

On the Computational Complexity of Two Frequent Set Generation Algorithms

Alan P. Sprague
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, Alabama 35294-1170

July 5, 2004

Abstract

Frequent set generation is normally the first step toward association rule generation, and is usually considered the bottleneck step. Accordingly, many algorithms to generate frequent sets have been proposed. Evaluation of these algorithms is normally done empirically: their running times have been compared on a few or a substantial number of data files. Evaluation of the computational complexity of these algorithms has not much been pursued except to note that every algorithm requires exponential time, because the output size can be exponential in terms of the input size. We develop an alternate method of evaluating algorithms for frequent set generation: we propose evaluating the algorithms according to their computational complexity on several abstractly defined infinite families of data files. We perform this evaluation for Apriori and FP-growth on two infinite families of data files, and show that on both families, a variant of FP-growth is unboundedly faster than Apriori as the data file size increases.

1 Introduction

In this paper we use computational complexity to shed light on the relative execution speed of frequent set generation algorithms, as the input data file size grows unboundedly. In particular we study Apriori [3, 4, 2] and a variant of FP-growth [7, 6]. This variant differs from the original FP-growth in two ways: (1) the order on items is static, not modified at each conditional tree; (2) conditional trees are constructed directly from the parent tree, not by first constructing the “conditional pattern base” [7, 6]. We assume main memory execution.

Algorithms to generate frequent sets have been a concern because of (1) the importance of frequent sets as the first step toward generating association rules, and (2) the large amount of computation performed by these algorithms. Analysis of the computational complexity of these algorithms has commonly ended with the observation

that execution time is exponential in terms of the input size (because output size is exponential in terms of the input size). A rare paper that pushes beyond this is [5]. Instead, comparison of algorithms has normally been done by comparing the execution time of implementations, on benchmark data sets. A recent systematic initiative in this direction is the workshop “Frequent Itemset Mining Implementations” that took place in November 2003 [1].

We will quickly study the computational complexity of Apriori; the number of incrementations of support will form the basis for the lower bound derived for the computational complexity of Apriori. We will study the computational complexity of a variant of FP-growth. As may be expected, the number of nodes in the FP-trees will play an important role in that complexity.

The trouble is that the number of support incrementations and the number of FP-trees are incommensurate in general. However, they may be related to each other for specific data files and specific families of data files. We will define certain infinite families of data files (each family containing infinitely many data files; each data file is finite). One of these families models certain dense data files: in it, all small (and perhaps medium sized) itemsets are records. The other family models data files arising from rectangular arrays: there are t attributes, and each record contains one value for each of the t attributes. We will show that on both these families, the variant of FP-growth is unboundedly faster than Apriori as the data files grow unboundedly large, unless the size of each record is bounded.

Thus, like comparisons of run time on benchmark files, our comparison is primarily on specific data file families. Unlike comparisons of run time on benchmark files, our method gives clear extrapolations to arbitrarily larger files. This method of evaluating algorithms forms a complement to empirical testing.

After some definitions, we describe the families of data files in Section 2. Section 3 establishes a lower bound on the execution time of Apriori on the two families. A variant of FP-growth and its computational complexity are the topic of Section 4, and the next two sections specialize this to the two families of data files. Sections 7 and 8 contain the major results, and the final section concludes.

1.1 Preliminaries

We will describe a data file as a pair (I, \mathcal{D}) where I is a set of items, \mathcal{D} is a set of *records* (or *transactions*), and each record is a subset of I . (More precisely, since multiple records can contain exactly the same set of items, \mathcal{D} is a list of records rather than a set of records.) The number of records is written $|\mathcal{D}|$; $\text{size}(\mathcal{D}) = \sum\{|r| : r \in \mathcal{D}\}$. A usual, an *itemset* is a subset s of I , and where $|s| = j$, s is also called a *j-itemset*.

We let \mathcal{I} represent the power set of I — the set of all itemsets. The set of j -itemsets is written \mathcal{I}_j .

The *support* of an itemset s is the number of records r such that $s \subseteq r$; we write

this as $\text{suppt}(s)$. s is called a *frequent set* if $\text{suppt}(s) \geq \sigma$, where σ is an integer known as the *minimum support threshold*. We represent the set of all frequent itemsets by \mathcal{F} (or by $\mathcal{F}^{(\sigma)}$ in situations where we are balancing two minimum support thresholds). Also, \mathcal{F}_j or $\mathcal{F}_j^{(\sigma)}$ is the set of all j -itemsets that are frequent: $\mathcal{F}_j = \mathcal{F} \cap \mathcal{I}_j$.

We extend the support notation as follows: where \mathcal{J} is a set of itemsets, $\text{suppt}(\mathcal{J}) = \sum_{s \in \mathcal{J}} \text{suppt}(s)$.

Both Apriori and FP-growth rely on the existence of a linear ordering on items. We let $I = \{a_1, a_2, \dots, a_{|I|}\}$, where $a_i < a_j$ iff $i < j$. Commonly the items are small integers: $a_i = i$.

$O()$, $\Omega()$, and $\Theta()$ have their standard meanings: $O(f(n))$ is the set of functions $g(n)$ such that for some constant c , $g(n) \leq f(n)$ for all large n . When $g(n)$ satisfies this condition, we write $g(n) \in O(f(n))$. Similarly, $\Omega(f(n))$ is the set of functions $g(n)$ such that for some positive constant c , $g(n) \geq cf(n)$ for all large n . Lastly, $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$.

