help yield better results. Moving on from this glaring flaw, all other parts of this program were implemented well. Calculating supports and generating subsets could be improved upon, but more time would be needed to figure out what excessive repetition is occurring when the program is running. The item translation array can stay the same. Choosing to go through the file initially is a choice that we would repeat in the future, as this doesn’t take much time to find all the items that actually exist. It is worth saving the space of that the transactions array takes up by not just assuming from the file name that there are a set number of items. Also, this program will work with any file in a similar format named anything. It is not reliant on this naming convention making it more versatile for different situations.

# Reference

[1] W. K. Chen, “Data Structures for Selective Association Mining,” PH.D. dissertation, CACS, Univ. LA, Lafayette, LA, 2005.

[2] W.K. Chen, “CHAPTER 1: Introduction,” Ph.D. dissertation,

1. To check the CPU speed, in the command prompt, enter msinfo32, and all the needed information are available in the pop-up window. [↑](#footnote-ref-1)