A Space Optimization for FP-Growth

Eray Özkural and Cevdet Aykanat

September 9, 2004

Abstract

Frequency mining problem comprises the core of several data mining algorithms. Among frequent pattern discovery algorithms, FP-Growth employs a unique search strategy using compact structures resulting in a high performance algorithm that requires only two database passes. We introduce an enhanced version of this algorithm called FP-Growth-Tiny which can mine larger databases due to a space optimization eliminating the need for intermediate conditional pattern bases. We present the algorithms required for directly constructing a conditional FP-Tree in detail. Our experiments demonstrate that our implementation has a performance comparable to the original algorithm while reducing memory use more than twofold.

1 Introduction

Frequency mining is the discovery of all frequent patterns in a transaction or relational database. Frequent pattern discovery comprises the core of several data mining algorithms such as association rule mining and sequence mining [10], and it dominates the running time of these algorithms.

Frequency mining involves a transaction database $T = \{X | X \subseteq I\}$ which consists of a set of transactions each of which are drawn from a set I of items. The mining algorithm finds all patterns that occur with a frequency satisfying a given absolute support threshold ϵ. In practice, the number of items $|I|$ is in the order of magnitude of 10^3 and more. The number of transactions is much larger, at least 10^5. A pattern is $X \subseteq I$, a subset of I, and the set of all patterns is 2^I. The frequency function $f(T, i) = |\{x \in Y | Y \in T\}|$ computes the number of times a given item $i \in I$ occurs in the transaction set T; it is extended to sets of items $f(T, X) = |\{X \subseteq Y | Y \in T\}|$ to compute the frequency of a pattern.

Frequency mining is the discovery of all frequent patterns in a transaction set with a frequency of support threshold ϵ and more. The set of all frequent patterns is $\mathcal{F}(T, \epsilon) = \{X | X \in 2^I \wedge f(T, X) \geq \epsilon\}$. In the algorithms, the set of frequent items $F = \{x \in I \mid f(T, x) \geq \epsilon\}$ may require special consideration. A significant property of frequency mining known as downward closure states that if $X \in \mathcal{F}(T, \epsilon)$ then $\forall Y \subset X, Y \in \mathcal{F}(T, \epsilon)$ [2].

An inherent limitation of frequency mining is the amount of main memory available [8]. In this paper, we present a space optimization to FP-Growth algorithm and we demonstrate its impact on performance with experiments on synthetic and real-world datasets. In the next section, we give the background on the frequency mining problem. Section 3 explains our algorithm and Section 5 presents the experiments, following that we present our conclusions.

2 Background

2.1 Compact Structures

Compact data structures have been used for efficient storage and query/update of candidate item sets in frequency mining algorithms. algoSEAR [12], SPEAR [12], and DIC[6] use tries[1] while FP-GROWTH [10] uses FP-Tree which is an enhanced trie structure.

Using concise structures can reduce both running time and size requirements of an algorithm. Tries are well known structures that are widely used for storing strings and have decent query/update performance. The forementioned algorithms exploit this property of the data structure for better performance. Tries are also efficient in storage. A large number of strings can be stored in this dictionary type which would not otherwise fit into main memory. For frequency mining algorithms both properties are critical as our goal is to achieve efficient and scalable algorithms. In particular, the scalability of these structures are unmatched [10] as they allow an algorithm to track the frequency information of the candidate patterns for very large databases. The FP-Tree structure in FP-GROWTH allows the algorithm to maintain *all* frequency information in the main memory obtained from two database passes. Using the FP-Tree structure has also resulted in novel search strategies.

A notable work on compact structures is [14] in which a binary-trie based summary structure for representing transaction sets is proposed. The trie is further compressed using Patricia tries. Although significant savings in

[1] Also known as prefix trees.

storage and improvements in query time are reported, the effectiveness of the scheme in a frequency mining algorithm remains to be seen.

In this paper, we introduce an optimized version of FP-GROWTH. A closer analysis of it is in order.

2.2 FP-Growth Algorithm

The FP-GROWTH algorithm uses the frequent pattern tree (FP-Tree) structure. FP-Tree is an improved trie structure such that each itemset is stored as a string in the trie along with its frequency. At each node of the trie, *item*, *count* and *next* fields are stored. The *items* of the path from the root of the trie to a node constitute the item set stored at the node and the *count* is the frequency of this item set. The node link *next* is a pointer to the next node with the same *item* in the FP-Tree. Field *parent* holds a pointer to the parent node, *null* for root. Additionally, we maintain a header table which stores heads of node links accessing the linked list that spans all same items. FP-Tree stores only frequent items. At the root of the trie is a *null* item, and strings are inserted in the trie by sorting item sets in a unique[2] decreasing frequency order [10].

