CSE304 Introduction to Data Mining
Group #2
Contact point XXX@unist.ac.kr

Assignment #1

1 Apriori Algorithm

1.1 Introduction

We live in an era of immense information, with huge amounts of data created every day. It's very important to find relationships and patterns in this huge amount of data. One approach to do this is with a problem called the 'frequent pattern mining problem'. This problem finds sets of items that show up often in the data. For example, this technique can be used in recommendation systems that suggest products to users based on their preferences. By analyzing the items that a user frequently purchases with, the system can discover patterns and suggest other items with these patterns.

Older algorithms for frequent pattern mining, like AIS or SETM, weren't very efficient at solving this problem efficiently. These algorithms tried to check every possible candidate set, which used a lot of computer resources and was too slow, especially with big and complex data. To solve these problems, the Apriori algorithm [1]. was developed.

The main idea of the Apriori algorithm is to use 'support', a value that indicates how often a specific set of items occurs within the dataset, to remove groups of items that don't show up often before they get checked. This greatly cuts down the number of candidate sets we need to look at. The Apriori algorithm filters the candidate sets before we start checking them, which means there's less data to check and the overall performance is better.

By using and testing the Apriori algorithm, we can see that it solves the frequent pattern mining problem much faster and more effectively than earlier algorithms. Also, this helps us understand not just the good points of the Apriori algorithm but also its weak points, showing us where we need to make it better. By continually analyzing and improving like these algorithms, we can create more advanced and efficient data mining techniques.

1.2 Problem Statement

Suppose that a series of transactions are given. For a specific itemset(combination of item(s)), the proportion of transactions that contain the itemset is called support. The goal of the algorithm is to find all itemsets whose supports are greater than minsup threshold(i.e. frequent).

Theorem 1 (Apriori principle)**.** *If an itemset is frequent, then all of its subsets are also frequent.*

Proof. Suppose a frequent itemset A has an infrequent subset B. Since the transactions that contain A must contain B, the support of A is not greater than that of B. Therefore, if B is infrequent, then A must be infrequent as well, which contradicts to the initial condition. □

1.3 Algorithm Description and Analysis

Algorithm 1 Apriori Algorithm

```
1: $L_1 = \{\text{large 1-itemsets}\}$;
2: for $(k = 2; L_{k-1} \neq 0; k++)$ do
3:     $C_k$ = apriori-gen($L_{k-1}$);
4:     for transactions $t \in \mathcal{D}$ do
5:         $C_t$ = subset($C_k, t$);
6:         for candidates $c \in C_t$ do
7:             c.count++;
8:         end for
9:     end for
10:    $L_k = \{c \in C_k \mid \text{c.count} \geq \text{minsup}\}$
11: end for
12: return $\cup_k L_k$
```

L_k is a set of frequent k-itemsets(itemsets with k items), and C_k is a set of candidates for L_k. After L_1 is initially set, candidates C_k are generated with L_{k-1}. Next, for each transaction, count variables of the candidates included in the transactions are incremented. Then candidates whose count variables have higher values than minsup threshold are inserted into L_k. The process above is iterated for $k = 2, 3, ...$ until L_{k-1} is empty. Finally, the union of all L_ks becomes the desired answer.

Algorithm 2 Candidate Generation

```
1: procedure APRIORI-GEN($L_{k-1}$)
2:     insert into $C_k$                                              ▷ Self-join
3:     select $p.\text{item}_1$, $p.\text{item}_2$, ..., $p.\text{item}_{k-1}$, $q.\text{item}_{k-1}$
4:     from $L_{k-1}\ p$, $L_{k-1}\ q$
5:     where $p.\text{item}_1 = q.\text{item}_1$, ..., $p.\text{item}_{k-2} = q.\text{item}_{k-2}$, $p.\text{item}_{k-1} < q.\text{item}_{k-1}$
6:
7:     for itemsets $c \in C_k$ do                                    ▷ Pruning
8:         for $(k-1)$-subsets $s$ of $c$ do
9:             if $(s \notin L_{k-1})$ then
10:                delete $c$ from $C_k$
11:            end if
12:        end for
13:    end for
14:    return $C_k$
15: end procedure
```

The algorithm to generate candidates consists of two parts, **Self-Join** and **Pruning**.