2 The Data Files

In this section we describe two mathematically defined families of data files. Each family has infinitely many data files, of increasing size. Because they are mathematically defined, it is possible to predict the behavior of algorithms on them, and to establish the computational complexity of algorithms on them.

2.1 Limited Power Set

Let n be a positive integer and $1 \leq t \leq n$. The limited power set data file $\mathcal{LP}(n, \leq t)$ is the data file (I, \mathcal{D}) where $|I| = n$, and \mathcal{D} is all itemsets s such that $|s| \leq t$.

We define the integer $\binom{n}{\leq t}$ as follows:

$$\binom{n}{\leq t} = \sum_{j=0}^t \binom{n}{j}.$$

Where (I, \mathcal{D}) is the $\mathcal{LP}(n, \leq t)$ data file, the number of records is $\binom{n}{\leq t}$, and $\text{size}(\mathcal{D}) = \sum_{j=0}^t j \binom{n}{j}$. Since $j \binom{n}{j} = n \binom{n-1}{j-1}$, $\text{size}(\mathcal{D}) = n \binom{n-1}{\leq t-1}$.

Note that if the minimum support threshold σ is set at 1, then the set of frequent sets is equal to \mathcal{D} , so the sum of sizes of frequent sets is $n \binom{n-1}{\leq t-1}$.

Section 3 will make use of the following proposition.

Proposition 1 (a) For $0 \leq j \leq t$, $\text{suppt}(\mathcal{I}_j) = \binom{n}{j} \binom{n-j}{\leq t-j}$.

(b) For $0 \leq j \leq t/2$, $\text{suppt}(\mathcal{I}_j) \geq \text{suppt}(\mathcal{I}_{t-j})$.

Proof. To prove part (a): $\text{suppt}(\mathcal{I}_j) = \binom{n}{j} \binom{n-j}{\leq t-j}$ because there are $\binom{n}{j}$ j -itemsets, each of which is contained in $\binom{n-j}{u-j}$ u -itemsets (for $j \leq u \leq n$), so each is contained in $\binom{n-j}{\leq t-j}$ records of \mathcal{D} .

Proof of (b): By (a), for any j ($0 \leq j \leq t$),

$$\text{suppt}(\mathcal{I}_j) = \sum_{u=j}^t \binom{n}{j} \binom{n-j}{u-j} = \sum_{u=j}^t \binom{n}{u} \binom{u}{j}. \quad (1)$$

Replacing j by $t-j$ in this expression, we have

$$\text{suppt}(\mathcal{I}_{t-j}) = \sum_{u=t-j}^t \binom{n}{u} \binom{u}{t-j}. \quad (2)$$

We are to show that, where $0 \leq j \leq t/2$, the quantity in Equation 1 is at least as large as that in Equation 2. It is sufficient to ignore terms of Equation 1 having $u < t-j$, and show that the remaining terms of Equation 1 are never less than the corresponding terms of Equation 2. I.e., it is sufficient to show that $\binom{u}{j} \geq \binom{u}{t-j}$ for $t-j \leq u \leq t$. This is evident from the following: For any h , $\binom{u}{h} > \binom{u}{j}$ iff h is strictly between j and $u-j$. Since $t-j \geq \max(j, u-j)$ then $t-j$ is not strictly between j and $u-j$, so $\binom{u}{t-j} \leq \binom{u}{j}$. \square

2.2 Cartesian Product

Some naturally arising data files are basically rectangular arrays. Each row of the matrix is a record and each column is an attribute. Hence each record is a sequence of n data values (for some n), each data value being in the domain of that attribute. If each attribute has the same number of data values, the data file has the general structure of the Cartesian product data file that we define next.

Let c and t be positive integers. Let $C = \{1, 2, \dots, c\}$. Let $A = \{1, 2, \dots, t\}$; each member of A will be called an *attribute*. A function $f : A \rightarrow C$ maps each attribute to a value in C ; we wish to consider f as a set of t ordered pairs (i, j) with $i \in A, j \in C$. (This is the standard formal way to view a function.) Define $I = \{(i, j) : i \in A, j \in C\}$, and call members of I *items*. Finally, we let \mathcal{D} be the set of all functions from A to C (so “record of the data file” and “function from A to C ” are equivalent notions). Then $|\mathcal{D}| = |C|^{|A|} = c^t$. Hence $\text{size}(\mathcal{D}) = t|\mathcal{D}| = tc^t$.

(\mathcal{D}, I) will be denoted by $\mathcal{CP}(c, t)$ and be called a *Cartesian product* data file. Where $c = 2$ and $t = 3$ and item $(1,1)$ is written as a_1 , item $(1,2)$ is written a_2 , item $(2, j)$ is written as b_j (for $j = 1, 2$), and $(3, j)$ is written c_j , \mathcal{D} is the set of 8 records

$$\begin{aligned} & \{\{a_1, b_1, c_1\}, \{a_1, b_1, c_2\}, \{a_1, b_2, c_1\}, \{a_1, b_2, c_2\}, \\ & \{a_2, b_1, c_1\}, \{a_2, b_1, c_2\}, \{a_2, b_2, c_1\}, \{a_2, b_2, c_2\}\}. \end{aligned}$$

In a 3-question questionnaire where attribute 1 is gender (male or female), attribute 2 is education (low, high), and attribute 3 is income (low, high), the 8 records in \mathcal{D} correspond to the results of the polling of 8 respondents where by some coincidence the 8 respondents were distributed equally among the 8 possible outcomes.

