**Final Report**

**DESIGN AND ANALYSIS OF ALGORITHM**

TOPIC: A SURVEY OF ITEMSET MINING

Lecturer: **Mr. Nguyen Chi Thien**

Students: **Ho Vinh Tuong - 520K0091**

**Nguyen Pham Phu Thinh - 520V0012**

**Chibuike Timothy Benedict - 519K0078**

Course: **24**

**Final Report**

**DESIGN AND ANALYSIS OF ALGORITHM**

TOPIC: A SURVEY OF ITEMSET MINING

Lecturer: **Mr. Nguyen Chi Thien**

Students: **Ho Vinh Tuong - 520K0091**

**Nguyen Pham Phu Thinh - 520V0012**

**Chibuike Timothy Benedict – 519K0078**

Course: **24**

# ACKNOWLEDGEMENT

We send our most sincere thanks to Mr. Nguyen Chi Thien for always supporting and accompanying us to complete the mid-term report of Design and Analysis of Algorithms in the best way. We always hope that you will always be the teacher that accompanies and gives our students the best knowledge so that they can go out and have the most solid knowledge. Once again, thank you very much for your time.

The team would also like to thank the Faculty of Information Technology and the school for creating conditions both physically and mentally so that the group could complete the mid-term report in the best way.

We would like to thank all the teachers who have supported us in the best way. We sincerely thank you!

# THE REPORT IS COMPLETED AT TDT UNIVERSITY

I hereby declare that this is the product of our own project and under the guidance of Mr. Nguyen Chi Thien. The research contents and results in this topic are honest and have not been published in any form before. The data in the tables for analysis, comments and evaluation are collected by the author himself from different sources, clearly stated in the reference section.

In addition, the project also uses a number of comments, assessments as well as data from other authors, other agencies and organizations, with citations and source annotations.

If I find any fraud, I will take full responsibility for the content of my project. Ton Duc Thang University is not related to copyright and copyright violations caused by me during the implementation process (if any).

*HCMC, December 3rd 2022*

*Authors*

*Ho Vinh Tuong*

*Nguyen Pham Phu Thinh*

*Chibuike Timothy Benedict*

# CONFIRMATION AND ASSESSMENT SECTION

**The evaluation of lecturer**

HCMC, December 3rd, 2022

**The evaluation of examiner**

HCMC, December 3rd, 2022

# SUMMATION

This report describes 6 algorithms for itemset mining, which is an important subfield of data mining.

For each algorithm, we clarify the logic to solve the problem in Python, their own asymptotic time and space complexity.

After that, we display graphics of running time and memory usage.

Contents

[ACKNOWLEDGEMENT 1](#_Toc120995417)

[THE REPORT IS COMPLETED AT TDT UNIVERSITY 2](#_Toc120995418)

[CONFIRMATION AND ASSESSMENT SECTION 3](#_Toc120995419)

[SUMMATION 35](#_Toc120995420)

[INTRODUCTION 36](#_Toc120995421)

[I. ALGORITHMS 38](#_Toc120995422)

[A. APRIORI 38](#_Toc120995423)

[B. APRIORI-TID 40](#_Toc120995424)

[C. ECLAT 42](#_Toc120995425)

[D. Hmine 45](#_Toc120995426)

[E. FP-Growth 47](#_Toc120995427)

# INTRODUCTION

Introduced in 1993 by Agrawal and Srikant as large itemset mining, the task of discovering itemsets is nowadays called frequent itemset mining (FIM).

Itemset mining consists of discovering interesting and useful patterns in transaction databases. It is used for discovering groups of items (itemsets) that appear frequently together in transactions made by customers. In other words, that is discovering groups of attribute values frequently co-occurring in databases. There are domains of numerous application such as bioinformatics, text mining, product recommendation, e-learning, and web click stream analysis. As a result, itemset mining is a well-known.

For example, given a database of customer transactions. One can analyze that database and discover that many customers buy taco shells with peppers (a frequently itemset). Then he can understand customer behaviors, which is applied for strategic marketing decisions.

A transaction database can look like:

|  |  |
| --- | --- |
| TID (transaction ID) | Transaction |
| T1 | a, c, d |
| T2 | b, c, e |
| T3 | a, b, c, e |
| T4 | b, e |
| T5 | a, b, c, e |

Let us explain more about FIM and transaction database:

I = {i1, i2, i3, …, im} is a set of items

D = {T1, T2, T3, …, Tn} is a set of transactions such that , q is a unique identifier.).