- **Self-join**: All k-itemsets then can be made by combining two itemsets in L_{k-1} are inserted into C_k. Two itemsets whose items are identical except the last one are selected. Then k-itemsets are formed by joining the common subset and last items of each itemset. Lastly, the merged k-itemset is inserted into C_k.
- **Pruning**: All itemsets in C_k are examined. If one of the subset of the itemset is not included in C_{k-1}(i.e. the subset is infrequent), then the itemset is excluded from C_k.

After self-join and pruning, C_k becomes the desired candidate set for L_k.

1.4 Example

TID	itemset
1	$\{a\}$
2	$\{a,b,d\}$
3	$\{a,d\}$
4	$\{b,c\}$
5	$\{a,b,c,d\}$
6	$\{c,d\}$

Figure 1: Example transactions

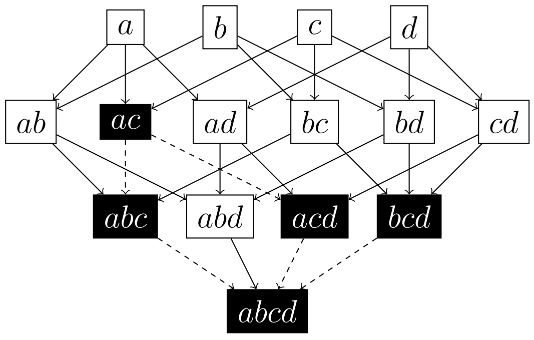


Figure 2: Graph of candidates

Example 1. *Suppose that the minsup for the example is* $1/3$*. All 1-itemsets are included in at least 2 transactions, which means they are all frequent. Next, all 2-itemset candidates are generated from 1-itemsets, and their supports are examined. As a result,* $\{a, c\}$ *turns out to be infrequent. This prunes not only* $\{a, c\}$ *but all supersets(*$\{a, b, c\}$*,* $\{a, c, d\}$*, and* $\{a, b, c, d\}$*). Similarly, 3-itemset candidates are generated(*$\{a, b, d\}$ *and* $\{b, c, d\}$*) and* $\{b, c, d\}$ *are pruned. 4-itemset candidates cannot be made since there is only 1 frequent 3-itemset(*$\{a, b, d\}$*), which makes the examination process halt. Finally, all frequent k-itemsets are merged and returned.*

1.5 Advantages and Disadvantages

In an exhaustive search, we must generate candidates for all possible combinations. However, Apriori algorithms reduce the number of these candidates, which is a good way to reduce the search time. Also, it is a intuitive algorithm, so it is easy to understand and implement.

However, this is possible when there is sufficient minimum support and when the entire dataset is not very large. If minimum support is low, almost all candidates must be considered, which again consumes a lot of time. Also, because algorithms must explore the entire dataset multiple times, larger datasets will also consume a lot of time.

1.6 Experiment

We implemented the algorithm using the basic functions of Java and verified its performance through three datasets. The implementation code and dataset were all put on this GitHub link[1]. The implementation was based on the paper.

[1] `https://github.com/hhg000726/dm`

The first dataset groceries.csv consisted of 9835 transactions with an average of 4.41 items with 169 types of items. The second dataset test1.csv of similar size is a change in the format of files on this link[2]. It consisted of 7501 transactions with an average of 5.95 items with 137 types of items. The third much larger dataset test2.csv is a change in the format of files on this link[3]. It consisted of 23178 transactions with an average of 22.65 items with 4006 types of items.

All experiments were conducted in a Windows 10 environment with AMD Processor 4.20 GHz and 16 GB of RAM.

Minimum support	0.09	0.23	0.0005	0.0011
Associations	10	101	1001	11390
Time(ms)	82	646.6	4427.6	38825.2

Figure 3: Apriori algorithm to groceries.csv

The experimental results for the first set, grocereis.csv, are shown in Figure 3. The algorithm spent more time with low minimum support and more associations to search for, but it increased to a little less than proportional to the number of associations.

We should think about keeping the candidate sets larger as there are more associations to look for. It can be predicted that only the appropriate candidate sets have been left so that the time does not increase too much. Otherwise, if all combinations are considered, time will increase exponentially.