Where the minimum support threshold is set at 1, the number of frequent sets of size j is $\binom{t}{j}c^j$, so the number of frequent sets is $\sum_{j=0}^t \binom{t}{j}c^j = (c+1)^t$. Also, the sum of sizes of frequent sets is $\sum_{j=0}^t j \binom{t}{j}c^j = tc(c+1)^{t-1}$. (This may be justified as follows: $\sum_{j=0}^t j \binom{t}{j}c^j = \sum_{j=1}^t t \binom{t-1}{j-1}c^j = tc \sum_{j=1}^t \binom{t-1}{j-1}c^{j-1} = tc(c+1)^{t-1}$.)

$\text{suppt}(\mathcal{F}_k^{(1)}) = \binom{t}{k}c^k$ because the number of frequent k -sets is $\binom{t}{k}c^k$, each having support c^{t-k} . (The support of each frequent k -set s is c^{t-k} because s has a determined value in k attributes; there are $t-k$ attributes on which the value is not determined, so there are c^{t-k} ways to extend s to a function from A to C .) Finally, $\text{suppt}(\mathcal{F}^{(1)}) = 2^t c^t$ (by computing the sum $\sum_{k=0}^t \text{suppt}(\mathcal{F}_k^{(1)})$). (Alternately, $\text{suppt}(\mathcal{F}^{(1)}) = 2^t c^t$ because each of the c^t records contributes 1 to the support of 2^t itemsets.)

3 A lower bound for Apriori

In this section we develop a lower bound on the computational complexity of Apriori. This bound takes two forms: one for arbitrary minimum support threshold σ , and one specifically for $\sigma = 1$. From this we will obtain a lower bound for the two data file families.

Apriori's method of generating frequent sets is to generate candidates and then to read the data file to compute the support of the candidates. It computes the support of candidates by initializing support at zero, and then incrementing support whenever a record containing the candidate is encountered. Accordingly, the computational complexity of Apriori is at least as great as the number of incrementations that are performed, namely $\sum_{s \in \mathcal{F}} \text{suppt}(s)$. (This ignores candidates that are not frequent.) We state this as a proposition:

Proposition 2 *Using Apriori, the execution time to generate all frequent sets is $\Omega(\text{suppt}(\mathcal{F}))$.*

Where $\sigma = 1$, this takes the following form:

Corollary 3 *Where the minimum support threshold σ is 1, the execution time of Apriori is $\Omega(\sum_{r \in \mathcal{D}} 2^{|r|})$.*

Proof. This follows from Proposition 2 by counting pairs $\{(r, s) : r \in \mathcal{D}, s \in \mathcal{F}, s \subseteq r\}$, as follows.

Organize the count by frequent set s : s is the second member of $\text{suppt}(s)$ pairs. Therefore the number of pairs is $\sum \{\text{suppt}(s) : s \in \mathcal{F}\}$.

Organize the count by record r : each record r is the first member of $2^{|r|}$ pairs, since (where $\sigma = 1$) every subset of r is in \mathcal{F} .

Thus, $\sum\{\text{suppt}(s) : s \in \mathcal{F}\} = \sum_{r \in \mathcal{D}} 2^{|r|}$. \square

We note that both bounds hold for any frequent set generation algorithm that determines support of a frequent set by a succession of incrementations.

Also, both bounds hold regardless of whether Apriori uses or omits pruning while generating candidates.

Let \mathcal{D} be the data file $\mathcal{CP}(c, t)$. We will compute a lower bound on the computational complexity of Apriori on \mathcal{D} , where the minimum support threshold $\sigma \leq c^{t/2}$. We start with a lemma.

Lemma 1 *If $\mathcal{D} = \mathcal{CP}(c, t)$ and $\sigma \leq c^{t/2}$, then $\text{suppt}(\mathcal{F}^{(\sigma)}) \geq \text{suppt}(\mathcal{F}^{(1)})/2$.*

Proof. Let s be a k -itemset, and s be contained in at least one record (hence the items of s are from distinct attributes of the Cartesian product file). In Section 2.2 it was established that $\text{suppt}(s) = c^{t-k}$. Hence if $k \leq t/2$ then $\text{suppt}(s) \geq c^{t/2}$, so $\text{suppt}(s) \geq \sigma$. In conclusion, if $s \in \mathcal{F}^{(1)}$ and $|s| \leq t/2$ then $s \in \mathcal{F}^{(\sigma)}$.

From Section 2.2, $\text{suppt}(\mathcal{F}_k^{(1)}) = c^t \binom{t}{k}$. Hence $\text{suppt}(\mathcal{F}^{(1)})$ is the sum of values on row t of the Pascal triangle (multiplied by c^t), while $\text{suppt}(\mathcal{F}^{(\sigma)})$ is the sum of values on the left half of row t of the Pascal triangle (multiplied by c^t). Hence $\text{suppt}(\mathcal{F}^{(\sigma)}) \geq \text{suppt}(\mathcal{F}^{(1)})/2$. \square

Proposition 4 *If $\mathcal{D} = \mathcal{CP}(c, t)$ and $\sigma \leq c^{t/2}$, then the time complexity of Apriori is $\Omega(2^t c^t)$.*

Proof. By Lemma 1 and Section 2.2, $\text{suppt}(\mathcal{F}^{(\sigma)}) \geq \text{suppt}(\mathcal{F}^{(1)})/2 = 2^t c^t / 2$. Since $\text{suppt}(\mathcal{F}^{(\sigma)})$ is a lower bound on the computational complexity of Apriori, then Apriori takes $\Omega(2^t c^t)$ time. \square

Next we establish a lower bound on the time complexity of Apriori on the limited power set data file.