From the below table, we have I = {a, b, c, d, e} and D = {T1, T2, T3, T4, T5}. T1 represents a customer who has bought the item a, c and d. The others are the same.

X is a set of items, which is a subset of I. We denote the length of X (the number of items in X) is k (or |X|). Each X has its own support (or absolute support), which is the number of transactions containing X. More generally, . For instance, sup({a, b}) is 2 because they appear in T3 and T5.

Another quantity we need to consider is minsup. It is the frequency of an X that users want to determine, thereby it is set by users . Let’s say minsup = 3, FIM will discover all X appearing not less than 3 times in the transaction database such as {a} : 3, {b} : 4, {c} : 4, {e} : 4, {a, c} : 3, {b, c} : 3, {b, e} : 4, {c, e} : 3, {b, c, e} : 3.

FIM is an enumeration problem. It enumerates all patterns that meet the minsup. Thus, there is always a single correct answer. However, FIM may be inefficient.

Consider that if there are m distinct items in transaction database, there are 2m – 1 possible itemsets. With the minsup = 1, FIM will generates a search space of 2m – 1 itemsets, which is more than the size of the data.

To avoid exploring the search space of all possible itemsets and process as efficiently as possible. An efficient FIM algorithm will use search space pruning techniques.

That means: for any itemsets X and Y such that X is subset of Y, it follows that sup(X) is greater or equal to sup(Y). Thus, if an itemset is infrequent, all its supersets are the same. Then we do not need to consider them.

FIM algorithms differ in:

* whether they use a depth-first or breadth-first search
* the type of database representation that they use internally or externally
* how they generate or determine the next itemsets to be explored in the search space
* how they count the support of itemsets to determine if they satisfy the minimum support constraint.

In this report, we introduce 6 famous algorithms. They are Apriori, AprioriTID, FP-Growth, Eclat, H-Mine and LCM.

All of them have the same input and output.

|  |  |  |
| --- | --- | --- |
| Algorithm | Type of search | Database representation |
| Apriori | Breadth-first | Horizontal |
| AprioriTID | Breadth-first | Vertical |
| Eclat | Depth-first | Vertical |
| FP-Growth | Depth-first | Horizontal |
| H-Mine | Depth-first | Horizontal |
| LCM | Depth-first | Horizontal |

# ALGORITHMS

## APRIORI

In this algorithm, the database is used for calculate the support of each k-size itemset.

Starting from the 1-size list of itemsets, C1, we get L1, a list of minsup-satisfied itemsets. The next step is generating C2 by combining each pair of 2 itemsets of L1. Note that, we only combine a pair if and only if two itemsets has the same items without the last one. Consequently, we can avoid getting infrequent or duplicated itemsets and the size of the new itemsets is k+1.

For example, L has {1, 2, 3}, {1, 2, 4} and {1, 3, 4}. After generating, C has {1, 2, 3, 4}, a 4-size itemset. The combinations of {1, 2, 3} and {1, 3, 4} or {1, 2, 4} or {1, 3, 4} are duplicated.

Repeat until L is empty

The result of Apriori is the union of all Lk

The code below is my basic implementation:

'''

The Apriori algorithm

input: path - txt file, minsup - integer number

output: res - list of frequent itemsets

'''

**def** **apriori**(path, minsup):

dbs, C = readTransDB(path) # C is the 1-size item set

L = getFrequentItemSets(C, dbs, minsup) # the frequent 1-size item set

res = [] # result

**while**(L):

res.append(L) # store the previous frequent itemsets L

C = candidateItemSets(L) # join step

L = getFrequentItemSets(C, dbs, minsup) # prune step

**return** res

The sub functions:

'''

functions reads txt file to retrieve data

input: input - path of txt file representing transaction database

output: listOfTIDs - a list of transactions; listOfItems - list of distinct items

'''

**def** **readTransDB**(input):

**with** open(input, 'r') **as** f:

listOfTIDs = [line.strip().split(' ') **for** line **in** f]

listOfItems = sorted({item **for** tid **in** listOfTIDs **for** item **in** tid})

**return** listOfTIDs, listOfItems

'''

function removes less-support-frequent itemsets from C

input: C - list of itemsets; transDB - list of transactions; minsup - an integer number

output: L - list of frequent itemsets

'''

**def** **getFrequentItemSets**(C, transDB, minsup):

L = []