Minimum support	0.152	0.0415	0.018	0.0033
Associations	10	100	1000	10241
Time(ms)	72.2	439.8	2262.4	19818.6

Figure 4: Apriori algorithm to test1.csv

The experimental results for the first set, test1.csv, are shown in Figure 4. In the case of small minimal support and many associations, it is reduced to nearly half the time of the first dataset.

Minimum support	0.054	0.0295	0.0113
Associations	10	100	1000
Time(ms)	237.4	4738.4	316230.2

Figure 5: Apriori algorithm to test2.csv

The experimental results for the first set, test1.csv, are shown in Figure 5. It can be seen that time has increased significantly as the size of the data grows. In particular, it can be seen that the more associations you need, the more time you increase beyond proportion.

1.7 Conclusion and Future Directions

In all datasets, the algorithm is expected to perform better than an exhaustive search. In particular, small datasets showed slightly less time than proportional to the number

[2]`https://www.kaggle.com/datasets/d4rklucif3r/market-basket-optimisation`
[3]`https://archive.ics.uci.edu/dataset/352/online+retail`

of associations. Through this, we thought that the solution to the problem could be sufficiently improved through the improvement of the algorithm.

One of the big problems with algorithms is that the entire data still has to be explored multiple times. As we saw in the third dataset, this is fatal for large datasets. The following algorithms that can improve this will be discussed.

2 FP-growth Algorithm

2.1 Introduction

The FP-Growth algorithm [3] can overcome the limitations of the Apriori algorithm. The Apriori algorithm suffers from inefficiency due to multiple scans of data and the generation of numerous candidate item sets. To address this, the FP-Growth algorithm provides a method to efficiently extract frequent item sets by scanning the entire data only twice.

The FP-Growth algorithm consists of two steps. In the first step, the algorithm scans the data to calculate the frequency of each item. Based on this, it selects only the high-frequency items to generate frequent item sets. The FP-Tree generated during this process is a specialized data structure that compresses and sorts the data, removing unnecessary data to increase processing speed.

In the second step, through a process called FP-Growth, the algorithm uses the constructed FP-Tree to extract the required frequent item sets. This process utilizes the structure of the FP-Tree to directly extract item sets without separately generating candidate sets, thereby significantly reducing time and memory usage.

By implementing the FP algorithm directly, we can verify its efficiency over the Apriori algorithm. However, it's also important to analyze the limitations of this algorithm.

2.2 Problem Statement

We solve the same problem as the one solved by the Apriori algorithm.

2.3 Algorithm Description and Analysis

Algorithm 3 FP-Tree Structureing Algorithm

```
1: $L$ = {large 1-itemsets};                                  ▷ Main-Loop
2: $root$ = new Node with nothing
3: for transaction $T$ in $D$ do
4:     $P$ = sorted $\{item | item \in T \text{ and } item \in L\}$
5:     if $P$ is not empty then
6:         $insert\_tree(P, root)$
7:     end if
8: end for
1: $p = P[0]$, $P$ = sublist of $P$ from index 1               ▷ Insert-Tree
2: if Tree $T$ has child $p$ then
3:     child.support++;
4: else
5:     Add new child $N$ with item $p$ and support 1 to $T$
6: end if
7: if $P$ is not empty then
```

```
8:     $insert\_tree(P, child)$
9: end if
```

Algorithm 3 describes the process of creating an FP tree. Main-loop performs the following task. First, L is initialized to a set of large 1-itemsets. It represents a set of single items with a support value greater than or equal to minimum support among all single items found in the data set. The root is initialized to a new node with nothing. This node is the root node of the FP tree. For each transaction T, do the following. Sorts and stores items in the transaction that are more than minimum support in P. If P is not empty, call the insert_tree function and insert P into the FP tree.

The insert_tree function does the following. Specify the first item in P as p, and remove it from P. If T has a child p, increase that child's support value. Otherwise, create a new child, N, and add it to T as a new node with item p and support 1. If P is not empty after this, recursively invoke the insert_tree function to insert the remaining entries of P for the child node. The algorithm is stored in a special tree structure called FP-Tree, which is used to identify frequent patterns in next step.

