Mining Frequent Sequential Patterns with First-Occurrence Forests

Erich A. Peterson
Department of Applied Science
University of Arkansas at Little Rock
2801 S. University Ave.
Little Rock, AR 72204

Peiyi Tang
Department of Computer Science
University of Arkansas at Little Rock
2801 S. University Ave.
Little Rock, AR 72204

ABSTRACT

In this paper, a new pattern-growth algorithm is presented to mine frequent sequential patterns using First-Occurrence Forests (FOF). This algorithm uses a simple list of pointers to the first-occurrences of a symbol in the aggregate tree [1], as the basic data structure for database representation, and does not rebuild aggregate trees for projection databases. The experimental evaluation shows that our new FOF mining algorithm outperforms the PLWAP-tree mining algorithm [2] and the FLWAP-tree mining algorithm [3], both in the mining time and the amount of memory used.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data mining*

General Terms

Algorithms, Performance, Design

Keywords

Projection Database, Frequent Patterns

1. INTRODUCTION

As datasets from scientific and commercial applications become larger and larger, the information and knowledge "hidden" within them becomes increasingly valuable. One area of data mining, which has attracted a significant amount of research, is that of mining frequent sequential patterns from sequences. The reason for this is the large number of domains and applications that can benefit from it. Mining web log, protein, and DNA sequences are just a few of these applications.

Due to the exponential search space and large datasets to mine, data mining algorithms and methods must be efficient in both time and space (i.e. to complete the mining process in the shortest time), using the least amount of memory space. The most notable algorithms of frequent sequential pattern mining[1] include the apriori-based algorithms such as GSP [4] and pattern-growth algorithms such as the WAP-tree mining algorithm [5], the PLWAP-tree mining algorithm [2] and the FLWAP-tree mining algorithm [3]. The pattern growth algorithms find frequent patterns in a depth-first search of the search space. They grow frequent patterns by mining increasingly smaller projection databases, and thus, are faster than apriori-based algorithms. For pattern-growth algorithms, database representation has a strong impact on both the mining time and the memory used.

The WAP-tree (*Web Access Patterns Tree*) mining algorithm [5] uses the *aggregate* tree [1] to represent a sequence database, where all nodes of the same label are linked when the tree is built. It grows the suffixes of frequent patterns and has to rebuild a new aggregate tree for each projection database. Thus, it uses a lot of memory. The PLWAP-tree (*Pre-Order Linked WAP-tree*) mining algorithm [2] links the nodes of the same symbol in a pre-order traversal. It grows the prefixes of frequent patterns and uses position code to determine the boundary of projection databases. Thus, it does not need to rebuild aggregate trees for projection databases. The FLWAP-tree (*First-Occurrence Linked WAP-tree*) mining algorithm [3] links only the first-occurrences of each symbol in the tree and outperforms the PLWAP-tree in mining time tremendously. However, it has to rebuild the aggregate trees for projection databases and uses a lot of memory.

While each of the above mentioned algorithms were able to reduce either the mining time or memory utilization, they were not able to reduce both. Moreover, all of them are based on the concept of the linked tree: some nodes of the same symbols are linked somehow to facilitate mining. They all strayed away from the simplicity found in the original aggregate tree for database representation. In this paper, we get rid of the concept of the linked tree and propose a new pattern-growth algorithm based on the concept of *First-Occurrence Forests* (FOF). We use forests of first-occurrence subtrees as our basic data structure for the database representation, and employ a simple list of tree node pointers to first-occurrences in the aggregate tree. There is no need to rebuild aggregate trees for projection databases. The first-occurrences of a symbol are found using a depth-first search of the aggregate tree on-the-fly. The experimentation evalu-

ACM-SE '08, March 28–29, 2008, Auburn, AL, USA.

[1]Frequent sequential pattern mining in general mines frequent sequences of itemsets [4] from sequence itemset databases. In this paper (and others [5, 2, 3]), sequential patterns are refered to as sequences of items or symbols.

ation shows that our algorithm is faster and uses less memory than both the PLWAP-tree mining algorithm [2] and the FLWAP-mining algorithm [3].

The rest of this paper is organized as follows: Section 2 provides the preliminaries of conditional searching for all pattern-growth algorithms. Section 3 describes our new First-Occurrence Forests (FOF) mining algorithm. Section 4 presents the results of experimental evaluations. Finally, Section 5 concludes the paper.