**for** itemSet **in** C:

count = **0**

**for** tid **in** transDB:

**if** set(itemSet).issubset(set(tid)):

count += **1**

**if** count >= minsup:

L.append(itemSet)

**return** L

'''

function candidates list of itemsets from L

input: L - list of frequent itemsets

output: C - list of increase-in-size-1 itemsets

'''

**def** **candidateItemSets**(L):

C = []

**for** i **in** range(len(L)):

**for** j **in** range(i+**1**, len(L)):

**if** L[i][:-**1**] == L[j][:-**1**]:

C.append(sorted(set(L[i]).union(set(L[j]))))

**return** C

## APRIORI-TID

AprioriTID is a variation of the Apriori algorithm. It was proposed in the same article as Apriori as an alternative implementation of Apriori. It produces the same output as Apriori. But it uses a different mechanism for counting the support of itemsets after the first pass.

Rather, the set is used for this purpose.

Each member of corresponding to transaction t is

C and L are calculated as same as Apriori.

To calculate, we need the previous and the latest C. For example, is generated from C2 and .

For each itemset in C, we determine 2 subsets. One does not contain the last item and one does not contain the near-end item. If both are in the transaction in the previous , that itemset will be in the corresponding TID in the new .

For instance, consider {1 2} and {1 3} in C. The previous is:

|  |  |
| --- | --- |
| TID | Set |
| T100 | {{1}, {3}, {4}} |
| T200 | {{2}, {3}, {5}} |

With {1 2}, we have {1} and {2}. Both are not subset of T100 or T200 so the transactions of new will not have them.

With {1 3}, {1} and {3} are born. They only appear in T100, there for the new is:

|  |  |
| --- | --- |
| TID | Set |
| T100 | {{1 3}} |

The code below clarifies how to get

'''

function creates the candidate itemsets for passing

input: prevPassC - the previous passed itemsets, C - the candidate itemsets

output: passC - the candidate itemsets for passing

'''

**def** **getPassC**(prevPassC, C):

passC = []

**if** (C):

k = len(C[**0**]) - **1** # last index

**for** t **in** range(len(prevPassC)):

Ct = []

**for** c **in** C:

a = c[:-**1**]

b = c[:k-**1**] + c[k:]

**if** a **in** prevPassC[t] **and** b **in** prevPassC[t]:

Ct.append(c)

**if** Ct:

passC.append(Ct)

**return** passC

The AprioriTID:

'''

The aprioriTID

input: path - txt file, minsup - int

output: res - list of frequent itemsets

'''

**def** **aprioriTID**(path, minsup):

C\_, C = readTransDB(path)

L = getFrequentItemSets(C, C\_, minsup)

res = []

**while** (L):

res.append(L)

C = candidateItemSets(L)

C\_ = getPassC(C\_, C)

L = getFrequentItemSets(C, C\_, minsup)

**return** res

Read file:

'''

functions reads txt file to retrieve data

input: input - path of txt file representing transaction database

output: listOfTIDs - a list of transactions; listOfItems - list of distinct items

'''

**def** **readTransDB**(input):

**with** open(input, 'r') **as** f:

listOfTIDs = [[list(item)

**for** item **in** line.strip().split(' ')] **for** line **in** f]

listOfItems = []

**for** tid **in** listOfTIDs:

**for** item **in** tid:

**if** item **not** **in** listOfItems:

listOfItems.append(item)

**return** listOfTIDs, listOfItems

## ECLAT

The main idea of Eclat is the algorithm scans the transaction database to get list of distinct items. For each item, Eclat finds list of TID they appear.

Eclat also needs to find C and L.

For getting Lk, from the Ck-1, we simply count the number of TID that an itemset appear. If it is greater than or equal to the minsup, we append it to Lk.

To generate Ck, we combine each pair of itemsets in Lk-1, the condition is that pair has the same items except for the last one. The interest is that we also get the union of their TID lists. Thus, we have a pair of itemset and its list of TID.

Repeat until L is empty

The result of Eclat is the union of all Lk .

