Node.js Implementation of FP-Growth and FP-Growth*

FP-Growth

FP-Growth is a frequent pattern mining algorithm proposed by Jiawei Han[1].It was proven to outperform other frequent pattern mining techniques such as the Apriori algorithm. It does so by using a divide and conquer strategy along with the FP-tree data structure to generate frequent patterns without candidate itemset generation. The FP-Tree is comprised of a set of linked nodes with an associated tree header. Each branch of the tree represents a potential frequent pattern and patterns with the identical prefixes are compressed. After its initial construction the FP-Tree is divided into smaller and smaller conditional FP-Trees until only a single branch and single set of frequent patterns remains.

FP-Growth*

In the paper “Fast algorithms for frequent itemset mining using FP-trees”[2], the authors found that the majority of the work being done (~80% of cpu time) by the FP-Growth algorithm was spent traversing trees. Whenever a new FP-Tree T_x is being constructed two traversals of the ancestor FP-Tree T are required. The first to generate the frequent one-itemsets in the conditional pattern base of T, and the second to construct the new FP-Tree T_x.

Thus the paper introduces a new data data structure to FP-Growth called the FP-Array. The FP-Array is a 2 dimensional (m - 1) by (m - 1) matrix where m = the number of frequent one-itemsets in the FP-Tree. Each cell corresponds to the count of a pair of items (i_j, i_k). Since the order of (i_j, i_k) is irrelevant only pairs where k < j are stored reduces the amount of space needed to store the FP-Array by more than half. Using the fp array the set of frequent one itemsets for any i_j is found by simply reading the row at j. With this the first traversal of the tree is unnecessary. Limitations of the FP-Growth* algorithm have to do with the amount of memory required for the FP-Array and, limited benefit when used on dense datasets.

---

[1] "Mining Frequent Patterns without Candidate Generation." Accessed December 1, 2017. https://www.cs.sfu.ca/~jpei/publications/sigmod00.pdf.

[2] "Fast algorithms for frequent itemset mining using FP-trees - IEEE Xplore." Accessed December 1, 2017. http://ieeexplore.ieee.org/document/1501819/.

Implementation

Using Node.js a server side javascript framework i implemented the following algorithms. Both FP-Growth and FP-Growth* use the same FP-Tree with **root**, **base**, and **header** attributes. **root** is a reference to the root; **base** consists of a pattern suffix, and **header** contains all frequent one-itemsets and a link to each occurrence of an item for they FP-Tree it they are associated with.

The first step is identical for the two algorithms aside from the construction of the FP-Array in FP-Growth*.

Inputs: database= initial database of tuples to mine, **minSup =** minimum support
Variables: oneSet = collection of frequent one itemsets.
Outputs: initial FP-Tree
Procedure ConstructFP-Tree(database, **minSup)**

```
1       For each tuple t in the database
2               For each Attribute a in t
3                       If a != exist in oneSet
4                               Insert a into oneSet
5                       Else
6                               Increment a's count
7                       End if
8               End for
9       End for

10      Prune all elements of oneSet that have a count < minSup and
        sort by decreasing count

11      Create the FP-Tree T  with its root labeled by a sentinel value

12      For each tuple t in database
13              sort t by the item order of oneSet and remove items not in oneSet
14*             Call FParrayInsert(T.FParray, t) increments the cell for all 2 item pairs in
t
15              Call FPinsert(T, t) which inserts all items from t recursively
16      Endfor
End
```

Line 14 is only used in FP-Growth

In my implementation i wished to reduce the amount of time needed when searching for an item in the array. To do with i created 2 (key, value) pair dictionaries with the key being the item's name and value being the index of the item in FPArray. Then by accessing the dictionary at the key i get the index of the item in the array.

The following procedures recursively generates all frequent items for an FP-Tree.