2. CONDITIONAL SEARCHING PRELIMINARIES

Let Σ be the set of symbols. A non-empty sequence s is a sequence of finite number of symbols from Σ, $s = s_1 \cdots s_m$, such that $s_i \in \Sigma$ for all $1 \leq i \leq m < \infty$ and s_i and s_j are not necessarily different for $i \neq j$. The *length* of sequence $s = s_1 \cdots s_m$ is m. The empty sequence denoted as ϵ is a special sequence of length 0. A sequence database D is a multi-set of finite sequences including the possible empty sequence. A pattern is also a non-empty sequence. A non-empty finite sequence is a subsequence of another sequence if it is embedded in that sequence. In particular, sequence $s' = s'_1 \cdots s'_n$ is a *subsequence* of sequence $s = s_1 \cdots s_m$, denoted as $s' \subseteq s$, if and only if $n \leq m$ and there exist $i_1, \cdots, i_n$ such that $1 \leq i_1 < \cdots < i_n \leq m$ and $s'_j = s_{i_j}$ for all $1 \leq j \leq n$. The empty sequence ϵ is a subsequence of any sequence. A sequence s in D is said to *support* pattern p if p is a subsequence of s. The *support* of pattern p in D, denoted as $Sup_D(p)$, is the number of sequences in D that support p. Given a threshold ξ in interval $(0, 1]$, a pattern p is *frequent* with respect to ξ and D if $Sup_D(p) \geq \xi|D|$, where $|D|$ is the number of sequences in D. $\xi|D|$ is called the *absolute threshold* and denoted as η. The frequent sequential pattern mining problem is to find all the frequent sequential patterns in D with respect to ξ. Since the empty pattern ϵ is a subsequence of every sequence in D, it is always frequent because its support is $|D|$ which is always greater than or equal to $\xi|D| = \eta$.

The pattern-growth algorithms for mining frequent sequential patterns mentioned previously, are all based on the principle of conditional searching. Fundamentally, pattern-growth algorithms find frequent sequential patterns by traversing the search space tree in the depth-first order, using smaller projection databases. The new approach presented in this paper is also a pattern-growth algorithm. For completeness, we present the basic principles of conditional searching and the abstract pattern-growth algorithm in this section. A detailed formal framework to reason about pattern-growth algorithms can be found in [3].

Given a symbol a from Σ and a sequence s that supports the single-symbol sequence a (i.e. $a \subseteq s$), the *a-prefix* of s is the prefix of s from the first symbol (the leftmost symbol) to the *first* occurrence of a inclusive. For example, the a-prefix of sequence $bcabad$ is bca rather than $bcaba$, because the first occurrence of a is the a underlined. The *a-projection* of s is what is left after the a-prefix is removed. In the example above, the a-projection of sequence $bcabad$ is bad, because the a-prefix is bca. Note that if a occurs only once and is the last symbol of s, the a-prefix is s itself and the a-projection is the empty sequence ϵ. For example, the d-projection of $bcabad$ is ϵ.

Given database D and a symbol $a \in \Sigma$, the *a-projection database* of D, denoted as D_a, is the multi-set of a-projections of the sequences in D that support a. That is,

$$D_a = \{a\text{-projection of } s \mid a \subseteq s \wedge s \in D\}$$

For example, the a-projection database of $D = \{cbaca, bcbaca\}$ is $D_a = \{ca, ca\}$.

According to the theory and formal framework of conditional searching [3], the set of all non-empty frequent sequential patterns in database D with respect to threshold ξ can be found by calling *Pattern-Grow*(ϵ, D, η) with $\eta = \xi|D|$ as shown in Figure 1 (reproduced from [3]).

```
main(database D, int η) {
   F ← Pattern-Grow(ϵ, D, η);
   return F;
}

function Pattern-Grow(pattern q, database D, int η) {
   F ← ∅;
   for each symbol a in Σ do
1:    if (Sup_D(a) ≥ η) then
2:       F ← F ∪ {q · a};
3:       Let D_a be the a-projection database of D;
4:       F ← F ∪ Pattern-Grow(q · a, D_a, η);
      endif
   endfor
   return F;
}
```