Proposition 5 *If $\mathcal{D} = \mathcal{LP}(n, \leq t)$ and $\sigma \leq \binom{n-t/2}{\leq t/2}$, then the time complexity of Apriori is $\Omega(2^t \binom{n}{t})$.*

Proof. $(t/2)$ -itemsets are frequent (since their support is easily seen to be $\binom{n-t/2}{\leq t/2}$), and likewise smaller itemsets are frequent.

Hence $\text{suppt}(\mathcal{F}^{(\sigma)}) \geq \sum_{i=0}^{t/2} \text{suppt}(\mathcal{I}_i)$. Since for each $i \leq t/2$, $\text{suppt}(\mathcal{I}_i) \geq \text{suppt}(\mathcal{I}_{t-i})$ (by Proposition 1(b)), then $\text{suppt}(\mathcal{F}^{(\sigma)}) \geq \text{suppt}(\mathcal{F}^{(1)})/2$. By Proposition 2 and Corollary 3, the execution time of Apriori on $\mathcal{LP}(n, \leq t)$ with $\sigma \leq \binom{n-t/2}{\leq t/2}$ is $\Omega(\text{suppt}(\mathcal{F}^{(\sigma)}))$, so is $\Omega(\sum_{r \in \mathcal{D}} 2^r)$, so is $\Omega(\binom{n}{t} 2^t)$. \square

4 FP-growth

In [7] Han, Pei and Yin introduced the new algorithm FP-growth. Two of its features are that the algorithm avoids the candidate generation of Apriori, and that its data structure (the FP-tree) tends to be much more compact than the data file itself.

In [7] it is suggested that items be ordered from most frequent to least. However, in the variant of FP-growth we treat here, items are in an arbitrary order, which is fixed throughout execution. Where b is an item, we use $I_{<b}$ to represent the set of items that precede b . Also, we will refer to suffixes of itemsets: let s, s' be subsets of I ; s' is a *suffix* of s if (letting the first element of s' be b) $s - I_{<b} = s'$, i.e., when writing elements of s in ascending order, the tail of the list equals s' .

The data file is represented by an *FP-tree*, which basically is a prefix tree or trie [8, 11]. In addition, there is a list of headers, one per frequent item, plus *item links* or *horizontal links* which form all nodes of the tree corresponding to an item b into a linked list having the header for b as head of the linked list.

In [7] it is established that the number of nodes (excluding the root) in a FP-tree is never more than the size of the data file. The notion of prefix-closed will enable us to say more. A database \mathcal{D} is said to be *prefix-closed* if for each record in \mathcal{D} , every prefix of it is also in \mathcal{D} .

Proposition 6 *If \mathcal{D} is prefix-closed and has no repeated records, then the number of nodes in the FP-tree for \mathcal{D} equals the number of records in \mathcal{D} .*

Proof. This is most easily seen if one constructs the FP-tree, processing from the shortest records in \mathcal{D} , progressively to the longest. The first record to process would then be \emptyset , i.e. the record containing no items; after processing it, the FP-tree has 1 node, the root. Each subsequent record would cause 1 additional node to be adjoined to the tree. \square

Although most databases are not prefix-closed, this proposition is still useful for them, as follows. If one can count the number of records that must be added to make the database prefix-closed, this proposition allows us to determine the number of nodes in the FP-tree (since the number of nodes in the FP-tree does not change while making the database prefix-closed).

This principle may also be stated:

Proposition 7 *For any data file \mathcal{D} , the number of nodes in the FP-tree for \mathcal{D} equals the number of distinct prefixes of records in \mathcal{D} .*

Let \mathcal{D} be a database and T be the FP-tree for \mathcal{D} . For an item b , the *conditional database* or *residual database* conditioned on b will be represented $R_b(\mathcal{D})$, and is defined as the set of truncated records $r \cap I_{<b}$ where r is a record of \mathcal{D} containing b : $R_b(\mathcal{D}) =$

$\{r \cap I_{<b} : r \in \mathcal{D}, b \in r\}$. The *conditional FP-tree* or *residual FP-tree* conditioned on b will be represented as $R_b(T)$, and is the FP-tree corresponding to $R_b(\mathcal{D})$.

FP-growth is a Divide and Conquer algorithm. The divide step is to create a series of residual FP-trees. There is nothing to do in the conquer step. As any divide and conquer algorithm, this algorithm solves “small” problem instances directly, and smashes “large” problem instances into several smaller ones (and makes a recursive call for each). In a little greater detail, a recursive call will be given a FP-tree T and have as its goal the generation of all frequent sets having some itemset α as suffix. If T is not a path (i.e., if this problem instance is “large”), then for each item b that is frequent in T , the residue $R_b(T)$ is formed and a recursive call is made, for the purpose of generating all frequent sets having $\{b\} \cup \alpha$ as suffix.

The main routine of this divide and conquer algorithm constructs the initial FP-tree to represent the database, and makes the initial call to the recursive subroutine. The pseudocode here is the same for FP-growth and its variant; they differ from each other in the procedure for constructing residual trees.

