# COL761 Assignment-1

## Compression using Frequent Itemset Mining

**Team:** COL761Eagles

### Names and contribution:

1) Vanshita Garg (2020EE30144) - 50%
2) Mohit Garg (2020AM10657) - 50%
3) Darpit Rangari (2020BB10044) - 0%

### Our Approach and Algorithms used

1) **FPTree Implementation** (fptree.cpp and fptree.h)

   We made use of FPTree to mine frequent patterns from the dataset and then use those for creating a compression dictionary to compress the database.

   We directly made use of the existing implementation in [1] and [3]. We adapted the idea of using map for FPNode's children from [3] which fastened the mining process.

   We also modified the implementation by using the idea of [2]- **SRMine**. This is basically used to speed up the construction of the FPtree. So instead of processing and inserting transactions one by one in the tree, we create a hashmap that stores the frequency of occurrence of every unique transaction. And then starts adding these unique transaction keys one by one, to the tree. This reduces the redundant time of adding multiple transactions again and again.

   At the same time, we also put a limit on the number of frequent patterns mined so that at the time of compression, we are not stuck with the problem of exploding number of frequent patterns as the support decreases.

**References :-**

We used the following resources for the tree implementation, as stated in class (that we are not required to code the tree from scratch and can use existing implementations) :-

[1] https://github.com/integeruser/FP-growth/tree/master
[2] https://link.springer.com/content/pdf/10.1007/s13369-021-06298-9.pdf
[3] https://github.com/ChinmayMittal/COL761-Data-Mining/blob/main/A1/fptree.cpp
[4] ChatGPT and stack overflow for basic file stream commands

## 2) **Compression Algorithm** (main.cpp)

For compression, the main problem was to set the support threshold as a tradeoff between the number of frequent patterns mined and the compression ratio.
For this we first tried to store some predefined thresholds in a vector and loop over that. However this method was giving sub-optimal results with large time.

The following heuristics were thereby adopted:

1) While reading the transactions from the given file, we divided them into buckets based on transaction size. So transactions with sizes less than 100 were placed in one bucket, with sizes between 100 and 1000 were placed in the other bucket and so on. Simultaneously, we also took care that none of the buckets had too few or too large transactions, except for the last bucket for which since size of each transaction in it will be too large, too less transactions in it would not cost us anything. The logic behind making these buckets was two-folded. Firstly, we will be processing only a limited number of transactions while looping over each bucket. This would speed up the FPtree process and its mining. Secondly, we worked under the assumption that transactions with similar sizes have higher chances of sharing patterns, than two transactions of which one is of length 2 and other of length 60 and so on.

2) Now for setting the support thresholds, we created a while loop wherein an fptree is generated for a threshold, mined and the obtained frequent patterns obtained are used to encode the transactions then only . That is a sort of dynamic approach is deployed.

After conducting several runs we saw that if we did not use higher thresholds to encode the frequent patterns in transactions in the beginning, then later when we use the smaller thresholds, then mined frequent patterns become so much in number, that it takes a lot of time. Therefore, while processing every bucket of transactions, we start using the support thresholds in descending order and keep on decreasing them step wise. We aimed at reducing threshold by bigger step sizes initially and then keep on decreasing the gap in consecutive threshold by reducing the step sizes.

3) By the above logic in (2), we started with a threshold of 1.0, and then until it reached 0.2, we decreased it in an alternating manner using step sizes of 0.1 and 0.05. Later we decrease the step sizes to 0.0125 and 0.0005.This ensures we capture the rare frequent itemsets as well. These values are based purely on experimentation.

4) At the same time, we added the time component to the algorithm. For every bucket we specify a time limit under which its compression has to complete, otherwise we move to the next bucket. This is basically done to be able to run the algorithm in less than hour . And for every successive bucket we keep on increasing this limit by 60 seconds, as successive buckets in our algorithm, have more number of tokens and hence require greater time. So this takes the overall algorithm limit to ( 5+6+7+8+9) min approximately for the largest of datasets. At  the same time in this implicitly we defined a time limit of 2 min for FPtree mining.