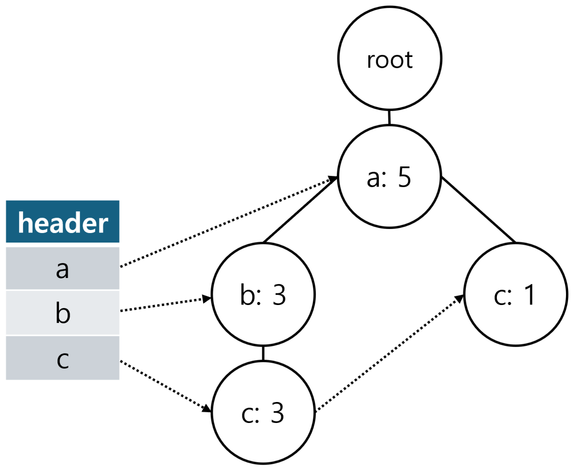
Algorithm 4 FP-Growth Algorithm

```
1: procedure FP-GROWTH($T, \alpha$)
2:     if Tree $T$ has single path then
3:         $s$ = minimum support in path
4:         for combiantion $c$ of nodes in path do
5:             generate pattern ($c$ following $\alpha$ with support $s$)
6:         end for
7:     else
8:         for node $\alpha_i$ in header table do
9:             $\beta = \alpha_i$ following $\alpha$
10:            construct $\beta$'s conditional pattern base
11:            construct $\beta$'s conditional FP-tree $Tree_\beta$
12:            if $Tree_\beta \neq \emptyset$ then
13:                call fp-growth with $Tree_\beta$, $\beta$
14:            end if
15:        end for
16:    end if
17: end procedure
```

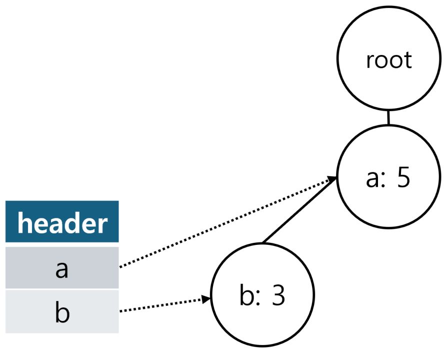
Algorithm 4 is used to find patterns that occur frequently in a dataset using FP trees. The process is as follows. FP-Growth takes the given FP tree T and the minimum support α as factors. If the FP tree T has a single path, it calculates the minimum support in the path and generates a pattern for each node combination of path with α.

Otherwise, if the FP tree T does not have a single path, repeat the following for each node in the header table. Concatenate the current node α_i and β. Then creates a conditional pattern base for β. It consists of the items in the path where **β** appears except for βs. Then creates a conditional FP tree for β, called $Tree_\beta$. It is created using a conditional pattern base. If $Tree_\beta$ contains at least one item, recursively call FP-Growth to find the pattern for $Tree_\beta$ and β.

2.4 Example



(a) FP-Tree



(b) After FP-Growth once

Figure 6: Example

Example 2. *Assume you have transaction data of (a, b, c), (a, d), (a, b, c), (a, c, d, e), (a, b, c, e) Let's think about finding associations that appear 3 or more. Like Apriori, initially count the number of each item as follows: a: 5, b: 3, c: 4, d: 2, e: 2. Here, only items of 3 or more are considered and the FP-Tree is constructed as shown in (). Afterward, FP-Growth is applied to pattern grow each element, and if only a single path remains, all combinations are generated, otherwise, it continues to be called recursively.*

2.5 Advantages and Disadvantages

By using FP tree, there is no need to create candidate sets and unnecessary information scans can be reduced. The entire dataset only needs to be scanned twice. In addition, information is compressed and stored in one tree, so it is much faster in time and less space can be used. This compressed and practical information contributes to fast pattern mining.

However, once we construct an FP tree, it will not be easy to modify when new data comes in and it cannot be analyzed until the FP tree is completed. In addition, using a recursive method while mining an FP tree will still require a lot of programming techniques to implement and analyze large data sets.

2.6 Experiment

All experiments were conducted by running the FP-Growth algorithm, which was made with Java code, in the same environment and dataset as in the Apriori algorithm.