FP-growth(\mathcal{D})

1. /* Goal: given DB \mathcal{D} , to generate all frequent itemsets. */
2. Construct the FP-tree for database \mathcal{D} ; call it T .
3. $\alpha = \emptyset$.
4. recursive-FP-growth(T, α).
5. /* The goal of this call is to generate all f.s. having α as suffix, i.e. all f.s. */

recursive-FP-growth(T, α)

1. /* Goal: generate all f.s. having α as suffix. T is the FP-tree representing the set of records containing α as subset. */
2. If T is a single path then
3. Generate all f.s. of the form $\beta \cup \alpha$ where β is a subset of the item headers of T .
4. Else
5. Print “f.s. α has support” T .root.support.
6. For each item header in T :
7. Let b be the item of that header.
8. Construct the residue $R_b(T)$; call it T_b .
9. recursive-FP-growth($T_b, b \cup \alpha$). /* Goal: generate all f.s. having $b \cup \alpha$ as suffix. */

We give an overview of the process of constructing a residual FP-tree. In [7, 6, p.242] it is specified that the residual tree be constructed by first constructing the residual database, and then from the residual database constructing the residual tree. Instead, we will construct the residual tree directly.

Let T be a FP-tree and b an item appearing in T . The residue $R_b(T)$ may be constructed by starting at the item header for b and following the horizontal links so as to encounter all the b -nodes in T . At each b -node, travel upward to find and mark all its ancestors. An algorithm to do this and copy the marked nodes so as

to construct $R_b(T)$, is presented in Subsection 4.2, and is linear in the number of nodes marked. Sometimes (for example, when the minimum support threshold σ is 1), the residual tree is isomorphic to the set of marked nodes and the copying process is straightforward. Other times, consolidation of paths occurs, because some items in $I_{<b}$ become not frequent. The algorithm to construct the residual tree can be made linear in the number of nodes marked in the parent tree, even when consolidation occurs.

4.1 Time complexity of the variant of FP-growth

The conclusion of the next proposition does not hold for FP-growth as described in [7, 6]. The critical differences between the variant of FP-growth and the original FP-growth are that the variant constructs each residue directly from its parent tree (not via a residual database), and items are not sorted while constructing residual trees.

Proposition 8 *The time complexity of the variant of FP-growth on a database \mathcal{D} equals the sum of the following three quantities:*

- *size(\mathcal{D}), i.e., the sum of the number of items in all records.*
- *the sum of the number of nodes in unconsolidated residual FP-trees.*
- *the size of the output, i.e. the sum of the number of items in all frequent sets.*

Proof. From the pseudocode above, it is clear that the computational complexity of the execution time is the sum of the three quantities:

- time to read data file \mathcal{D} and construct the original FP-tree. The time to read \mathcal{D} is $\Theta(\text{size}(\mathcal{D}))$, and the original FP-tree can be constructed in the same time bound.
- time to construct all residual FP-trees. By choosing to construct each residual FP-tree directly from its parent FP-tree as described in Section 4.2, the time to construct it is linear in the number of nodes of the parent tree marked during the process of constructing it. If no consolidation occurs, this number equals the number of nodes in the FP-tree being constructed.
- time to write frequent sets.

Each of these three quantities is linear in the corresponding quantity in the statement of the proposition. \square

4.2 Algorithm to construct a residual FP-tree: detailed look

We are interested in constructing a residual FP-tree directly from the parent tree, without first constructing the residual database; specifically, given the FP-tree T , we are

to construct $R_b(T)$. Imports and exports (i.e., goals) of the residual tree construction routine follow. We are given:

- An item a_k and a set of items α , where $\alpha \subseteq \{a_{k+1}, a_{k+2}, \dots, a_n\}$ (and $n = |I|$).
- The residual FP-tree conditioned on α , $R_\alpha(T)$ where T is the original FP-tree.

The subroutine should construct $R_\beta(T)$ where $\beta = \{a_k\} \cup \alpha$. While the algorithm makes use of some fields in nodes of $R_\alpha(T)$, it does not destroy $R_\alpha(T)$. The algorithm may be outlined as follows.

- Step 1 Mark all nodes in $R_\alpha(T)$ that are ancestors of an a_k -node. To do this, start at the a_k item header, and follow the horizontal a_k -links. At each a_k node in $R_\alpha(T)$, mark it and travel upwards, marking all its ancestors, until the root is reached or a node is already marked.
- Step 2 Compute the support (in the residue) of all the marked nodes, and also the support of each item header.
- Step 3 Construct the item headers for $R_\beta(T)$: one header for each item in $I_{<a_k}$ except those whose support does not meet the minimum support threshold. (Also, each item header in $R_\alpha(T)$ that has a corresponding item header in $R_\beta(T)$ should point to it.)
- Step 4 Copy the marked nodes except the a_k -nodes (i.e., construct $R_\beta(T)$), consolidating paths as appropriate. (Consolidation can occur as a result of items that did not meet the minimum support threshold.)

The tree traversal methods used in Steps 2 and 4 are as follows.

- Step 2: use postorder traversal.
- Step 4: use preorder traversal. Before creating a child of a node, check if there already exists a child for that item; if so augment its support. (Evidently, two paths are being consolidated.)

For this to be efficient, each node needs rapid access to its children. In particular, to be linear in the number of marked nodes (including the a_k nodes) of the parent tree, we would like that the following operations be $O(1)$ (or amortized $O(1)$):

- Find the first child of a node x .
- Given a child y of x , find the next child (or determine that y is the last child).
- Find the a_k child of node x , or determine that x has no a_k child.

This can be accomplished by giving each node a pointer to its first child and a pointer to its next sibling; in addition, if a node has more than u children (for some threshold u), the node is given an array of $|I|$ pointers. The value of u would be somewhat large so that relatively few nodes would have such an array, but nonetheless this can be expensive in space.