5) Finally we compress transactions in the ‘helper’ function in the main.cpp file. The main heuristic we used in this method is we arranged the patterns in decreasing order of size. This is necessary as this allows us to handle bigger patterns first which reduces the number of total tokens and hence reduces the time. Also we used only those patterns which encoded more than one item and had frequency of occurrence greater than 1.

6) Final detail is that we start with a label of -1 to encode our frequent itemsets as the items are all non-negative in the original database. We use long long integer type for this and keep on decrementing the label with new discovered itemsets.

Summarizing the algorithm, outer loop is for each bucket. Then inside that we make a while loop to mine patterns till threshold is greater than 0 and time elapsed is under a specified limit. For every threshold inside this inner loop, patterns are mined and simultaneously replace the bucket transactions using helper function.

## 3) **Decompression Algorithm** (decompress.cpp)

For decompression, we first read the input compressed file. The file is so saved that the first part of it saved the compression dictionary and the next saved the compressed database. We create a custom comparator function which keeps the keys in the dictionary in descending order. This is because in our compression algorithm we are giving the labels starting from -1 to our frequent patterns. And decreasing labels iteratively as patterns increase. So technically a label of -10 will be encoding transaction items with values >-10, that is between -1 to -9. It encode a future value lesser than it. Hence while reading the compressed file, we use this custom key comparator to store keys in descending order.
Using the helper function 'get_decompressed_dict', we create a new dictionary for which all negative keys correspond to itemsets with positive values using the above logic. This obtained dictionary is then used to decompress the file .

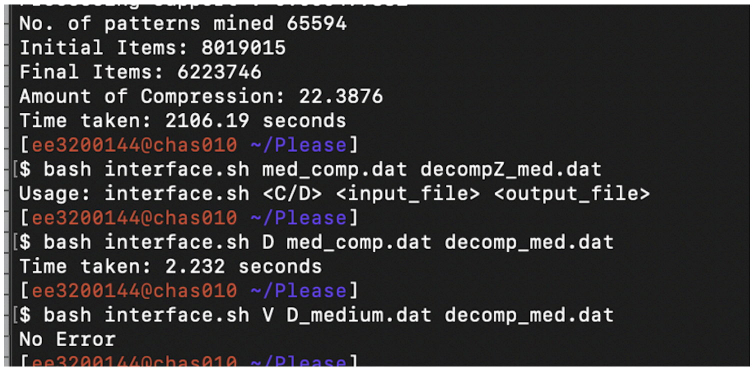
## 4) **Verification** (verify.cpp)

A verify.cpp file is created which checks if there is any error in the decompressed file and the original file. The transactions are read from the two file locations as a set of transactions, and every transaction as a set of items. Simple for loops are written to verify the same.

## 5) **Results and Observations**

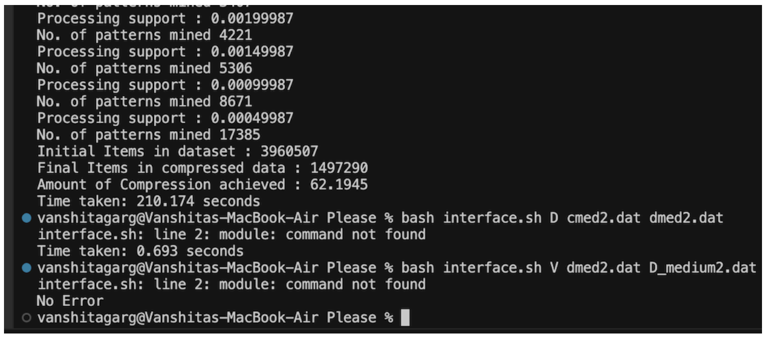
1) D_small.dat , D_medium.dat, D_medium2.dat

| | Compression % | Compression Time |
|---|---|---|
| D_small.dat | 88.13 % | 0.978 seconds |
| D_medium2.dat | 47.5307 % | 475.68 seconds |
| D_medium.dat | 22.3876 % | 2106.19 seconds |