Table 1 shows a sample transaction set and frequent items in descending frequency order. Figure 1 illustrates the FP-Tree of sample transaction set in Table 1. As shown in [10], FP-Tree carries complete information required for frequency mining and in a compact manner; the height of the tree is bounded by maximal number of frequent items in a transaction. MAKE-FP-TREE (Algorithm 1) constructs an FP-Tree from a given transaction set T and support threshold ϵ as described.

Transaction	Ordered Frequent Items
$t_1 = \{f, a, c, d, g, i, m, p\}$	$\{f, c, a, m, p\}$
$t_2 = \{a, b, c, f, l, m, o\}$	$\{f, c, a, b, m\}$
$t_3 = \{b, f, h, j, o\}$	$\{f, b\}$
$t_4 = \{b, c, k, s, p\}$	$\{c, b, p\}$
$t_5 = \{a, f, c, e, l, p, m, n\}$	$\{f, c, a, m, p\}$

Table 1: A sample Transaction Set

In Algorithm 3 we describe FP-GROWTH which has innovative features such as:

[2]All strings must be inserted in the same order; the order of items with the same frequency must be the same.

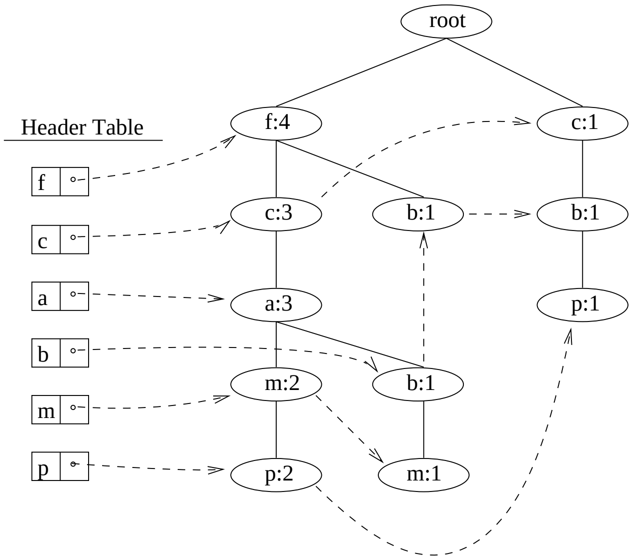

Figure 1: An FP-Tree Structure.

1. Novel search strategy
2. Effective use of a summary structure
3. Two database passes

FP-GROWTH turns the frequency k-length pattern mining problem into "a sequence of k-frequent 1-item set mining problems via a set of conditional pattern bases" [10]. It is claimed that with FP-GROWTH there is "no need to generate any combinations of candidate sets in the entire mining process"[3]. With an FP-Tree $Tree$ given as input the algorithm generates all frequent patterns. There are two points in the algorithm that should be explained: the single path case and conditional pattern bases. If an FP-Tree has only a single path, then an optimization is to consider all combinations of items in the path.[4] Otherwise, the algorithm constructs for each item a_i in the header table a *conditional pattern base* and an FP-Tree $Tree_\beta$ based on this structure for recursive frequency mining. Conditional pattern base is simply a compact representation of a derivative database in which only a_i and its prefix paths in the original $Tree$ occur. Consider path $< f : 4, c : 3, a : 3, m : 2, p : 2 >$ in $Tree$. For mining patterns including m in this path, we need to consider only the prefix path of m since the nodes after m will be mined elsewhere.[5]

[3] This is not an accurate picture as we will examine.
[4] Single path case is the basis of recursion in FP-GROWTH.
[5] In this case only p.

Considering the prefix path $< f : 4, c : 3, a : 3 >$ any pattern including m can have frequency equal to the frequency of m, therefore we may adjust the frequencies in the prefix path as $< f : 2, c : 2, a : 2 >$ which is called a *transformed prefix path* [10]. The set of transformed prefix paths of a_i forms a small database of patterns which co-occur with a_i and thus contains complete information required for mining patterns including a_i. Therefore, recursively mining conditional pattern bases for all a_i in $Tree$ is equivalent to mining $Tree$.[6] $Tree_\beta$ is simply the FP-Tree of the conditional pattern base.

