# COL761 : Assignment 1 - Writeup

Arushi Goyal (2020CS50418)
Ananya Mathur (2020CS50416)
Ishaan Govil (2020CS50497)

August 20, 2023

## 1 Algorithm

### 1.1 Frequent Itemset Generation

#### 1.1.1 FP Tree

The Frequent Pattern Tree (FP-Tree) is a compact data structure that stores quantitative information about frequent patterns in a database. Each transaction is read and then mapped onto a path in the FP-Tree. This is done until all transactions have been read. A Frequent Pattern Tree is made with the initial item sets of the database.

#### 1.1.2 FP Growth Algorithm

First, we compress the input database creating an FP-tree instance to represent frequent items. After this first step, the compressed database is divided into a set of conditional databases, each associated with one frequent pattern. We modified the standard FP-Tree implementation by stopping the pattern generation step once pattern length exceeds a fixed value. Since patterns are being formed incrementally this significantly reduces the computation time as very long patterns (greater than length 4) are not very useful in compression and don't provide a significant improvement in the compression ratio when weighted against the time it takes to check all the 4-sized itemsets. Finally, patterns with support greater than the supplied value and $\leq$ *threshold length* are outputted.

### 1.2 Compression Algorithm

- We first begin by running the *FP_tree* algorithm with required support to get the frequent itemsets stored as a vector of vectors named *patterns*.
- We then make a map for the frequent itemsets and assigning a key to each of these itemsets to represent them finally while compression. We also store the no of frequent itemsets in which each item lies while iterating over all the frequent itemsets in this step. We simply calculate the max item(max integer) present in the data, and start assigning keys from the max item onwards, to avoid any clashes between elements and keys.

- Next, We compress the transactions one at a time and independent from each other.
- For, each transaction we sort the elements of that transaction based on the no of patterns it appears in, in increasing order. This is done to first assign the items which appear in lesser no of patterns as compared to the other items.
- Next, the elements with smallest frequency in the frequent itemsets are considered first to greedily find out the longest set of elements which are a frequent pattern in the transaction. Once such a patterns is found at each step, the *current_key* variable is replaced with the key assigned to that frequent pattern, before finally pushing it into the *compressed_transactions* vector.
- The elements already used for compression into a frequent pattern are marked and remaining ones are considered in each iteration for compressing a frequent pattern in the transaction.
- We optimised this greedy compression algorithm by not checking/finding compression pattern existence for elements in the transaction which do not appear in any frequent pattern. This reduced time considerably for sparse datasets.
- We also store the maximum size of the frequent patterns that we have found and so once the size of the best pattern we have found for a particular element reaches maximum pattern size we break as other elements need not be checked as we will not be able to find a frequent pattern with the already considered items. This again reduced the running time of our code.
- We also store the patterns that are actually used by our compression algorithm to avoid printing all the patterns in the output file in a *patterns_used* set which is used while writing into the output file.

## 2 Decompression

For *decompression*, we first read the key-itemset pairs from the output file(which is present on the initial lines of the output file and create a map to store the patterns. Then we start reading the transactions one at a time, and for element we check whether their is any itemset which correspond to that element. Incase, it does we replace the key, with it's contents(the items in the itemsets) stored in the map, otherwise we keep the element unchanged. Once all the transactions have been processed, we finally write then in the decompressed file.

## 3 Acknowledgement

For FP-Tree implementation we referred to the following:

`https://github.com/Vedant2311/Data-Mining-Algorithms/tree/main/Frequent-Itemset-Mining`