HOMEWORK 02: Frequent Graph Pattern Mining

StudenID: 20127054

Name: Ngô Văn Trung Nguyên

Problem 1. Apriori-based Graph Mining (AGM)

- ***Describe Apriori-based Graph Mining (AGM) algorithms in a way you understand.***
  - Apriori-based Graph Mining (AGM) algorithms are a family of graph mining algorithms that aim to find frequent subgraphs in a given dataset of graphs. They are based on the Apriori principle, which states that if a set of items is frequent, then all of its subsets must also be frequent.
  - Firstly, the algorithm identifies all the frequent single-item subgraphs. Then, for each pair of items in the frequent single-item subgraphs, the algorithm generates a web address 2d array that counts the number of graphs containing the pair.
  - Next, the algorithm proceeds by generating all possible combinations of items derived from the frequent single-item subgraphs. These combinations are then used to create a list of subgraphs that contain those items. For each subgraph in the list, the algorithm uses the AprioriGraph function to recursively explore its adjacent vertices. If an adjacent vertex satisfies the condition of being higher than or equal to mincof, it is included in the subgraph. Additionally, the AprioriGraph function verifies if the subgraph meets the requirement of being higher than or equal to minsup, and if so, adds it to a separate list of frequent subgraphs.
  - Finally, the algorithm generates association rules from the frequent subgraphs. For each subgraph, the function generates all possible subsets of items and calculates the support of those subsets. If the support is higher than minsup, the function calculates the confidence of the rule. The rules that meet the minsup and minconf are added to a list called Rules.
- ***Explain the difference between the support of a frequent subgraph and the confidence of a rule in the context of the AGM algorithm.***
  - The support of a frequent subgraph measures how frequently a subgraph appears in the dataset. A subgraph is considered frequent if its support is greater than or equal to minsup. The higher the support value, the more frequently the subgraph occurs in the dataset, indicating its significance in the overall dataset.
  - The confidence of a rule measures the strength of the rule in predicting the occurrence of a certain subgraph. A rule consists of an antecedent subgraph and a consequent subgraph. The confidence of a rule is calculated as the proportion of transactions (or graphs) containing the antecedent subgraph that also contain the consequent subgraph. It indicates the likelihood that the occurrence of the antecedent will result in the occurrence of the consequent.
- ***Consider the following market transactional database.***

Transaction ID	List of item's in each transaction
1	Pasta, Fruits, Butter, Vegetables
2	Burgers, French Fries, Ketchup, Mayo
3	Burgers, French Fries, Ketchup
4	Burgers, Pasta, French Fries
5	Vegetables, Fruits
6	Fruits, Orange Juice, Vegetables
7	Burgers, French Fries, Vegetables

To find frequently occurring patterns in a given database, use the Apriori algorithm with a support threshold of 25%. After obtaining these frequent patterns, list all the association rules that are considered strong, with a confidence threshold of 50%. Prior to implementing the Apriori algorithm, convert the given database into a graph. It is important to note that only providing the final result will not be sufficient, so all the individual steps involved should be presented and explained in detail.

- support threshold is 25% and number of transactions are 7, so support no is 0.25*7=1.75 or 2
- Find all the items occur more than 1.75 or precisely atleast 2 times from the sample database. The items which qualify are Pasta (count: 2), Fruits (count: 3), Vegetables (count: 4), Burgers (count: 4), French Fries (count: 4), Ketchup (count: 2)
- Initializes a 2D matrix webaddr. Assume for a 2D matrix, vertical coordinates of the graph are named as the X coordinate and horizontal coordinates are called Y coordinate. The X and Y coordinates of the matrix are names of the items which qualified after previous step. i.e. Pasta, Fruits, Vegetables, Burgers, French Fries, Ketchup.