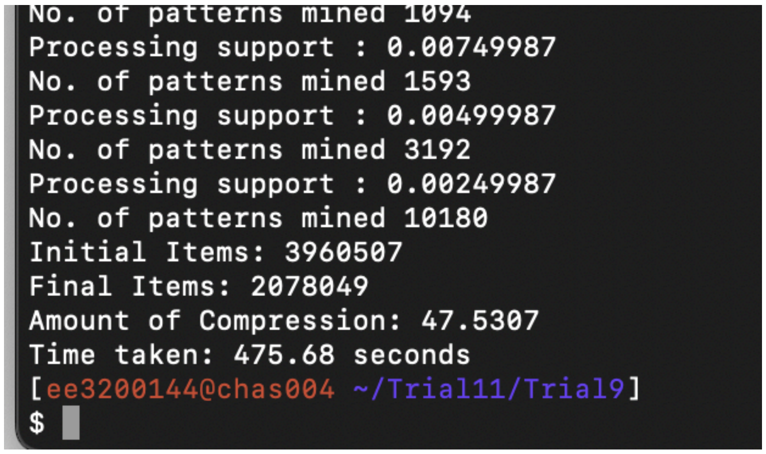
```
No. of patterns mined 65594
Initial Items: 8019015
Final Items: 6223746
Amount of Compression: 22.3876
Time taken: 2106.19 seconds
[ee3200144@chas010 ~/Please]
$ bash interface.sh med_comp.dat decompZ_med.dat
Usage: interface.sh <C/D> <input_file> <output_file>
[ee3200144@chas010 ~/Please]
$ bash interface.sh D med_comp.dat decomp_med.dat
Time taken: 2.232 seconds
[ee3200144@chas010 ~/Please]
$ bash interface.sh V D_medium.dat decomp_med.dat
No Error
```

D_medium.dat



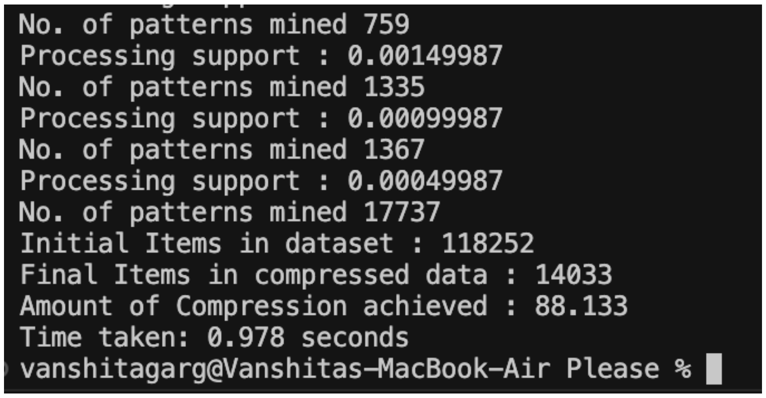
```
Processing support : 0.00199987
No. of patterns mined 4221
Processing support : 0.00149987
No. of patterns mined 5306
Processing support : 0.00099987
No. of patterns mined 8671
Processing support : 0.00049987
No. of patterns mined 17385
Initial Items in dataset : 3960507
Final Items in compressed data : 1497290
Amount of Compression achieved : 62.1945
Time taken: 210.174 seconds
vanshitagarg@Vanshitas-MacBook-Air Please % bash interface.sh D cmed2.dat dmed2.dat
interface.sh: line 2: module: command not found
Time taken: 0.693 seconds
vanshitagarg@Vanshitas-MacBook-Air Please % bash interface.sh V dmed2.dat D_medium2.dat
interface.sh: line 2: module: command not found
No Error
vanshitagarg@Vanshitas-MacBook-Air Please %
```