5 Complexity of the variant of FP-growth on the Limited Power Set

In this section we establish the computational complexity of the variant of FP-growth on the limited power set data file. In this section, the minimum support threshold σ is 1. Results of this section then provide an upper bound on the time complexity with higher support thresholds.

The FP-tree for $\mathcal{LP}(n, \leq t)$ will be denoted by $T(\mathcal{LP}(n, \leq t))$. The number of nodes in $T(\mathcal{LP}(n, \leq t))$ is $\binom{n}{\leq t}$. This is because the database is prefix closed and has that many records.

We will say that two FP-trees are *isomorphic* if they are isomorphic as trees, and in addition their horizontal links are preserved. In short, they have the same structure. By contrast, the supports of nodes need not be preserved. If two FP-trees are isomorphic and their corresponding data files are prefix closed, then the data files have the property that, after deleting duplicate records, some bijection from the items of the first data file to the items of the second carries the first data file to the second.

We will want to know the size (and structure) of residual FP-trees that are constructed during execution of variant-FP-growth on $\mathcal{LP}(n, \leq t)$. It turns out that residual FP-trees fall in the same family as their parent tree:

Proposition 9 *The a_k residue of $T(\mathcal{LP}(n, \leq t))$ is isomorphic to $T(\mathcal{LP}(k-1, \leq t-1))$, and has $\binom{k-1}{\leq t-1}$ nodes.*

Proof. The residual data file consists of all records of $\mathcal{LP}(n, \leq t)$ that contain a_k . The residual FP-tree represents the intersection of each such record with $I_{<a_k}$

Each such record (intersected with $I_{<a_k}$) is a subset of $I_{<a_k}$ and has at most $t-1$ items (since each initially contained a_k). Hence each record is in $\mathcal{LP}(k-1, \leq t-1)$. Conversely, for each record r in $\mathcal{LP}(k-1, \leq t-1)$, $r \cup \{a_k\}$ is a record in $\mathcal{LP}(n, \leq t)$ that maps to r . (That several different records contain a_k and have the same intersection with $I_{<a_k}$ affects the support of nodes but not the structure of the residue.) \square

This result may be generalized to the following.

Proposition 10 *Where α is a set of items ($1 \leq |\alpha| \leq t$), having a_k as its first item, the α residue of $T(\mathcal{LP}(n, \leq t))$ is isomorphic to $T(\mathcal{LP}(k-1, \leq t-|\alpha|))$, and has $\binom{k-1}{\leq t-\alpha}$ nodes.*

Since FP-growth treats residual FP-trees that are paths differently than other residues, it is useful to know which residues are paths.

Proposition 11 $T(\mathcal{LP}(n, \leq t))$ is a path iff $t = 0$ or $n \leq 1$.

Proof. We note that if $t \geq 1$ and $n \geq 2$ then $\{a_1\}$ and $\{a_2\}$ are both records, so the FP-tree is not a path. \square

The next lemma is a technical result needed for Proposition 12. In it the upper limit on the sum is written as m instead of the more proper $\min(m, u)$. This is permissible because each term where j exceeds u contributes zero to the sum.

Lemma 2

$$\sum_{j=0}^m \binom{m}{j} \binom{h}{\leq u-j} = \binom{m+h}{\leq u}.$$

Proof. By Vandermonde convolution [10, p.104],

$$\sum_{j=0}^m \binom{m}{j} \binom{h}{u-j} = \binom{m+h}{u}.$$

(Alternately, this formula is valid because both sides count the number of ways of choosing u balls out of an urn of m red balls and h green balls.) Hence:

$$\sum_{j=0}^m \binom{m}{j} \binom{h}{\leq u-j} = \sum_{s=0}^{\infty} \sum_{j=0}^m \binom{m}{j} \binom{h}{u-j-s} \quad (3)$$

$$= \sum_{s=0}^{\infty} \binom{m+h}{u-s} \quad (4)$$

$$= \binom{m+h}{\leq u} \quad (5)$$

\square

Proposition 12 The total number of nodes in all residues generated by variant-FPgrowth on the data file $\mathcal{LP}(n, \leq t)$ is $n \binom{n-1}{\leq t-1} - \binom{n-2}{\leq t-2}$.

Example: Where $n = 5$ and $t = 3$, this means $5 * 11 - 4 = 51$ nodes.

Proof. The first step is, what residues are generated by FP-growth? FP-growth generates a residue $R_{\alpha}(T)$ (where T is the original FP-tree) for every subset α of I such that

- (1) $1 \leq |\alpha| \leq t$, and

- (2) α does not contain both a_1 and a_2 . (This exclusion is due to the fact that such a residue should be a child of $R_{\alpha-a_1}(T)$; but since $\alpha - a_1$ contains a_2 , the residue $R_{\alpha-a_1}(T)$ is a path; hence the algorithm does not generate any child from $R_{\alpha-a_1}(T)$.)

Let ρ be the sum of sizes of residues satisfying (1), and τ be the sum of sizes of residues satisfying (1) and failing to satisfy (2). Then $\rho - \tau$ is the quantity we wish to compute.

We first show that $\rho = n \binom{n-1}{\leq t-1}$. Where $|\alpha| = s$ and the first item in α is a_k , $R_\alpha(T)$ is isomorphic to $T(\mathcal{LP}(k-1, \leq t-s))$, so it has $\binom{k-1}{\leq t-s}$ nodes.

