COL761

MININGMAVERICKS

Assignment 1

Chinmay Mittal *(2020CS10336)*
Parth Shah *(2020CS10380)*
Shubh Goel *(2020EE10672)*

Date of submission of Report: August 20, 2023

1 FP Tree Implementation

We use FP Trees for mining the frequent set of transactions in the database.

1.1 FPNode

We use this class to store the information of a node in the FP Tree

```
class FpNode
{
    public:
        const Item item;
        uint64_t frequency;
        std::shared_ptr<FpNode> next_node_in_ht;
        std::weak_ptr<FpNode> parent;
        std::map<Item, std::shared_ptr<FpNode>> children;

        FpNode(const Item&, const std::shared_ptr<FpNode>&);
};
```

1.2 FPTree

FPTree class is used for construction of the FPTree for the given database and for finding the conditional FPTree corresponding to a given set of items.

```
class FpTree
{
    public:
        std::shared_ptr<FpNode> root;
        std::map<Item, std::shared_ptr<FpNode>> header_table;
        std::map<Item, std::shared_ptr<FpNode>> last_node_in_header_table;
        std::map<Item, uint64_t> item_frequencies;
        uint64_t minimum_support_threshold;
        uint64_t total_transactions;
        uint64_t total_items;

        FpTree(const std::string&, float);
        FpTree(const std::vector<Transaction>&, float);
        FpTree(const std::vector<TransformedPrefixPath>& transactions,
    uint64_t);

        bool empty() const;
};
```

There are 3 different types of constructors used:

1. Construct the FPTree using the filepath and the minimum support threshold
2. Construct the FPTree using the list of transactions and the minimum support threshold
3. Construct the FPTree using a set of transformed prefix paths and minimum support threshold: A transformed prefix path is a path in the FP Tree corresponding to an item set and consisting of the frequency of that item set

1.3 mine_fptree(FPTree)

This function is used to mine the frequent patterns in the FPTree and save them in a list consisting of the patterns and their corresponding frequencies.

This function generates the conditional FPTrees for an item by creating a list of paths in the tree consisting that item with their frequencies and constructing a new FPTree using the third constructor in the FPTree class.

Then it recursively calls the mine_fptree function on all the conditional FPTrees till either there is only one path left or the tree is empty.

If the time after the first call to this function exceeds a certain value we return all the frequent patterns mined till that point and stop execution.

After trying out different examples we find 450 sec as the time to stop mining.

2 Compression Algorithm

Algorithm 1 Compression

```
1: Input: set_of_transactions
2: support_thresholds ← list of supports in decreasing order
3: compression_dictionary ← {}
4: key ← −1
5: current_transactions ← set_of_transactions
6: for support ∈ support_thresholds do
7:     fptree ← construct_FPtree(current_transactions, support)
8:     frequent_patterns ← mine_FPtree(fptree)
9:     compressed_transactions ← {}
10:    for transaction ∈ current_transactions do
11:        for pattern ∈ frequent_patterns do
12:            if pattern.size() ≥ 2 & pattern ∈ transaction then
13:                if pattern ∉ compression_dictionary.keys then
14:                    compression_dictionary(pattern)] ← key
15:                    decrement key
16:                end if
17:                replace pattern with compression_dictionary(pattern) in transaction
18:            end if
19:        end for
20:        compressed_transactions.push(transaction)
21:    end for
22:    current_transactions ← compressed_transactions
23: end for
```

The level of support chosen in the compression process has a direct impact on both the number of frequent item sets identified and the time required for mining. When a high support threshold is employed, fewer frequent item sets are discovered, resulting in a quicker mining process, but the achieved compression is relatively low. Conversely, opting for a low support threshold leads to higher compression and longer mining times. Thus, there exists a trade-off between the degree of compression and the mining duration. To address this trade-off, the algorithm employs a **dynamic approach**. Instead of utilizing a fixed support value, it conducts compression cycles using a list of support values in descending order.

Within each cycle, the algorithm identifies frequent item sets and replaces them with their compressed representations in the transaction data. These compressed patterns are stored in a map alongside their corresponding keys, which will later facilitate decompression. Subsequent compression cycles operate on this partially compressed dataset. Importantly, any subsets of

frequent items identified in previous cycles are not re-mined in the current cycle since they have already been replaced with compressed representations. This optimization significantly reduces computational overhead in subsequent mining cycles, making the process more efficient.

Heuristics applied for optimization

1. Each compression cycle of the algorithm is $O(N \times M)$, assuming constant time for pattern matching and FPtree mining. Here, N represents the size of the transaction database, and M represents the size of the frequent item sets mined during that specific cycle.

   In instances where, during a cycle, the number of frequent patterns mined falls below a certain predefined threshold value, the compression process for that cycle is omitted. Furthermore, the subsequent compression cycle is entirely skipped under the assumption that a marginal reduction in support values will not result in a substantial increase in the number of frequent item sets mined. This strategy helps optimize the algorithm's performance by avoiding unnecessary compression cycles when the frequent item sets mined are very less.

```
Input: set_of_transactions
support_thresholds ← list of supports in decreasing order
compression_dictionary ← {}
key ← −1
current_transactions ← set_of_transactions
iter ← 1
while iter ≤ support_thresholds.size() do
    fptree ← construct_FPtree(current_transactions, support(iter))
    frequent_patterns ← mine_FPtree(fptree)
    if frequent_patterns ≤ α then
        iter ← iter + 2
    else
        perform compression
        iter ← iter + 1
    end if
end while
```

