Frequent Pattern Growth (FP-Growth) Algorithm

An Introduction

Copyright 2008 Florian Verhein. Most figures derived from [1].

Outline

Introduction

FP-Tree data structure

Step 1: FP-Tree Construction

Step 2: Frequent Itemset Generation

Discussion
Introduction

- **Apriori**: uses a generate-and-test approach – generates candidate itemsets and tests if they are frequent
  - Generation of candidate itemsets is expensive (in both space and time)
  - Support counting is expensive
    - Subset checking (computationally expensive)
    - Multiple Database scans (I/O)
- **FP-Growth**: allows frequent itemset discovery without candidate itemset generation. Two step approach:
  - **Step 1**: Build a compact data structure called the *FP-tree*
    - Built using 2 passes over the data-set.
  - **Step 2**: Extracts frequent itemsets directly from the FP-tree
    - Traversal through FP-Tree

Core Data Structure: FP-Tree

- Nodes correspond to items and have a counter
- FP-Growth reads 1 transaction at a time and maps it to a path
- Fixed order is used, so paths can overlap when transactions share items (when they have the same prefix).
- In this case, counters are incremented
- Pointers are maintained between nodes containing the same item, creating singly linked lists (dotted lines)
- The more paths that overlap, the higher the compression. FP-tree may fit in memory.
- Frequent itemsets extracted from the FP-Tree.
Step 1: FP-Tree Construction (Example)

FP-Tree is constructed using 2 passes over the data-set:

- **Pass 1:**
  - Scan data and find support for each item.
  - Discard infrequent items.
  - Sort frequent items in decreasing order based on their support.
    - For our example: \(a, b, c, d, e\)
    - Use this order when building the FP-Tree, so common prefixes can be shared.

- **Pass 2:** construct the FP-Tree (see diagram on next slide)
  - Read transaction 1: \(\{a, b\}\)
    - Create 2 nodes \(a\) and \(b\) and the path \(null \rightarrow a \rightarrow b\). Set counts of \(a\) and \(b\) to 1.
  - Read transaction 2: \(\{b, c, d\}\)
    - Create 3 nodes for \(b, c\) and \(d\) and the path \(null \rightarrow b \rightarrow c \rightarrow d\). Set counts to 1.
    - Note that although transaction 1 and 2 share \(b\), the paths are disjoint as they don’t share a common prefix. Add the link between the \(b\)’s.
  - Read transaction 3: \(\{a, c, d, e\}\)
    - It shares common prefix item \(a\) with transaction 1 so the path for transaction 1 and 3 will overlap and the frequency count for node \(a\) will be incremented by 1. Add links between the \(c\)’s and \(d\)’s.
  - Continue until all transactions are mapped to a path in the FP-tree.
Step 1: FP-Tree Construction (Example)

FP-Tree size

- The FP-Tree usually has a smaller size than the uncompressed data – typically many transactions share items (and hence prefixes).
  - **Best case scenario**: all transactions contain the same set of items.
    - 1 path in the FP-tree
  - **Worst case scenario**: every transaction has a unique set of items (no items in common)
    - Size of the FP-tree is *at least* as large as the original data.
    - Storage requirements for the FP-tree are higher – need to store the pointers between the nodes and the counters.
- The size of the FP-tree depends on how the items are ordered
  - Ordering by decreasing support is typically used but it does not always lead to the smallest tree (it’s a heuristic).
Step 2: Frequent Itemset Generation

- FP-Growth extracts frequent itemsets from the FP-tree.
- Bottom-up algorithm – from the leaves towards the root
  - Divide and conquer: first look for frequent itemsets ending in \( e \), then \( de \), etc... then \( d \), then \( cd \), etc...
  - First, extract prefix path sub-trees ending in an item(set). *(hint: use the linked lists)*

![Complete FP-tree example](image)

→ **Example:** prefix path sub-trees

Step 2: Frequent Itemset Generation

- Each prefix path sub-tree is processed recursively to extract the frequent itemsets. Solutions are then merged.
  - *E.g.* the prefix path sub-tree for \( e \) will be used to extract frequent itemsets ending in \( e \), then in \( de \), \( ce \), \( be \) and \( ae \), then in \( cde \), \( bde \), \( cde \), etc.
  - Divide and conquer approach

![Prefix path sub-tree ending in e.](image)
Example

Let $minSup = 2$ and extract all frequent itemsets containing $e$.

