# COMP 4170 Data Mining Project

David Horsman

Kyle Shewchuk

Contents

[COMP 4170 Data Mining Project 1](#_Toc376000893)

[Description of the Algorithm 3](#_Toc376000894)

[How to Run the Algorithm 4](#_Toc376000895)

[Experimental Results 5](#_Toc376000896)

[Observations 6](#_Toc376000897)

[Suggestions 7](#_Toc376000898)

[Citation / References 8](#_Toc376000899)

# Description of the Algorithm

## (may include high-level pseudocode) and key data structures

We have implemented the FP-Growth algorithm. The program starts by collecting all the transactions. Each transaction is stored in a vector of ints and the whole group of transactions is stored in a vector, resulting in nested vectors. While the transactions are being collected from file it also builds the header table, which is a map of item to count. Here is pseudocode:

for (items in line)

{

push item to transaction

headerTable[item]++

}

push transaction to transactions

After everything is collected from file and the header table is built, the header table is sorted by item count. This is done by moving the map of item to count into a vector containing pairs of items and counts since vectors can be sorted.

and then the transactions are sorted by the ordering in the header table.

It uses a tree data structure to

# How to Run the Algorithm

## (e.g., command line)

1. Compile fpgrowth.cpp
   * g++ fpgrowth.cpp
2. Run the output
3. Follow the prompts; you will need to enter a filename that contains data in the same format as the precise examples provided for us and the minimum support.

# Experimental Results

## (with tables/figures)

# Observations

## (e.g., when the algorithms are good or bad? what are the observed/potential problems with the algorithm?)

1. It is a bit more work to build the header table as a map and then transfer it into a vector to be sorted.
2. To sort the transactions by the header table ordering, we are going through every item in the header table on each transaction and checking if it exists in that transaction instead of just looking at what exists in the transaction and ordering it as they appear in the header table.

Completeness

􀂃 Preserve complete information for frequent pattern mining

􀂃 Never break a long pattern of any transaction

Compactness

􀂃 Reduce irrelevant info—infrequent items are gone

􀂃 Never be larger than the original database (not counting node-links & the frequency field)

􀂃 Items in descending frequency order: The higher the frequencies of items, the higher is the likelihood for them to be shared

Divide-and-conquer:

􀂃 Decomposes both the mining task & DB according to the

frequent patterns obtained so far

􀂃 Leads to focused search of smaller databases

Other factors

􀂃 No candidate generation, no candidate test

􀂃 Compressed database: FP-tree structure

􀂃 No repeated scan of entire database

􀂃 Basic ops: Counting local frequent items and building

FP-subtree, no pattern search and matching

# Suggestions

## (e.g., how to overcome the observed problems/challenges?)

1. The alternative of building the header table as a vector of pairs where each pair is an item and its count would not necessarily be more efficient due to the unknown ordering of the pairs. If the first transaction contained 4 5 and the second contained 1 5, the entire vector would need to be checked for 1 and then 1 added after the 4 and 5 when it wasn't found. Next the entire vector would need to be checked for 5 and then the count for 5 would need to be increased when found.

Another method could be putting each item in the corresponding space in the vector, so 0 would be unused and headerTable[1] would be item 1.

# Citation / References

Leung, C. K. (2013). Unit 03 – Tree Based ARM [PDF]. Retrieved from http://courses.cs.umanitoba.ca/index.aspx?sec=6235