From the transaction database:

|  |  |
| --- | --- |
| TID | Transaction |
| T1 | {a, c, d} |
| T2 | {b, c, e} |
| T3 | {a, b, c, e} |
| T4 | {b, e} |
| T5 | {a, b, c, e} |

C1 will be:

|  |  |
| --- | --- |
| Item | TID-set |
| a | {T1, T3, T5} |
| b | {T2, T3, T4, T5} |
| c | {T1, T2, T3, T5} |
| d | {T1} |
| e | {T2, T3, T4, T5} |

Based on the number of TID, if the minsup is 3, L1 will be:

|  |  |
| --- | --- |
| Item | TID-set |
| a | {T1, T3, T5} |
| b | {T2, T3, T4, T5} |
| c | {T1, T2, T3, T5} |
| e | {T2, T3, T4, T5} |

C2:

|  |  |
| --- | --- |
| Item | TID-set |
| a, b | {T3, T5} |
| a, c | {T1, T3, T5} |
| a, e | {T3, T5} |
| b, c | {T2, T3, T5} |
| b, e | {T2, T3, T4, t5} |
| c, e | {T2, T3, T5} |

The eclat code:

'''

The Eclat algorithm

input: path - a path of a txt file, minsup - an integer number

output: res - list of frequent itemsets

'''

**def** **eclat**(path, minsup):

C = readFile(path)

L = frequentItemSets(C, minsup) # 1-size frequent itemsets

res = [] # the result list

**while** (L):

# add frequent itemsets (without their TIDs list) to res

**for** i **in** range(len(L)):

res.append(L[i][**0**])

C = candidateItemSets(L) # candidate k+1-size itemsets

L = frequentItemSets(C, minsup) # remove infrequent itemsets

**return** res

The other functions:

'''

function creates a list of items and their corresponding TIDs that they appear

input: path - path of txt file representing transaction database

output: C - a 3D list contains list of 1-size itemsets and list of the corresponding TIDs

'''

**def** **readFile**(path):

**with** open(path, 'r') **as** f:

listOfTIDs = [line.strip().split(' ') **for** line **in** f]

listOfItems = sorted({item **for** tid **in** listOfTIDs **for** item **in** tid})

C = [[[], []] **for** i **in** range(len(listOfItems))]

**for** x **in** range(len(listOfItems)):

C[x][**0**] = list(listOfItems[x])

C[x][**1**] = [y+**1** **for** y **in** range(len(listOfTIDs))

**if** set(listOfItems[x]).issubset(set(listOfTIDs[y]))]

**return** C

'''

function removes less-support-frequent itemsets from C

input: C - a 3D list of itemsets, minsup - an integer number

output: L - 3D list of frequent itemsets

'''

**def** **frequentItemSets**(C, minsup):

L = []

**for** row **in** range(len(C)):

**if** len(C[row][**1**]) >= minsup:

L.append(C[row])

**return** L

'''

function candidates increasing-in-1-size itemsets from L

input: L - 3D list of frequent itemsets

output: C - 3D list of itemsets

'''

**def** **candidateItemSets**(L):

C = []

**for** i **in** range(len(L)):

**for** j **in** range(i+**1**, len(L)):

row = [[], []]

**if** L[i][**0**][:-**1**] == L[j][**0**][:-**1**]:

row[**0**] = sorted(set(L[i][**0**]).union(set(L[j][**0**])))

row[**1**] = sorted(set(L[i][**1**]).intersection(set(L[j][**1**])))

C.append(row)

**return** C

## Hmine

This method is proposed for mining frequent patterns for the data sets that can fit in (main) memory, scalable algorithm for fast mining.

H-mine is integrated with FP-growth dynamically by detecting the swapping condition and constructing FP-trees for efficient mining.

First, following Apriori property. By scanning TID once, create the complete set of frequent items.

With minsup = 2.

|  |  |  |
| --- | --- | --- |
| TID | Items | Frequent-item projection |
| T100 | 1, 3, 4 | 3, 4 |
| T200 | 2, 3, 5 | 2, 3 |
| T300 | 2, 4 | 2, 4 |

Following the numeric order of frequent items (called F-List): 2, 3, 4 the complete set of frequent patterns can be partitioned into 3 subsets as follows: (1) those containing item 2; (2) those containing item 3 but no item 2; (3) those containing item 4 but no item 2 nor 3;  If the frequent-item projections of transactions in the database can be held in main memory. All items in frequent-item projections are sorted according to the F-list like T100 with 3, 4; Then, we can create “Header table H” with 3 fields: an item-id, element count, and hyper-link.  When the frequent-item projections are loaded into memory, those with the same first item (in the order of F-list) are linked together by the hyper-links as a queue, and the entries in the header table act as the heads of the queue. Whenever going through the new TID based on sorted by F list property, use hyper-links to check if the frequent items are available or not.