FP-Growth is indeed remarkable with its unique divide and conquer approach. Nevertheless, it does generate candidates contrary to the title of "…Without Candidate Generation" [10]. The conditional pattern base is clearly a set of candidates among which only some of them turn out to be frequent. The main innovation however remains intact: FP-Growth takes advantage of a tailored data structure to solve the frequency mining problem with a divide-and-conquer method and with demonstrated efficiency and scalability. Besides, the conditional pattern base is guaranteed to be smaller than the original tree, which is a desirable property. An important distinction of this algorithm is that, when examined within our taxonomy of algorithms, it employs a unique search strategy. When the item sets tested are considered, it is seen that this algorithm is neither DFS nor BFS. The classification for FP-Growth in Figure 3 of [11] may be misleading. As Hipp later mentions in [11], "FP-Growth does not follow the nodes of the tree …, but directly descends to *some part* of the itemsets in the search space". In fact, the part is so well defined that it would be unjust to classify FP-Growth as conducting a DFS. It does not even start with item sets of small length and proceed to longer item sets. Rather, it considers a *set of patterns at the same time* by taking advantage of the data structure. This search strategy makes it hard to classify FP-Growth in the context of traditional uninformed search algorithms.

Algorithm 1 Make-FP-Tree(DB, ϵ)

Compute F and $f(x)$ where $x \in F$
Sort F in frequency decreasing order as L
Create root of an FP-Tree T with label "null"
for all transaction $t_i \in T$ **do**
    Sort frequent items in t_i according to L. Let sorted list be $[p|P]$ where p is the head of the list and P the rest.
    Insert-Trie($[p|P]$)
end for

[6] Which is equivalent to mining the complete DB.

Algorithm 2 INSERT-TRIE($[p|P], T$)

if T has a child N such that $item[N] = item[p]$ **then**
    $count[N] \leftarrow count[N] + 1$
else
    Create new node N with $count = 1$, *parent* linked to T and node-link linked to nodes with the same item via *next*
end if
if $P \neq \emptyset$ **then**
    INSERT-TRIE(P, N)
end if

Algorithm 3 FP-GROWTH($Tree, \alpha$)

if $Tree$ contains a single path P **then**
    for all combination β of the nodes in path P **do**
        generate pattern $\beta \cup \alpha$ with support *minimum support of nodes in* β
    end for
else
    for all a_i in header table of $Tree$ **do**
        generate pattern $\beta \leftarrow a_i \cup \alpha$ with $support = support[a_i]$
        construct β's conditional pattern base and then β's conditional FP-Tree $Tree_\beta$
        if $Tree_\beta \neq \emptyset$ **then**
            FP-GROWTH($Tree_\beta, \beta$)
        end if
    end for
end if

3 An Improved Version of FP-Growth

During experiments with large databases, we have observed that FP-GROWTH was costly in terms of memory use. Thus, we have experimented with improvements to the original algorithm. In this section, we propose FP-GROWTH-TINY (Algorithm 4) which is an enhancement of FP-GROWTH featuring a space optimization and minor improvements. An important optimization eliminates the need for intermediate conditional pattern bases. A minor improvement comes from not outputting all combinations of the single path in the basis of recursion. Instead, we output a representation of this task since subsequent algorithms can take advantage of a compact representation for generating association rules and so forth.[7] Another improvement is pruning the infrequent nodes of the single path.

In the following subsection, the space optimization is discussed.

Algorithm 4 FP-GROWTH-TINY($Tree, \alpha$)

if $Tree$ contains a single path P **then**
  prune infrequent nodes of P
  if $|P| > 0$ **then**
    output "all patterns in 2^P and α"
  end if
else
  for all a_i in header table of $Tree$ **do**
    $Tree_\beta \leftarrow$ CONS-CONDITIONAL-FP-TREE($Tree, a_i$)
    output pattern $\beta \leftarrow a_i \cup \alpha$ with $count = f(a_i)$ $\triangleright$ $f(x)$ of $Tree$
    if $Tree_\beta \neq \emptyset$ **then**
      FP-GROWTH($Tree_\beta, \beta$)
    end if
  end for
end if

3.1 Eliminating Conditional Pattern Base Construction

The conditional tree $Tree_\beta$ can be constructed directly from $Tree$ without an intermediate conditional pattern base. The conditional pattern base in FP-GROWTH can be implemented as a set of *patterns*. A pattern in FP-GROWTH consists of a set of *symbols* and an associated *count*. With a counting algorithm and retrieval/insertion of patterns directly into the FP-Tree structure, we can eliminate the need for such a pattern base. Algorithm 5 constructs a conditional FP-Tree from a given $Tree$ and a symbol s for which the transformed prefix paths are computed.