1. Obtain the prefix path sub-tree for $e$:

2. Check if $e$ is a frequent item by adding the counts along the linked list (dotted line). If so, extract it.
   - Yes, count = 3 so \{e\} is extracted as a frequent itemset.

3. As $e$ is frequent, find frequent itemsets ending in $e$. i.e. $de$, $ce$, $be$ and $ae$.
   - i.e. decompose the problem recursively.
   - To do this, we must first to obtain the conditional FP-tree for $e$.

Conditional FP-Tree

- The FP-Tree that would be built if we only consider transactions containing a particular itemset (and then removing that itemset from all transactions).
- **Example**: FP-Tree conditional on $e$.
To obtain the *conditional FP-tree* for *e* from the *prefix sub-tree* ending in *e*:

- Update the support counts along the prefix paths (from *e*) to reflect the number of transactions containing *e*.
  - *b* and *c* should be set to 1 and *a* to 2.

- Remove the nodes containing *e* – information about node *e* is no longer needed because of the previous step.
Conditional FP-Tree

To obtain the conditional FP-tree for \( e \) from the prefix sub-tree ending in \( e \):

- Remove infrequent items (nodes) from the prefix paths
- \textbf{E.g.} \( b \) has a support of 1 (note this really means \( be \) has a support of 1). i.e. there is only 1 transaction containing \( b \) and \( e \) so \( be \) is infrequent – can remove \( b \).

\begin{itemize}
  \item \textbf{Question:} why were \( c \) and \( d \) not removed?
\end{itemize}

\textbf{Example (continued)}

- 4. Use the the conditional FP-tree for \( e \) to find frequent itemsets ending in \( de \), \( ce \) and \( ae \)

  - Note that \( be \) is not considered as \( b \) is not in the conditional FP-tree for \( e \).
  - For each of them (e.g. \( de \)), find the prefix paths from the conditional tree for \( e \), extract frequent itemsets, generate conditional FP-tree, etc... (recursive)
  - \textbf{Example:} \( e \rightarrow de \rightarrow ade \) (\( \{d,e\}, \{a,d,e\} \) are found to be frequent)
Example (continued)

4. Use the conditional FP-tree for e to find frequent itemsets ending in de, ce and ae

Example: $e \rightarrow ce$ ($\{c, e\}$ is found to be frequent)

- etc... ($ae$, then do the whole thing for $b$, etc)

Result

- Frequent itemsets found (ordered by suffix and order in which they are found):

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Frequent Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>{e}, {d,e}, {a,d,e}, {c,e}, {a,e}</td>
</tr>
<tr>
<td>d</td>
<td>{d}, {c,d}, {b,c,d}, {a,c,d}, {b,d}, {a,b,d}, {a,d}</td>
</tr>
<tr>
<td>c</td>
<td>{c}, {b,c}, {a,b,c}, {a,e}</td>
</tr>
<tr>
<td>b</td>
<td>{b}, {a,b}</td>
</tr>
<tr>
<td>a</td>
<td>{a}</td>
</tr>
</tbody>
</table>
Discussion

- Advantages of FP-Growth
  - only 2 passes over data-set
  - “compresses” data-set
  - no candidate generation
  - much faster than Apriori

- Disadvantages of FP-Growth
  - FP-Tree may not fit in memory!!
  - FP-Tree is expensive to build
    - Trade-off: takes time to build, but once it is built, frequent itemsets are read off easily.
    - Time is wasted (especially if support threshold is high), as the only pruning that can be done is on single items.
    - support can only be calculated once the entire data-set is added to the FP-Tree.

References

  - Chapter 6: Association Analysis: Basic Concepts and Algorithms