Header table H

|  |  |  |  |
| --- | --- | --- | --- |
| Item-id | 2 | 3 | 4 |
| Support count | 2 | 2 | 2 |
| Hper-links |  |  |  |

From the F-list go through TID we got 3 => 4.

So, value 3 will contain a hyper-link to lead into TID 1, to get frequent item set {3,4}. The same way with all the item set, for the longer item set it will connect with another variable behind it.

Here is the Hmine function:

**def** **hmine**(prefix=None, prefixlen=**0**, rowlist=None):

**if** prefix **is** None:

prefix = []

**if** rowlist **is** None:

rowlist = []

**for** row **in** rowlist: # Traversing the header table (rowlist)

newRowlist = []

mapItemRow.clear()

# traversing all pointers of row object in row list and building new recursive sub-level header

**for** pointer **in** row.pointer:

pointer += **1**

**if** cell[pointer] == -**1**:

**continue**

# Generating the row objects and incresing the support for all the unique items in row objects

**while** cell[pointer] != -**1**:

item = cell[pointer]

**if** mapItemRow.get(item, None) == None:

rowItem = Row(item)

rowItem.support = **1**

rowItem.pointer.append(pointer)

mapItemRow[item] = rowItem

**else**:

mapItemRow[item].support += **1**

mapItemRow[item].pointer.append(pointer)

pointer += **1**

# Appending only those row objects which have support greater than min\_support

**for** entry **in** mapItemRow:

currentRow = mapItemRow[entry]

**if** currentRow.support >= minSupport:

newRowlist.append(currentRow)

# Calling writeOut function to generate the frequent items and store in output list

writeOut(itemsetBuffer, prefixlen, row.item, row.support)

# Sorting newRowlist in lexical order

**if** len(newRowlist) != **0**:

newRowlist = sorted(newRowlist, key=**lambda** x: x.support)

# Store current row item in buffer before recursion so that it can be used to build the frequent itemset values

itemsetBuffer[prefixlen] = row.item

hmine(prefix, prefixlen + **1**, newRowlist) # recursively calling Hmine algorithm

hmine(itemsetBuffer, **0**,rowlist) # Calling Hmine algorithm for first time using empty Buffer and 0 as prefixlength and initial value of rowlist Header.

itemset\_count = len(out\_frequents\_items) # Total number of frequent\_items for given input\_file dataset

## FP-Growth

The FP-growth function performs the data by using FP-Tree. By using a recursive function to connect the frequent items in every TID with each other.

Ex:

|  |  |
| --- | --- |
| TID | Items |
| 1 | f, a, c, d, g, i, m, p |
| 2 | a, b, c, f, l, m, o |
| 3 | b, f, h, j, o |
| 4 | b, c, k, s, p |
| 5 | a, f, c, e, l, p, m, n |

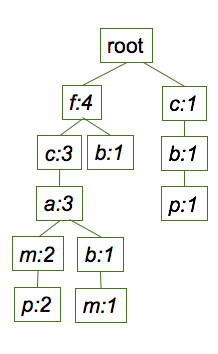
With minsup = 3, we a list of frequent item:

|  |  |
| --- | --- |
| Items | Sup |
| f | 4 |
| c | 4 |
| a | 3 |
| b | 3 |
| M | 3 |
| p | 3 |

Next, order each transaction's newly discovered Level 1 popularity items by decreasing popularity:

|  |  |  |
| --- | --- | --- |
| TID | Items | Public items |
| 1 | f, a, c, d, g, i, m, p | f, c, a, m, p |
| 2 | a, b, c, f, l, m, o | f, c, a, b, m |
| 3 | b, f, h, j, o | f, b |
| 4 | b, c, k, s, p | c, b, p |
| 5 | a, f, c, e, l, p, m, n | f, c, a, m, p |

Next, move through the level 1 popular items in order of increasing support, p,m,b, a,c,f. Build the conditional pattern-base and conditional FP-Trees for each item. Create FB-Tree:



Beginning with item p, its conditional pattern base is all of the FP-prefix Tree's paths, specifically fcam:2 and cb:1, while traveling from root = null to node p. (the number that follows) are how many times each of those prefixes appear in relation to node p.

After merging all the pathways and maintaining the nodes with the sum of the counts sup = 3, we create a conditional FP-Tree from this pattern:

Only c:3 satisfies the requirement after the fcam:2 and cb:1 remix to f:2, c:3, a:2, m:2, and b:1.

Therefore, p and cp are the most typical patterns that contain. Do the same with the rest of the item set. We got a table.