The improved procedure first counts the symbols in the conditional tree without generating an intermediate structure and constructs the set of frequent items. Then, each transformed prefix path is computed as patterns retrieved from $Tree$ and are inserted in $Tree_\beta$.

[7] For the FIMI workshop, we output all patterns as required. It can be argued that a meaningful mining of all frequent patterns must output them one by one.

Algorithm 5 CONS-CONDITIONAL-FP-TREE($Tree, s$)

```
table ← itemtable[Tree]
list ← table[symbol]
Tree′ ← MAKE-FP-TREE
▷ Count symbols without generating an intermediate structure
node ← list
while node ≠ null do
    COUNT-PREFIX-PATH(node, count[Tree])
    node ← next[node]
end while
for all sym ∈ range[count] do
    if count[sym] ≥ ϵ then
        F[Tree′] ← F[Tree′] ∪ sym
    end if
end for
▷ Insert conditional patterns to Tree_β
node ← list
while node ≠ null do
    pattern ← GET-PATTERN(node)
    INSERT-PATTERN(Tree′, pattern)
    node ← next[node]
end while
return Tree′
```

COUNT-PREFIX-PATH presented in Algorithm 6 scans the prefix paths of a given node. Since the pattern corresponding to the transformed prefix path has the count of the node, it simply adds the count to the count of all symbols in the prefix path. This step is required for construction of a conditional FP-Tree directly since an FP-Tree is based on the decreasing frequency order of F. This small algorithm allows us to compute the counts of the symbols in the conditional tree in an efficient way, and was the key observation in making the optimization possible.

Algorithm 6 COUNT-PREFIX-PATH($node, count$)

```
prefixcount ← count[node]
node ← parent[node]
while parent[node] ≠ null do
    count[symbol[node]] ← count[symbol[node]] + prefixcount
    node ← parent[node]
end while
```

Algorithm 7 GET-PATTERN($node$)

```
pattern ← MAKE-PATTERN
if parent[node] ≠ null then
    count[pattern] ← count[node]
    currnode ← parent[node]
    while parent[node] ≠ null do
        symbols[pattern] ← symbols[pattern] ∪ symbol[currnode]
        currnode ← parent[currnode]
    end while
else
    count[pattern] ← 0
end if
return pattern
```

Algorithm 7 retrieves a transformed prefix path for a given node excluding node itself and Algorithm 8 inserts a pattern into the FP-Tree. GET-PATTERN computes the transformed prefix path as described in [10]. INSERT-PATTERN prunes the items not present in the frequent item set F of $Tree$[8] and sorts the pattern in decreasing frequency order to maintain FP-Tree properties and adds the obtained string to the FP-Tree structure. The addition is similar to insertion of a single string, with the difference that

[8] Which does not have to be identical to the F of calling procedure.

insertion of a pattern is equivalent to insertion of the symbol string of the pattern *count*[*pattern*] times.

Algorithm 8 INSERT-PATTERN($Tree, pattern$)

1: $pattern \leftarrow pattern \cap F[Tree]$
2: Sort pattern in a predetermined frequency decreasing order
3: Add the pattern to the structure

The optimization in Algorithm 5 makes FP-GROWTH more efficient and scalable by avoiding additional iterations and cutting down storage requirements. An implementation that uses an intermediate conditional pattern base structure will scan the tree once, constructing a linked list with transformed prefix paths in it. Then, it will construct the frequent item set from the linked list, and in a second iteration insert all transformed prefix paths with a procedure similar to INSERT-PATTERN. Such an implementation would have to copy the transformed prefix paths twice, and iterate over all prefix paths three times, once in the tree, and twice in the conditional pattern list. In contrast, our optimized procedure does not execute any expensive copying operations and it needs to scan the pattern bases only twice in the tree. Besides efficiency, the elimination of extra storage requirement is significant because it allows FP-GROWTH to mine more complicated data sets with the same amount of memory.

An idea similar to our algorithm was independently explored in FP-GROWTH* by making use of information in 2-items [9]. In their implementations, Grahne and Zhu have used strategies based on 2-items to improve running time and memory usage, and they have reported favorable performance, which has also been demonstrated in the benchmarks of the FIMI '03 workshop [3].