2. Due to the reason explained in the first point, before proceeding to compression process, we ensure that the number of frequent items mined are always less than a predefined upper-bound,W. We designed the following algorithm to achieve this. We repeatedly mine the transactions with increasing support values until the size of frequent item sets falls below the upper bound.

```
curr_threshold ← support[i]
frequent_patterns ← mine_tree(FPTree, curr_threshold)
in_while ← 0
while frequent_patterns.size() ≥W do
    curr_threshold ← (past_threshold + curr_threshold)/2
    frequent_patterns ← mine_tree(FPTree, curr_threshold)
    in_while ← 1
end while
past_threshold ← curr_threshold
i ← i − in_while
```

3. let **Z** for an item set $\mathcal{I}$ be defined as:

$$\mathbf{Z} = (\mathcal{I}.size - 1) \times (\mathcal{I}.frequency - 1)$$

Before performing the compression(line 10 in Algorithm 1), we sort the mined frequent item sets in the decreasing order of **Z**.

Rationale :The amount of compression achieved by an Item, $\mathcal{I}$ will be $(\mathcal{I}.size \times \mathcal{I}.frequency) - (\mathcal{I}.frequency) - (\mathcal{I}.size + 1)$, where the first term represents the number of characters removed, second term represents the number of keys replaced with the item set, and the third term represents the number of characters added in the compression dictionary. On rearranging the terms, we get $\mathbf{Z} - 2$. Hence, if two frequent items sets, $\mathcal{I}_1$ and $\mathcal{I}_2$ are present in a transaction with $\mathcal{I}_1 \cap \mathcal{I}_2 \neq \phi$ and $\mathbf{Z}_{\mathcal{I}_1} > \mathbf{Z}_{\mathcal{I}_2}$, it is better to replace $\mathcal{I}_1$.

Here the value of W is can be hard-coded as well as calculated using the size of the dataset.

4. For certain support thresholds it can so happen that we generate a lot of frequent patterns and then processing all these patterns for each transaction can become quite expensive. We have done some experiments to prune the frequent patterns. We pick only the 10000 best patterns according to the heurestic defined above.

5. We also add time limits to the iterative algorithm and the mining process and ensure that our algorithm terminates in a certain time (40 minutes in entirety and 450 seconds per mining process)

Example run

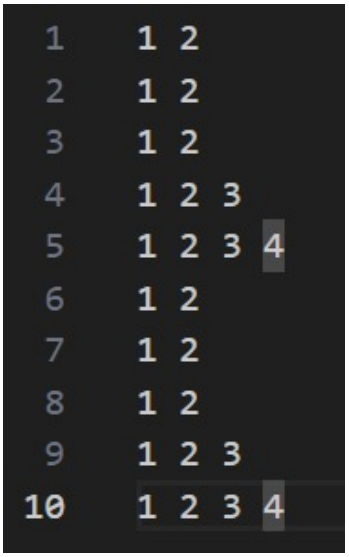
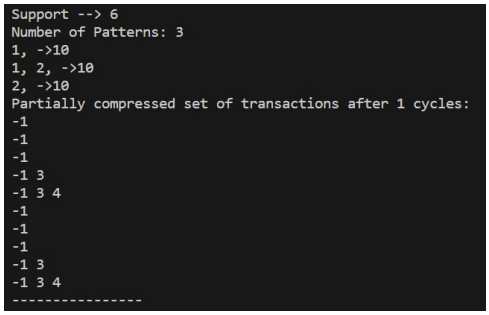


Figure 1: Input set of transactions

List of support thresholds = $\{6, 4, 2\}$



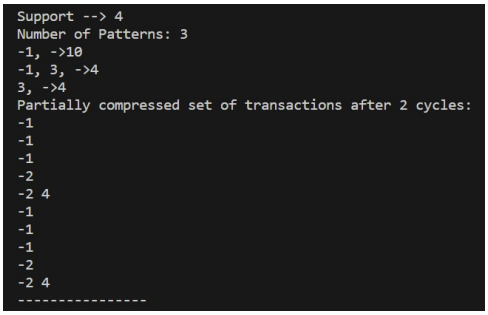


Figure 2: Algorithm run

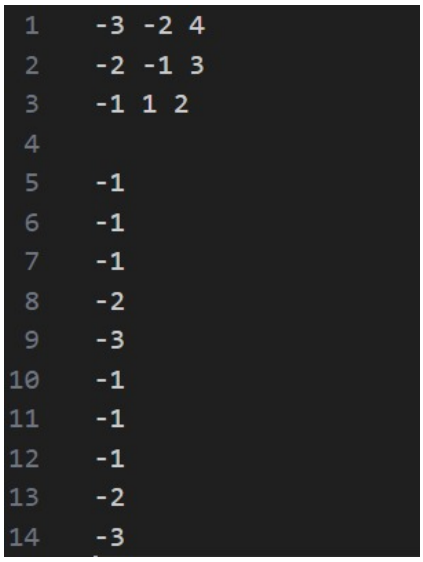


Figure 3: Output (Initial Items = 26, Final Items = 19)

3 Results on given datasets

```
Initial Items: 118252
Final Items: 14027
Amount of Compression: 88.138
Time taken: 7.294 seconds
```

(a) D_small.dat

```
Initial Items: 8019015
Final Items: 6220765
Amount of Compression: 22.4248
Time taken: 415.786 seconds
```

(b) D_medium.dat

```
Initial Items: 3960507
Final Items: 1734601
Amount of Compression: 56.2026
Time taken: 484.861 seconds
```

(c) D_medium2.dat

```
Initial Items: 109360594
Final Items: 98892108
Amount of Compression: 9.57244
Time taken: 1930.85 seconds
```

(d) large.dat

Figure 4: Test datasets results