COL761 ASSIGNMENT-1

Data Compression Using FP Growth Algorithm

Aaryan Goyal *(2020EE10454)*
Atif Anwer *(2020EE10479)*

August 20, 2023

Contents

1 Basic Algorithm

Our basic objective for the assignment was to write an efficient data compression and decompression algorithm with proper heuristics to decide on the **minimum support threshold**.

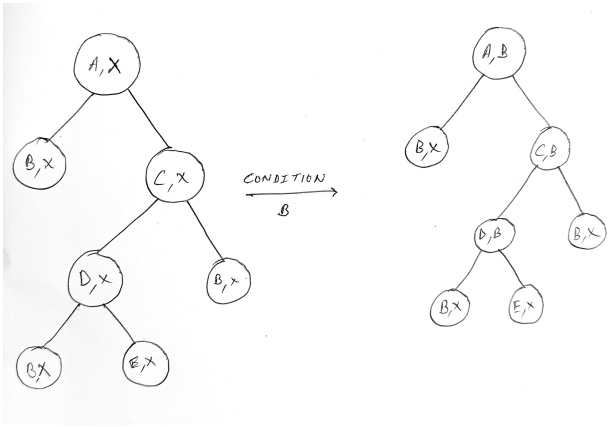
For this purpose, we **Frequent Pattern (FP)** mining technique and implemented an FP-Tree (Taking reference from a crude implementation found on GitHub).

2 Algorithmic Details & Optimizations

2.1 FP Tree

The implementation obtained from GitHub reconstructed transactions from the FP-Tree to create every conditional FP-Tree. This can become extremely inefficient when the depth of the tree increases in the case of large datasets. Hence, we came up with an optimized FP-growth and recovery algorithm to reuse the original FP-tree for all conditional trees in every recursion level. This implementation is structured as follows :

- Nodes in the FP-Tree are wrapped with a node-wrapper *FPNodeWrapper*, which consists of a shared pointer to the node and shared pointers to the next and last Node-Wrapper in the linked list of the header table(the nodes itself were stripped of the pointers in the linked list to allow reuse).
- The header table maps the item value to a *FPNodeWrapper*, which represents the start node of the concerned linked list of nodes in the FP-Tree.
- Every node in the FP-Tree also stores the *conditional frequency* and value of the item which it has been conditioned to. This will help in the recovery process to undo the conditioning.



- To condition a tree with a given item, the algorithm iterates through the linked list of Node-Wrappers header table and for each node, the frequency is bubbled-up(updated on the conditional frequency) along with updation of the condition value. After this, a new header table is constructed for nodes with the given conditioning using a simple Depth-First Search.
- To recover the fptree from conditioning, we iterate through the linked list of the header table for the conditioned item, and the conditional frequencies are updated for the concerned nodes by bubbling up the conditional frequencies of the children that were part of the tree before conditioning(This information is obtained from the initial conditioning and the value of the current conditioning).

- All the references were created dynamically with shared and weak pointers to prevent memory leaks and allow efficient memory usage for deep recursions.

2.2 Data Processing (Compression & Decompression)

2.2.1 Compression

For this step, we wrote a method which took in as input the frequent patterns generated by the FP Tree, and the transaction database & created the compressed .dat file.

To do this, we scan through each transaction (sorted by decreasing order of frequency), and keep adding items to a pattern as long as we find it as a frequent pattern in the mapping. Once, we find that some pattern is not frequent, we remove the last added item and add a value corresponding to this pattern to the new compressed transaction and start over with an empty pattern for the remaining items. Also, for each transaction, after generating the compressed transaction, we write it to the new compressed file instead of storing it somewhere else to minimize memory usage.

In the compressed file, the first line contains a single integer which enumerates the number of transactions in the file (say n). The next n lines are the compressed transactions. All the remaining lines after that are the frequent patterns from the FP Tree. For these, the first element of the line is the mapping value while rest of the values in the line make up the frequent pattern.