4 Implementation Notes

We have made a straightforward implementation of FP-GROWTH-TINY and lincensed it under GNU GPL for public use. It has been written in C++, using GNU g++ compiler version 3.2.2.

For variable length arrays, we used `vector<T>` in standard library. For storing transactions, patterns and other structures representable as strings we have used efficient variable length arrays. We used `set<T>` to store item sets in certain places where it would be fast to do so, otherwise we have used sorted arrays to implement sets.

No low level memory or I/O optimizations were employed.

A shortcoming of the pattern growth approach is that it does not seem to be very memory efficient. We store many fields per node and the algorithm consumes a lot of memory in practice.

The algorithm has a detail which required a special code: sorting the frequent items in a transaction according to an order L, in line 2 of Algorithm 1 and line 2 of Algorithm 8. For preserving FP-Tree properties all transactions must be inserted in the very same order.[9] The transactions are sorted first in order of decreasing frequency and secondarily in order of items to achieve a unique frequency decreasing order. Using this procedure, we are not obliged to maintain an L.

5 Performance Study

In this section we report on our experiments demonstrating the performance of FP-Growth-Tiny. We have measured the performance of Algorithm 3 and Algorithm 4 on a 2.4Ghz Pentium 4 Linux system with 1GB memory and a common 7200 RPM IDE hard disk. Both algorithms were run on three synthetic and five real-world databases with varying support threshold. The implementation of the original FP-Growth algorithm is due to Bart Goethals.[10]

We describe the data sets used for our experiments in the next two subsections. Following that, we present our performance experiments and interpret the results briefly, comparing the performance of the improved algorithm with the original one.

5.1 Synthetic Data

We have used the association rule generator described in [2] for synthetic data. Synthetic databases in our evaluation have been selected from [16] and [15]. These databases have been derived from previous studies [2, 13, 1]. Table 2 explains the symbols we use for denoting the parameters of association rule generator tool. The experimental databases are depicted in Table 3. In all synthetic databases, $|I|$ is 1000, and $|\mathcal{F}_{max}|$ is 2000. The original algorithm could not process T20.I6.D1137 in memory therefore the number of transactions was decreased to 450K.

[9] For patterns also in our implementation.

[10] Goethals has made his implementation publicly available at http://www.cs.helsinki.fi/u/goethals/

$\lvert T\rvert$	Number of transactions in transaction set
$\lvert t\rvert_{avg}$	Average size of a transaction t_i
$\lvert f_m\rvert_{avg}$	Average length of maximal pattern f_m
I	Number of items in transaction set
$\lvert\mathcal{F}_{max}\rvert$	Number of maximal frequent patterns

Table 2: Dataset parameters

Name	$\lvert T\rvert$	$\lvert t\rvert_{avg}$	$\lvert f_m\rvert_{avg}$
T10.I6.1600K	1.6×10^6	10	6
T10.I4.1024K	1.024×10^6	10	4
T15.I4.367K	3.67×10^5	15	4
T20.I6.450K	4.5×10^5	20	6

Table 3: Synthetic data sets

5.2 Real-World Data

We have used five publicly available datasets in the FIMI repository. `accidents` is a traffic accident data [7]. `retail` is market basket data from an anonymous Belgian retail store [5]. The `bms-webview1`, `bms-webview2` and `bms-pos` datasets are from a benchmark study described in [4]. Some statistics of the datasets are presented in Table 4.

Name	$\lvert T\rvert$	$\lvert I\rvert$	$\lvert t\rvert_{avg}$
accidents	3.41×10^5	469	33.81
retail	8.82×10^4	16470	10.31
bms-pos	5.16×10^5	1657	6.53
bms-webview1	5.96×10^4	60978	2.51
bms-webview2	7.75×10^4	330286	4.62

Table 4: Real-world data sets

5.3 Memory Consumption and Running Time

The memory consumption and running time of FP-Growth-Tiny and FP-Growth are plotted for varying relative supports from 0.25% to 0.75% in Figure 2 for synthetic databases and Figure 3 for real-world databases except for accidents database which is a denser database that should be mined at