The experimental results for the first set, test1.csv, are shown in Figure 7. We can see that the algorithm solved the problem well in a sufficient time for all minimum supports before the error occurred. This is a much better result than Apriori. For larger minimum support, the program failed to manage them properly and caused errors as the amount of memory used and the recursive function increased.

Minimum support	0.09	0.023	0.005	0.0011	0.0002	0.0001
Associations	10	100	1001	11390	790072	memory error
Time(ms)	72.8	125.4	179.6	284.4	907.2	-

Figure 7: FP-Growth algorithm to groceries.csv

Minimum support	0.152	0.0415	0.0118	0.0033	0.0009	0.0003	0.0002
Associations	10	100	1000	10241	116943	891974	memory error
Time(ms)	62.8	116.4	165.6	275.2	545.8	1114	-

Figure 8: FP-Growth algorithm to test1.csv

The experimental results for the first set, test1.csv, are shown in Figure 8. Like the first data set, we can see that the algorithm solved the problem in a sufficient amount of time for minimum support where the algorithm did not cause memory errors.

Minimum support	0.054	0.0295	0.0113	0.005	0.003	0.002	0.001
Associations	10	100	1000	10426	89265	1318569	memory error
Time(ms)	227.8	409.2	1147.2	2356.6	4748.6	13367.8	-

Figure 9: FP-Growth algorithm to test2.csv

The experimental results for the first set, test1.csv, are shown in Figure 9. Although it took longer than previous datasets, we can still see that the execution before the memory error was resolved within a short period. However, as the minimum support became smaller and the number of associations to be searched increased, we could see a significant increase in time.

2.7 Conclusion and Future Directions

The FP-Growth algorithm always performed better on our datasets than the Apriori algorithm. In particular, it could be seen that the increasing time was not as great as the data size increased compared to the Apriori algorithm. However, it was still possible to see a noticeable increase in time as the data size increased, and it is not expected to be enough to be applied to the enormous amount of datasets that need to be dealt with in practice. One of the problems we expect is that if the shape of the created FP-Tree is not suitable for the problem, much improvement cannot be achieved. In addition, configuring and recursing FP-Trees for large datasets can lead to a significant memory burden.

To improve the FP-Growth algorithm, there're several algorithm have been proposed. FP-Bonsai algorithm [2] is one of those which improve the performance of FP-Growth by using the ExAnte data-reduction technique. There're another algorithm NONORDFP Algorithm [4] which conduct faster traversal, faster allocation by using the Trie data structure to store the large amount of data.

And also, we use several java internal method and library to improve the performance of some operations in above two algorithms. For example, we used HashMap structure instead of HashTree, and it performs the containing operation in O(1) time complexity. Like this way, if there's a development of something method to perform useful operation like making the conditional pattern base.

However essentially, the main usage and background of the association rule mining is finding the meaningful rule in the large amount of dataset such as transaction, and making the recommendation system to user, but despite these improvement efforts the FP-Growth isn't used in many current recommend system. As described above, the amount of data becomes vast day by day, but the FP-Growth isn't as fast and still ineffective in the huge dataset.

Because of these limitations, collaborative filtering algorithm which make a recommendation by user-based or item-based is mainly used, and recently, with the development of deep learning, there're many personalized-recommendation system using the deep learning technology. In these changes, FP-Growth alrogithm doesn't seem like a good aption.

References

[1] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. 1215: 487–499, 1994.

[2] F. Bonchi and B. Goethals. Fp-bonsai: the art of growing and pruning small fp-trees. In *Advances in Knowledge Discovery and Data Mining: 8th Pacific-Asia Conference, PAKDD 2004, Sydney, Australia, May 26-28, 2004. Proceedings 8*, pages 155–160. Springer, 2004.

[3] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2):1–12, 2000.

[4] B. Rácz. nonordfp: An fp-growth variation without rebuilding the fp-tree. In *FIMI*. Citeseer, 2004.

Name	Sign	Individual Contribution	Percentage
홍승표	홍승표	Writing - Original Draft, Formal analysis	33%
임형근	임형근	Validation, Conceptualization, Methodology	33%
Hong Hyeongi	홍현기	Software	33%