2.2.2 Decompression

For this step, we read the compressed data file, to generate the compressed transactions and the frequent pattern mapping. After this, we simply iterate over all the transactions to generate the decompressed data file with the help of values from the frequent pattern mapping.

2.2.3 Some Key Optimizations

- Instead of computing the frequency of each item in the transaction database again (to sort each transaction in order of decreasing order of item frequency), we save this frequency map when we do this initially during construction of FPTree and store it as a parameter of the tree to access while data compression.
- As mentioned in the class and on Piazza, we ignore the cases of repeated items in a transaction due to which we are able to construct the decompressed transaction in a single pass of the transaction.
- For the mapping values, we use negative integers as our transaction database only contains positive integers. This helps decreasing the complexity during compression as it skips the time taken to find all the positive integers not used in the transaction (as there are no negative integers in the transaction database, we don't get stuck in ambiguous mapping and can just use every negative number)
- As the items in a transaction are pre-processed to be ordered in decreasing order of frequency, once we find singleton patterns with frequency less than the minimum support threshold, that means we are at a stage where no frequent pattern can exist (as even single items are not frequent). So, once we reach this stage, we break out of the while loop and just concatenate the remaining transaction as it to the compressed transaction.
- The patterns of size 1 are not used as mapping as they just consist of a single element and instead those single elements are directly used.

- While writing to the compressed .dat file, we just store the mapping of patterns which are used in the compressed transactions instead of all the patterns bringing down the mapping size written to the .dat file by a very significant amount, from **O(n)** to **O($log_2 n$)**

2.3 Heuristics For Deciding Minimum Support Threshold

After trying out many possibilities, the heuristic to decide the minimum support threshold we finally went with was as follows:

- We first calculate the total number of items across all the transactions in the database while also maintaining the count of each item in a map.
- Now, we find the average frequency of all items (say i) for which the following is true:

$$frequency_{Item_i} > \frac{total\_items\_count}{number\_of\_unique\_items}$$

- The value we get out of this, we use the closest integer to it as the threshold for the frequent pattern mining.

3 Experiments & Results

NOTE: All the results below have been computed on the interactive mode of HPC, giving a bit slower results than actual benchmark values.

3.1 Benchmarking server

```
[ee1200479@chas140 ~]
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Model name:            Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz
Stepping:              2
CPU MHz:               2496.898
BogoMIPS:              4993.79
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0-5,12-17
NUMA node1 CPU(s):     6-11,18-23
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush d
ts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon pebs bts r
ep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est t
m2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave
 avx f16c rdrand lahf_lm abm epb invpcid_single intel_ppin ssbd ibrs ibpb stibp tpr_shadow vnmi flexprio
rity ept vpid fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid cqm xsaveopt cqm_llc cqm_occup_llc dt
herm arat pln pts md_clear spec_ctrl intel_stibp flush_l1d
```

3.2 *D_small* Dataset

- Minimum Support Threshold = 1500
- Compression time = 35.28 sec
- Decompression time = 0.07 sec
- Compression $\left(100 - \frac{\text{\#integers in compressed file} \times 100}{\text{\#integers in raw file}}\right)$ = 49.21%

3.3 *D_medium* Dataset

- Minimum Support Threshold = 4500
- Compression time = 31.22 sec
- Decompression time = 2.41 sec
- Compression $\left(100 - \frac{\text{\#integers in compressed file} \times 100}{\text{\#integers in raw file}}\right)$ = 17.75%

3.4 *D_medium2* Dataset

- Minimum Support Threshold = 1000
- Compression time = 335.68 sec
- Decompression time = 14.78 sec
- Compression $\left(100 - \frac{\text{\#integers in compressed file} \times 100}{\text{\#integers in raw file}}\right)$ = 19.08%

3.5 *D_large* Dataset

- Minimum Support Threshold = 60000
- Compression time = 2137.75 sec
- Decompression time = 23.27 sec
- Compression $\left(100 - \frac{\text{\#integers in compressed file} \times 100}{\text{\#integers in raw file}}\right)$ = 5.48%

4 References

[1] Crude FP Growth Implementation - *https://github.com/integeruser/FP-growth*