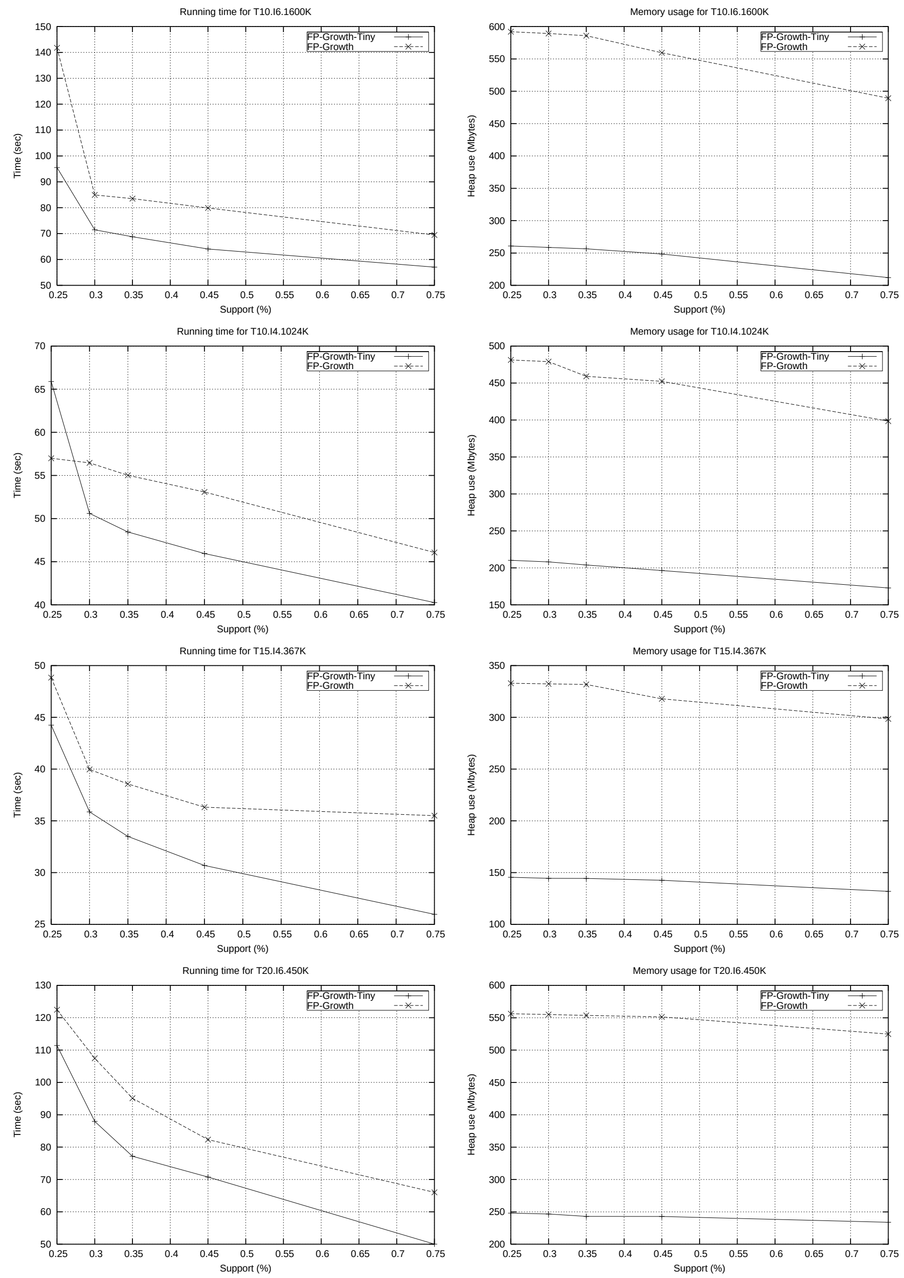

Figure 2: Time and space performance of FP-Growth-Tiny and FP-Growth on synthetic databases

