Department of Computer Science & Engineering

A1: TRANSACTIONAL DATA COMPRESSION

Abhinav Barnawal *Shreyansh Singh* *Si Siddhanth Raja*
: *2020CS50415* *2020CS10385* *2020CS50443*
Class: *2301-COL761*
Session: *2023-24*
: *cs5200415@iitd.ac.in* *cs1200385@iitd.ac.in* *cs5200443@iitd.ac.in*

Course: *Data Mining* – Instructor: *Prof. Sayan Ranu*
Submission date: *August 20, 2023*

Approaches

Naive FP Growth

We implemented from scratch the naive version of FP-Growth Algorithm to find the frequent item sets among the transactions.

1. We build the complete FP-Tree because it is a prefix tree that fantastically captures the whole set of transactions by ordering the items by their frequency. This helps us manage better the huge set of transactions and capture repetitions efficiently.
2. The FP-Growth algorithm tries to mine every possible frequent item-set among the transactions. We map these itemsets to single integers inorder to reduce the space occupied by these frequent patterns.
3. Once we build a map of encoding, we scan transactions one-by-one and greedily apply the longest encoding from the start until either every element of the transaction is encoded or there exists no encoding for the remaining elements.
4. We, then, write each transaction in encoded form in the compressed file followed by the encodings.

With the naive approach, we ran experiments and obtained the following results:

File Size	Compression
336 KB	30%
16 MB	3%
32 MB	5%

Modified Pattern Mining

We implemented a modified version of FP-Tree Algorithm in order to fit within the time constraints while achieving a decent compression. The naive FP-Growth algorithm has exponential time complexity and hence is infeasible over large files. So, we modified the algorithm and rather invented new methods that suit well to our problem at hand.

1. We build the complete FP-Tree.
2. The FP-Growth algorithm takes asymptotically exponential amount of time. Therefore, we implement just a prefix based encoding following the heuristic that the most frequent item-sets will most likely be near the root of the tree owing to the frequency-based ordering of items.
3. Once we build a map of encoding, we scan transactions from the FP-Tree and apply the same greedy technique to encode the transaction. Note that the transactions generated from the FP-Tree will have elements ordered in decending order of their frequency. We also maintain a record of encodings that are actually used, using a bitmap.
4. We, then, write each transaction in encoded form in the compressed file followed by only the used encodings.

With our first approach, we ran experiments with a Minimum Support Value of 20 and obtained the following results:

File Size	Compression
336 KB	50%
16 MB	7%
32 MB	23%
512 MB	6%

Further Improvements

With not very satisfactory results, we explored the caveats in our approach and improved them on the following lines:

1. We analyzed that since we did not enocde the less frequent items that were down the FP-Tree, there were no encodings at the end of the transactions. Therefore, we also build a reverse tree such the the least frequent items are mostly near the root of the tree and create encodings with a lesser Minimum Support Value (one-fourth of the original). We then use these new encodings to encode transactions from the back. Not to our surprise, we got a consistent $\sim 4\%$ improvement in compression almost irrespective of the file size as demostrated below. However, we could not run the largest data set due to time constraints.

File Size	Compression
336 KB	55%
16 MB	11%
32 MB	27%
512 MB	–

2. In our initial approach, mining only the longest frequent prefixes pruned a lot of frequent item-sets away from the root, due to which we did not get enough encodings to encode the transactions, which led to low compression. To counter that, we created a **Residual Tree** such that every discarded node of the FP-Tree that it not encoded due to its low support is merged into an initially empty tree, the residual tree. The new tree collects all such discarded subtrees (collection of itemsets) in the same manner as a prefix tree. We then mine the residual tree to get more encoding to enrich our encodings map. As a result, we get $\sim 2\times$ more encodings in the map than with our naive method and hence a significantly improved compression without much extra running time.

File Size	Compression
336 KB	74%
16 MB	17%
32 MB	33%
512 MB	15%

Acknowledgement

Our sincere thanks to Prof. Sayan Ranu for his expertise and guidance that led to the completion of this work. We also thank the respected Teaching Assistants for their immense support and almost instant replies to our doubts, without whom it would have never been possible to present this work.