Inputs: T = FP-Tree to mine frequent patterns from
Variables: oneSet = collection of frequent one itemsets. **minSup** = minimum support
Outputs: All frequent patterns of **T**
Procedure FP-Growth(T) and FP-Growth*(T)[1]

```
1       if contains a single path P
2               For every subset p of P
3                       output frequent itemset (p U T.base) with minimum support of p
4               End for
5       else
6               For each i in T.header
7                       output frequent itemset Y = T.base U i with support = i.support
8                       initialize new conditional FP-Tree T_Y
9                       if T.FParray is defined
10*                             create T_Y's header from T.FParray at i
11                      else
12                              create T_Y's header by scanning T
13                      end if
14                      construct T_Y and T_Y.FParray using T as a conditional base
15                      if T_Y is not empty
16                              call FP-Growth(T)
17                      end if
18              End for
19      End
```

*Line 10 and its associated if statement is not needed in normal FP-growth as it relates to the creation of the FParray

Evaluation

Based on the reduced cpu time of FP-Growth* that the paper claims. I chose to evaluate execution time of generating frequent-itemsets for FP-Growth, and FP-Growth*. Using a database generated using Spotify's web API, that has tuples with 5 attributes on average, and sizes of 1178 to . Each algorithm was run 4 times "execution time" in the table below is the average of those results.

Results

Due to an error in the implementation of one of the algorithms the number of frequent patterns generated is not identical on when the database inputted has more than a paltry number of tuples. Because my implementation of FP-Growth* generates less patterns than FP-Growth the results of my evaluation loses most meaning, as the differences in execution time can be explained by the smaller set generated. If the algorithm was operating correctly i would only expect a small increase in speed with the smaller more dense datasets that increases as the dataset becomes larger and more sparse, since FP-Growth* improvements are best shown on sparse datasets.

This error could relate to the way that headers are initialized using the FParray. The paper specifies that the header can be initialized using only a row of the FParray and that is the way i implemented it. I believe that was probably only for the one specific case mentioned in the paper and that i should be using the columns that relate to the item as well. Another likely candidate is the minimum support pruning method used, as when the minimum support increases so does the discrepancy in number of patterns generated. Due to time constraints i was unable to test these hypotheses.

With the database size of 10200 and minimum support of 8, FP-Growth* generated almost half as many frequent itemsets. As of the time i am writing this paper i am unsure why the discrepancy is so large for that instance.

FP-Growth FP-Growth*

Database Size	Minimum support	Number of Freq Patterns	Execution time in ms	Percentage difference (time)
447	4	686	9.47	9.16%
		620	8.64	
1178	4	5740	31.83	8.48%
		4368	29.24	
5505	6	14355	82.85	14.86%
		8544	71.39	
10200	6	101492	319.24	16.42%
		76389	270.79	
10200	8	49121	196.50	32.69%
		25730	141.28	
14941	8	138852	411.1	15.71%
		100183	351.19	
14941	10	75918	284.26	19.77%
		93194	346.63	
28987	10	148323	665.27	34.44%
		122790	469.79	
28987	20	28669	338.28	22.14%
		21235	270.84	

Conclusion

Over the course of the project i learned a substantial amount about tree generation and traversal in Javascript. The language is ill suited for these kind of operations due to limited access to memory pointers. Another takeaway is that the generation of basic frequent itemsets gives limited information to me as a human due to the sheer amount of data that is created. It would be much better to generate maximal or closed frequent itemsets if your goal is human readable information. I attempted an FPMax* implementation aswell but unfortunately it was unsuccessful.

At this point i feel i have a much better grasp of how frequent itemsets are generated by the FP-Growth algorithm than i did at the start. I was surprised by how much memory the algorithm used on larger datasets ~30 000 tuples i often came close to using 8 Gigs of memory at the deepest levels of recursion.

Some interesting results of the frequent pattern mining

- From my dataset the longest frequent patterns were almost always from classical genres this is likely due to the fact that songs often have identical titles appended with “variations” or the key they are performed.
- The word love occurs in quite frequently in song titles of several genres.
- The highest support 2 item set from ~30 000 records with min support of 5 was
    - (genre: classical, not popular (30 - 39) | support: 1424)
      (popularity is a 0 - 100 scale)
- Other interesting high support itemsets
    - (remastered, genre: rock | support: 278)
- From a smaller dataset that only contained song names
    - (1998, remastered | support: 38)
    - (Jamaican, version | support: 20)
    - (Theme, of | support: 24)

Im sure there are many more but it is difficult to find interesting patterns that are longer than 2 elements as when they are longer the frequent itemsets become flooded with permutations of the same song titles.