The number of residues that are isomorphic to $T(\mathcal{LP}(k-1, \leq t-s))$ equals the number of subsets α of I such that $|\alpha| = s$ and the first item in α is a_k ; there are $\binom{n-k}{s-1}$ such subsets. Hence $T(\mathcal{LP}(k-1, \leq t-s))$ appears $\binom{n-k}{s-1}$ times as a residue.

Hence ρ , the sum of the sizes of all residues, equals

$$\sum_{k=1}^n \sum_{s=1}^{\infty} \binom{n-k}{s-1} \binom{k-1}{\leq t-s}$$

We have written ∞ as the upper limit on the inner sum; this is equivalent to writing $\min(t, n-k+1)$ as the upper limit, since each term beyond $\min(t, n-k+1)$ contributes zero to the sum.

By Lemma 2, this quantity (ρ) equals $\sum_{k=1}^n \binom{n-1}{\leq t-1}$, so it equals $n \binom{n-1}{\leq t-1}$.

Now that we have found ρ , our next objective is τ . τ is the sum of sizes of all residues $R_\alpha(T)$ such that $|\alpha| \leq t$ and α does contain both a_1 and a_2 . Each such residual FP-tree has only a single node — the root. Hence the sum of sizes of such residues equals the number of such residues. The number of subsets α of I such that $\{a_1, a_2\} \subseteq \alpha$ and $|\alpha| \leq t$ is $\binom{n-2}{\leq t-2}$. Hence $\tau = \binom{n-2}{\leq t-2}$.

Finally, $\rho - \tau = n \binom{n-1}{\leq t-1} - \binom{n-2}{\leq t-2}$. \square

Proposition 13 *The computational complexity of the variant of FP-growth on $\mathcal{LP}(n, \leq t)$ is $\Theta(n \binom{n-1}{\leq t-1})$.*

Proof. The computational complexity of this algorithm $\mathcal{LP}(n, \leq t)$ is the sum of three quantities:

- The size of the input file. This quantity is $n \binom{n-1}{\leq t-1}$.
- The sum of the sizes of residual FP-trees. This quantity is $n \binom{n-1}{\leq t-1} - \binom{n-2}{\leq t-2}$, by Proposition 12, so it is $\Theta(n \binom{n-1}{\leq t-1})$.
- The sum of the sizes of the frequent sets. This equals the size of the input file.

6 Complexity of the variant of FP-growth on the Cartesian Product Data File

Let \mathcal{D} be the Cartesian product data file $\mathcal{CP}(c, t)$. We will perform the analysis supposing that the ordering on items is what we call the *standard ordering*: if items a and b are of attributes i and j respectively and $i < j$, then $a < b$. Also, throughout this section, the minimum support threshold is 1.

The number of nodes in the FP-tree for $\mathcal{CP}(c, t)$ equals the number of distinct prefixes occurring in the input file. The prefixes of size 1 are precisely the items of attribute 1: each record contains 1 item of attribute 1, and it will be the first item in c^{t-1} records. In general, the prefixes of size j are all j -tuples of items, where the i th coordinate is filled by an item of attribute i ($1 \leq i \leq j$). Hence the number of prefixes occurring in the input file is $\sum_{j=0}^t c^j = (c^{t+1} - 1)/(c - 1)$, and the number of nodes in the FP-tree for the file is the same.

Let T be the FP-tree for $\mathcal{CP}(c, t)$. For an itemset s we simplify the notation for the residue of s : $R_s(T)$ will be written as $R(s)$, and where s is a singleton $\{b\}$, we write $R(b)$. Let $1 \leq k \leq t$. For any item b of attribute k , the residual FP-tree $R(b)$ is isomorphic to the FP-tree for $\mathcal{CP}(c, k - 1)$. Inductively, then every residual FP-tree encountered in the execution of the variant of FP-growth on $\mathcal{CP}(c, t)$ (with the standard ordering) is isomorphic to $\mathcal{CP}(c, u)$ for some $u < t$.

Proposition 14 *The sum of the sizes of all residues (excluding the original FP-tree) is $\sum_{j=1}^t c(c+1)^{t-j}(c^j - 1)/(c - 1)$. This number is no more than $tc(c+1)^{t-1}$.*

Proof. Where $s \subseteq I$, a residue $R(s)$ is computed iff s contains at most one item of each attribute (hence s is contained in at least one record) and $s \neq \emptyset$. Let $s = \{b_1, b_2, \dots, b_j\}$ where $b_1 < b_2 < \dots < b_j$, and let b_1 be of attribute k . Then $R(s)$ is isomorphic to the FP-tree for $\mathcal{CP}(c, k - 1)$ so $R(s)$ has $(c^k - 1)/(c - 1)$ nodes. Also, the number of subsets s' of I such that $R(s')$ is isomorphic to $R(s)$ is $c(c+1)^{t-k}$, for every itemset having first item in attribute k is such a s' and s' can be constructed by choosing an arbitrary item in attribute k , and for each attribute after k either choosing an item of that attribute or choosing no item of that attribute ($c + 1$ alternatives). Hence the sum of nodes in all residues is $\sum_{k=1}^t c(c+1)^{t-k}(c^k - 1)/(c - 1)$.