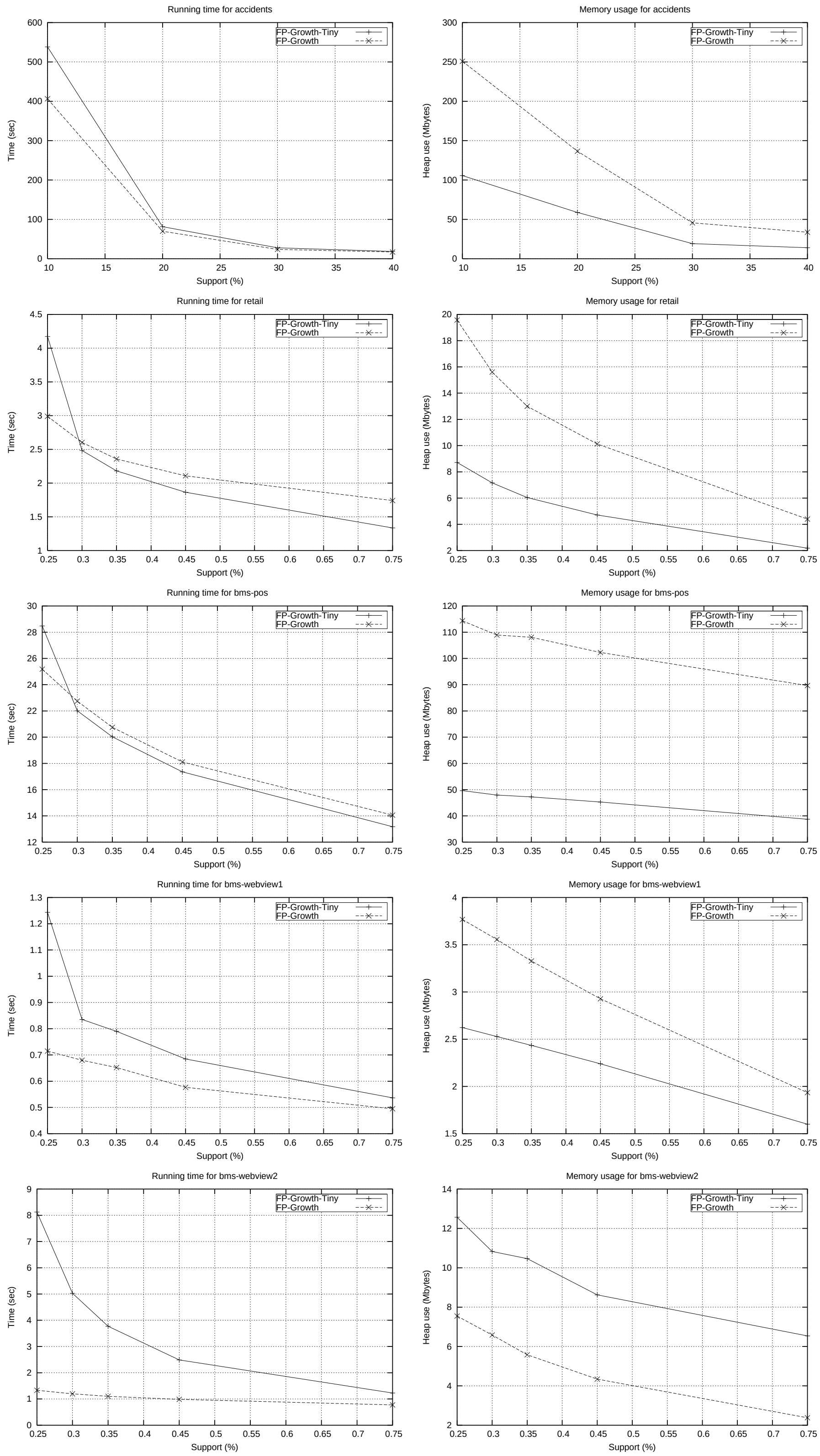

Figure 3: Time and space performance of FP-Growth-Tiny and FP-Growth on real-world databases

10% and more. The implementations are run inside a typical KDE desktop session. The running time is measured as the wall-clock time of the system call. The memory usage is measured using the GNU glibc tool `memusage`, considering only the maximum heap size since stack use is much smaller than heap size. Time and space performances are shown side by side in the figures to emphasize the relation between memory saving and decreased running time: the more memory is saved, the faster FP-Growth-Tiny becomes than FP-Growth. The plots for synthetic datasets are similar among themselves, while we observe more variation in real-world datasets.

Memory is saved in all databases, except in bms-webview2, which requires 2.74 times the memory used in FP-Growth; this has an adverse effect on running time as discussed below. In others, we observe that memory usage reduces down to 41.5% in accidents database with 4% support, which is 2.4 times smaller than FP-Growth.

Due to the optimization, our implementation can process larger databases than the vanilla version. For most problem instances, the memory consumption has been reduced more than twofold compared to the original algorithm. An obvious advantage of our approach is that with the same amount of memory, we can process more complicated databases.[11] The experiments overall show that the conditional pattern base construction which we have eliminated has a significant space cost during the recursive construction of conditional FP-Trees.