	Pasta	Fruits	Vegetables	Burgers	French Fries	Ketchup
Pasta	0	0	0	0	0	0
Fruits	0	0	0	0	0	0
Vegetables	0	0	0	0	0	0
Burgers	0	0	0	0	0	0
French Fries	0	0	0	0	0	0
Ketchup	0	0	0	0	0	0

- A loop is run for every transaction in the sample database from tiD = 1 to tiD = 7. Generates combinations of two items for every transaction. Checks whether the combinations generated are present in matrix webaddr. If the combinations are present then increase the value by 1.
- After the execution:

	Pasta	Fruits	Vegetables	Burgers	French Fries	Ketchup
Pasta	0	1	1	1	1	0
Fruits	0	0	3	0	0	0
Vegetables	0	0	0	1	1	0

Burgers	0	0	0	0	4	2
French Fries	0	0	0	0	0	2
Ketchup	0	0	0	0	0	0

- Global 2D list variable “Answer” is declared so that all discovered frequent patterns in the form of lists are stored in this variable. Another list variable “result_list” is declared.
- Finds the frequent patterns from the webaddr above, a loop is run for each vertex found in Pasta, Fruits, Vegetables, Burgers, French Fries, Ketchup. Every time “for” loop iterates, variable “result_list” is emptied.
-
- “result_list” is declared empty

Null	Null	Null	Null

- We add Pasta. We cannot find (Pasta -> something) meets minsup. Continue
- We add Fruits in (we skip Pasta because Pasta row does not have anything >= 2):

Fruits	Null	Null	Null

- Then Vegetables (Fruits -> Vegetables is 3)

Fruits	Vegetables	Null	Null

- We cannot find (Vegetables -> something) meets minsup. Append [Fruits, Vegetables] to Answers.
- We add Vegetables. We cannot find (Pasta -> something) meet minsup. Continue

Vegetables	Null	Null	Null

- We add Burgers

Burgers	Null	Null	Null

- We add French Fries

Burgers	French Fries	Null	Null

- (French Fries -> Ketchup) meets minsup. We add Ketchup

Burgers	French Fries	Ketchup	Null

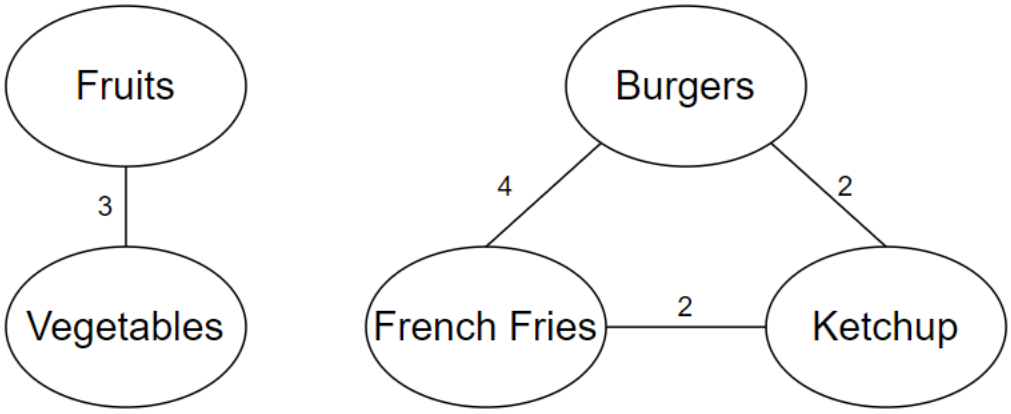
- Check [Burgers, French Fries, Ketchup] support = 2. Append to Answers.
- Delete French Fries. Add Ketchup

Burgers	Ketchup	Null	Null

- Append [Burgers, Ketchup]
- Add French Fries and Ketchup

French Fries	Ketchup	Null	Null

- Append [French Fries, Ketchup]
- **We have the graph:**



- **We have these frequent patterns:**
  - [Fruits, Vegetables]
  - [Burgers, French Fries, Ketchup]
  - [Burgers, French Fries]
  - [Burgers, Ketchup]
  - [French Fries, Ketchup]