To establish an upper bound on this quantity: let $\alpha_j = c(c+1)^{t-j}(c^j - 1)/(c - 1)$. Then the quantity we are interested in is $\sum_{j=1}^t \alpha_j$. It may be verified that the sequence $\alpha_1, \alpha_2, \dots, \alpha_t$ is a decreasing sequence. Hence $\sum_{j=1}^t \alpha_j \leq t\alpha_1 = tc(c+1)^{t-1}$. \square

Proposition 15 *The time complexity of the variant of FP-growth on $\mathcal{CP}(c, t)$ (with $\sigma = 1$, and the standard ordering on items) is $O(tc(c+1)^{t-1})$.*

Proof. The execution time of the algorithm on $\mathcal{CP}(c, t)$ with the standard ordering on items is the sum of 3 quantities:

- $\text{size}(\mathcal{D})$ (namely tc^t);
- the sum of the sizes of all FP-trees generated (namely the $(c^{t+1} - 1)/(c - 1)$ nodes in the original FP-tree plus at most $tc(c + 1)^{t-1}$ nodes in all residues);
- the sum of the number of items in all frequent sets (namely $tc(c + 1)^{t-1}$).

□

7 Apriori is unboundedly worse than variant-FPgrowth on the Cartesian Product

In this section we compare the computational complexity of Apriori and variant-FPgrowth on the Cartesian product file $\mathcal{CP}(c, t)$, when the minimum support threshold is no more than $c^{t/2}$. We will show that as t increases unboundedly, the ratio of the execution time of Apriori versus the execution time of variant-FPgrowth also grows unboundedly. More exactly, where f_A and f_F are respectively the execution time of Apriori and variant-FPgrowth on $\mathcal{CP}(c, t)$ (with the minimum support threshold at most $c^{t/2}$), we will show that $f_A \in \Omega((4/3)^t f_F/t)$.

Proof of this may be explained fairly briefly, modulo one step which is formulated as a lemma and proven immediately after.

Proof. By Proposition 4, $f_A \in \Omega(2^t c^t)$.

It has been shown in Proposition 14 that, when the minimum support threshold is 1, the number of nodes in all FP-trees generated by variant-FPgrowth (and hence the execution time of variant-FPgrowth) is $O(tc(c + 1)^{t-1})$. It is easy to see that, as the minimum support threshold increases, the number of nodes and execution time of variant-FPgrowth decrease. Hence $f_F \in O(tc(c + 1)^{t-1})$. Lastly, in Lemma 3 we show that $f_A/f_F \in \Omega((4/3)^t/t)$. □

Lemma 3

$$\frac{2^t c^t}{tc(c + 1)^{t-1}} \in \Omega((4/3)^t/t).$$

Proof.

$$\frac{2^t c^t}{tc(c + 1)^{t-1}} = \left(\frac{2c}{c + 1}\right)^{t-1} \frac{2}{t} \geq \left(\frac{4}{3}\right)^{t-1} \frac{2}{t} = \frac{3}{2} \left(\frac{4}{3}\right)^t / t \in \Omega((4/3)^t/t).$$

□

8 Apriori is unboundedly worse than variant-FPgrowth on the Limited Power Set

In this section we compare the computational complexity of Apriori and variant-FPgrowth on the limited power set data file $\mathcal{LP}(n, \leq t)$, when the minimum support threshold is no more than $\binom{n-t/2}{\leq t/2}$. Analysis will be limited to $t \leq .4n$.

By Proposition 13, the computational complexity of variant-FPgrowth on $\mathcal{LP}(n, \leq t)$ with $\sigma = 1$ is $\Theta(n \binom{n-1}{\leq t-1})$, which equals $\Theta(t \binom{n}{t})$ (since $t \leq .4n$). It is clear that as σ increases, the execution time of variant-FPgrowth does not rise; hence for all values of σ , variant-FPgrowth is an $O(t \binom{n}{t})$ algorithm.

Since the time complexity of variant-FPgrowth is $O(t \binom{n}{t})$ while Apriori is $\Omega(2^t \binom{n}{t})$ by Proposition 5, Apriori is unboundedly slower than variant-FPgrowth, as t becomes large.

9 Conclusion

We have shown that a variant of FP-growth is unboundedly faster than Apriori on two families of data files, as the data files become unboundedly large and the largest record simultaneously becomes large.

It has been observed that there are two basic families of frequent set generation algorithms [9]. One family, including Apriori, generates candidates and then determines their support by reading the data file. The other family (including FP-growth) computes intersections of records. The lower bound (Proposition 2) that we computed for Apriori will be valid for any algorithm in the first family that maintains the data file as a list of records. This suggests that algorithms in the first family might consider maintaining the data base as a trie, such as the FP-tree.

References

- [1] <http://fimi.cs.helsinki.fi/fimi03>.
- [2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, 1996.
- [3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. Technical Report RJ9839, IBM, 1994.
- [4] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th VLDB Conf.*, pages 487–499, 1994.

- [5] J. de Graff, W. Kusters, W. Pijls, and V. Popova. A theoretical and practical comparison of depth first and FP-growth implementations of Apriori. In *Proc. 14th Belgium-Netherlands Artif. Intell. Conf.*, pages 115–122, 2002.
- [6] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, San Francisco, 2001.
- [7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. 2000 ACM SIGMOD Intl. Conf. on Management of Data*. ACM, 2000.
- [8] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1973.
- [9] G. Liu, H. Lu, J. Yu, W. Wang, and X. Xiao. AFOPT: an efficient implementation of pattern growth approach. In *FIMI'03 Workshop (in conjunction with ICDM)*, 2003.
- [10] E. Packel. Permutations and combinations. In K. Rosen, editor, *Handbook of Discrete and Combinatorial Mathematics*, pages 96–104. CRC Press, 2000.
- [11] Robert Sedgewick. *Algorithms in C*. Addison-Wesley, Reading, MA, 1990.