The running time behaviors of two algorithms are quite similar on the average. Our algorithm tends to perform better and is faster in higher support thresholds, while in lower thresholds the performance gap becomes closer. FP-Growth-Tiny runs faster except in bms-webview1, bms-webview2 and lower thresholds of T10.I4.1024K. In bms-webview1 database, FP-Growth-Tiny runs 10-27% slower; in bms-webview2 database we observe that FP-Growth-Tiny has slowed down by a factor of 5.56 for 0.25% support threshold, and slowdown is observed also for other support thresholds (down to 50%). In T10.I4.1024K we see 12% slowdown for 0.25% support and 2% slowdown for 0.3% support. In other problem instances FP-Growth-Tiny, runs faster, up to 28.5% for retail dataset at 0.75% support.

We had expected that improving space utilization would remarkably decrease the running time. However, we have not observed as large an improvement as we would have liked in running time. On the other hand, our trials show signiciant improvement in memory use contrasted to vanilla FP-Growth, allowing us to mine more complicated/larger datasets with the

[11]Note that FP-Growth uses a compressed representation of frequency information, whose size may be thought of as related to complexity of the dataset.

same amount of memory.

The adverse situation with bms-webview2 and bms-webview1 shows that the performance study must be extended to determine whether the undesirable behavior recurs at a large scale, since these are both sparse data sets coming from the same source. At any rate, a closer inspection of FP-Growth-Tiny seems necessary. We anticipate that the benchmark studies at the FIMI workshop will illustrate its performance more precisely.

6 Conclusions

We have presented our version of FP-Growth which sports multiple improvements in Section 3. An optimization over the original algorithm eliminates a large intermediate structure required in the recursive step of the published FP-Growth algorithm in addition to two other minor improvements.

In Section 5, we have reported the results of our performance experiments on synthetic and real-world databases. The performance of the optimized algorithm has been compared with a publicly available FP-Growth implementation. We have observed more than twofold improvement in memory utilization over the vanilla algorithm. In the best case, memory size has become 2.4 times smaller, while in the worst case memory saving was not possible in a small real-world database. Typically, our implementation makes better use of memory, enabling it to mine larger and more complicated databases that cannot be processed by the original algorithm. The running time behavior of both algorithms are quite similar on the average; FP-Growth-Tiny runs up to 28.5% percent faster, however it may run slower in a minority of instances.

References

[1] R. Agrawal and J. C. Shafer. Parallel mining of association rules. *IEEE Trans. On Knowledge And Data Engineering*, 8:962–969, 1996.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[3] Mohammed Javeed Zaki Bart Goethals. Fimi '03: Workshop on frequent itemset mining implementations. In *Proceedings of the ICDM 2003*

Workshop on Frequent Itemset Mining Implementations, Melbourne, Florida, USA, 2003.

[4] Zijian Zheng Blue. Real world performance of association rule algorithms. In *KDD 2001.*

[5] Tom Brijs, Gilbert Swinnen, Koen Vanhoof, and Geert Wets. Using association rules for product assortment decisions: A case study. In *Knowledge Discovery and Data Mining*, pages 254–260, 1999.

[6] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 255–264. ACM Press, 05 1997.

[7] Karolien Geurts, Geert Wets, Tom Brijs, and Koen Vanhoof. Profiling high frequency accident locations using association rules. In *Proceedings of the* 82^{nd} *Annual Transportation Research Board*, page 18pp, Washington DC (USA), January 2003.

[8] B. Goethals. Memory issues in frequent itemset mining. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*, Nicosia, Cyprus, March 2004.

[9] Gösta Grahne and Jianfei Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *Proceedings of the First IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03).*

[10] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In Weidong Chen, Jeffrey Naughton, and Philip A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, May 2000.

[11] Jochen Hipp, Ulrich Güntzer, and Gholamreza Nakhaeizadeh. Algorithms for association rule mining – a general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, July 2000.

[12] Andreas Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. Technical Report CS-TR-3515, College Park, MD, 1995.

[13] Ashoka Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *The VLDB Journal*, pages 432–444, 1995.

[14] Dow-Yung Yang, Akshay Johar, Ananth Grama, and Wojciech Szpankowski. Summary structures for frequency queries on large transaction sets. In *Data Compression Conference*, pages 420–429, 2000.

[15] Mohammed Javeed Zaki, Srinivasan Parthasarathy, and Wei Li. A localized algorithm for parallel association mining. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 321–330, 1997.

[16] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. Parallel algorithms for discovery of association rules. *Data Mining and Knowledge Discovery*, 1(4):343–373, 1997.