- Conf(Fruits → Vegetables) = $\frac{support(Fruits,\text{Vegetables})}{support(Fruits)} = \frac{3}{4} = 0.75$
- Conf(Vegetables → Fruits) = $\frac{support(Fruits,\text{Vegetables})}{support(\text{Vegetables})} = \frac{3}{3} = 1$
- Conf(Burgers, French Fries → Ketchup) = $\frac{support(\text{Burgers ,French Fries,Ketchup})}{support(\text{Burgers,French Fries})} = \frac{2}{4} = 0.5$
- Conf(Burgers, Ketchup → French Fries) = $\frac{support(\text{Burgers ,French Fries,Ketchup})}{support(\text{Burgers,Ketchup})} = \frac{2}{2} = 1$
- Conf(French Fries, Ketchup → Burgers) = $\frac{support(\text{Burgers ,French Fries,Ketchup})}{support(\text{French Fries,Ketchup})} = \frac{2}{2} = 1$
- Conf(Burgers → French Fries, Ketchup) = $\frac{support(\text{Burgers ,French Fries,Ketchup})}{support(\text{Burgers})} = \frac{2}{4} = 0.5$
- Conf(French Fries → Burgers, Ketchup) = $\frac{support(\text{Burgers ,French Fries,Ketchup})}{support(\text{French Fries})} = \frac{2}{4} = 0.5$
- Conf(Ketchup → Burgers, French Fries) = $\frac{support(\text{Burgers ,French Fries,Ketchup})}{support(\text{Ketchup})} = \frac{2}{2} = 1$
- **After calculating, we have these association rules:**
  - Fruits →Vegetables
  - Vegetables → Fruits
  - Burgers, French Fries → Ketchup

+ Burgers, Ketchup → French Fries
+ French Fries, Ketchup → Burgers
+ Burgers → French Fries, Ketchup
+ French Fries → Burgers, Ketchup
+ Ketchup → Burgers, French Fries

Problem 2. Graph-based Substructure Pattern Mining (gSpan)

- ***Describe gSpan algorithms in a way you understand.***
  - The gSpan algorithm creates common subgraphs from the bottom up, using a unique DFS code as the canonical representation of the subgraph. Duplicate subgraphs are eliminated by selecting the minimal DFS code in the dictionary order. These techniques help gSpan efficiently extract frequent subgraphs from large graph datasets.
- ***What is the comparison made by gSpan algorithm while testing for isomorphism between two graphs and what is the reason for this comparison ?***
  - The gSpan algorithm performs a comparison between two graphs to test for isomorphism. Specifically, it compares the canonical code representations of the two graphs. The canonical code is a unique and minimal representation of a graph that preserves its structure and connectivity. It is generated by traversing the graph using DFS, following a specific set of rules to determine the order in which nodes and edges are encoded.
  - The reason for comparing the canonical codes is to check if the two graphs have the same structure, disregarding any labeling or ordering differences. By comparing the canonical codes, gSpan algorithm can identify whether the graphs are isomorphic (i.e., structurally identical) or not. This comparison is crucial because it allows gSpan algorithm to efficiently discover frequent subgraphs in a given set of graphs. By identifying isomorphic subgraphs, gSpan avoids redundant exploration of similar patterns and improves the efficiency of mining frequent subgraphs.
- ***Which two main expenditures of Apriori were targeted by gSpan for reduction ?***
  1. *Candidate generation*: Apriori generates a large number of candidate itemsets, many of them are not frequent and must be pruned. This process can be computationally expensive and time-consuming.
  2. *Frequent itemset counting*: Apriori needs to scan the entire dataset multiple times to count the support of each candidate itemset. This can also be a time-consuming process, especially for large datasets.
- ***Draw a DFS tree corresponding to the following DFS code.***

Edge	Code
0	(0, 1, Y, a, X)
1	(1, 2, X, a, X)
2	(2, 0, X, b, Y)
3	(2, 3, X, c, Z)

4	(3, 0, Z, b, Y)
5	(0, 4, Y, d, Z)