D_medium2.dat (On MacM1)



```
No. of patterns mined 1094
Processing support : 0.00749987
No. of patterns mined 1593
Processing support : 0.00499987
No. of patterns mined 3192
Processing support : 0.00249987
No. of patterns mined 10180
Initial Items: 3960507
Final Items: 2078049
Amount of Compression: 47.5307
Time taken: 475.68 seconds
[ee3200144@chas004 ~/Trial11/Trial9]
$
```

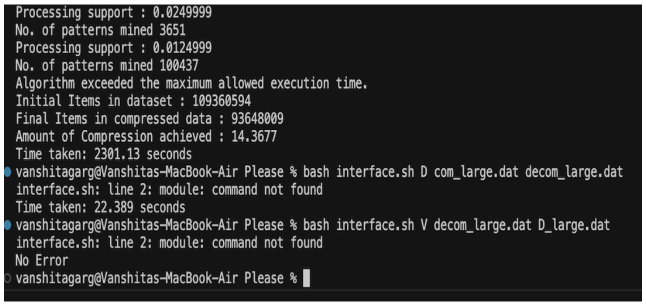
D_medium2.dat (On HPC)
(for the same algo on mac,it runs better)



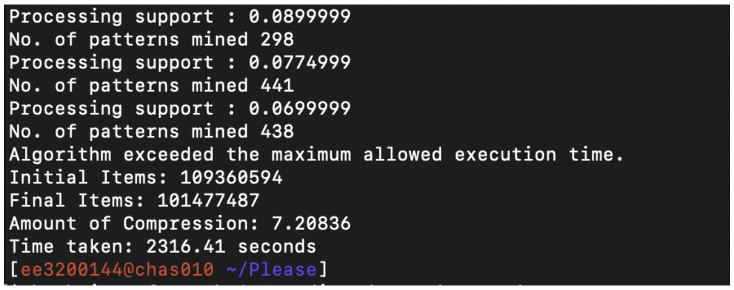
```
No. of patterns mined 759
Processing support : 0.00149987
No. of patterns mined 1335
Processing support : 0.00099987
No. of patterns mined 1367
Processing support : 0.00049987
No. of patterns mined 17737
Initial Items in dataset : 118252
Final Items in compressed data : 14033
Amount of Compression achieved : 88.133
Time taken: 0.978 seconds
vanshitagarg@Vanshitas-MacBook-Air Please %
```

D_small.dat

2) D_large.dat



```
Processing support : 0.0249999
No. of patterns mined 3651
Processing support : 0.0124999
No. of patterns mined 100437
Algorithm exceeded the maximum allowed execution time.
Initial Items in dataset : 109360594
Final Items in compressed data : 93648009
Amount of Compression achieved : 14.3677
Time taken: 2301.13 seconds
vanshitagarg@Vanshitas-MacBook-Air Please % bash interface.sh D com_large.dat decom_large.dat
interface.sh: line 2: module: command not found
Time taken: 22.389 seconds
vanshitagarg@Vanshitas-MacBook-Air Please % bash interface.sh V decom_large.dat D_large.dat
interface.sh: line 2: module: command not found
No Error
vanshitagarg@Vanshitas-MacBook-Air Please %
```



```
Processing support : 0.0899999
No. of patterns mined 298
Processing support : 0.0774999
No. of patterns mined 441
Processing support : 0.0699999
No. of patterns mined 438
Algorithm exceeded the maximum allowed execution time.
Initial Items: 109360594
Final Items: 101477487
Amount of Compression: 7.20836
Time taken: 2316.41 seconds
[ee3200144@chas010 ~/Please]
```

| | Compression % | Compression Time |
|---|---|---|
| Results on MacM1 | 14.3677 % | 2301.13 seconds |
| Results on HPC | 7.20836 % | 2316.41 seconds |

We observed that M1 had better speed hence greater compression in lesser time with the same algorithm. Note this happened only for  D_large dataset. For all the rest the results were still comparable.

## Declaration

1) Besides the references used for FPtree construction and mining, we also used the above mentioned references for seeing the syntax of input/output streaming along with usage of ChatGPt and stack overflow to see specific C++ methods and functionalities like std::include, comparator functions, etc.