Figure 1: Abstract Pattern-Growth Mining Algorithm

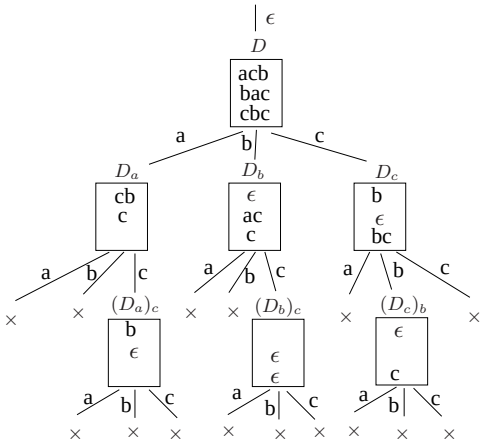
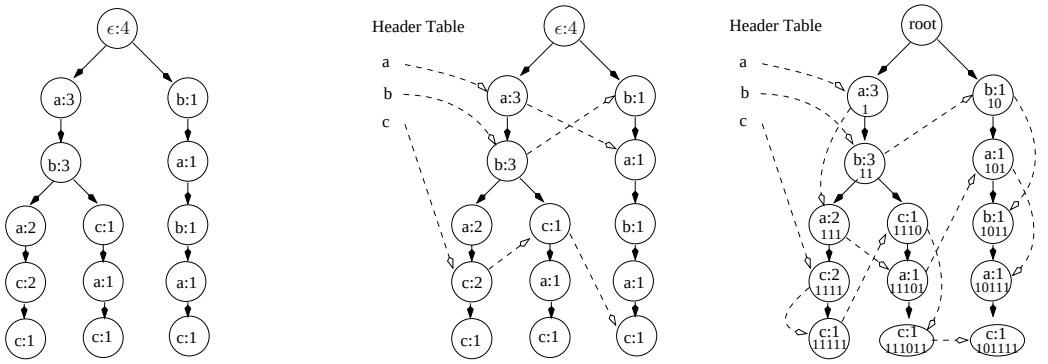


Figure 2: Example of Pattern-Growth Mining

Figure 2 (reproduced from [3]) illustrates how the pattern-growth mining algorithm in Figure 1 mines the frequent patterns from database $D = \{acb, bac, cbc\}$ with respect to $\xi = 2/3$ ($\eta = 2$). Each node represents a database with its contents in the box. Parentheses in the names of the projection databases are used to show the projection databases of projection databases. For example, $(D_a)_c$ is the c-projection database of D_a. Node $\times$ represents a database whose size is below $\eta = 2$, where the pattern growth stops. Each edge represents the symbol considered for possible extension of the pattern. A path from the root node to a node other than $\times$ is a frequent pattern. The frequent patterns mined in the depth-first order of their discovery are: $\epsilon, a, ac, b, bc, c, cb$.



(a) Aggregate Tree (b) FLWAP-Tree (c) PLWAP-Tree

Figure 3: Various WAP-tree Implementations

3. MINING FREQUENT PATTERNS WITH FIRST-OCCURRENCE FORESTS

The algorithm in Figure 1 is an abstract algorithm, in that it does not specify (1) how a database is represented, (2) how to find the projection database of a symbol and (3) how to find the support of a symbol in the database (i.e. $Sup_D(a)$). All these questions are important because they have a profound impact on the performance of the mining algorithm.

In this section, we present an implementation of the abstract algorithm, using forests of aggregate trees as the basic data structure to represent databases. In the next section, it will be shown that the our implementation and algorithm is faster and uses less memory than both the PLWAP algorithm [2] using PLWAP-trees and the FLWAP algorithm [3] using FLWAP-trees. We start with the basic data structure for database representation.

3.1 Forest of First-Occurrence Subtrees

The aggregate tree [1] is a compact data structure used to represent a sequence database. Each sequence in the sequence database, is represented by a path from the root node to the node of the last symbol of the sequence. The path from the root node to any other node, is a common prefix of possibly multiple sequences in the database. Each node has a label for the symbol and a count for the number of sequences that share this node in their paths. The same idea was used in the WAP-tree [5], and in the FP-tree for frequent itemset mining in [6].

The aggregate tree can be constructed by entering each sequence in D into the initial tree, which has only one root node pointed by a pointer *current-node*. For a sequence $s_1 \cdots s_n$ the construction enters each symbol s_i $(i = 1, \cdots, n)$ using the following algorithm. If *current-node* has a child with label s_i, increase the count of that child and make *current-node* point to that child. Otherwise, create a new child node with label s_i and count 1, and make *current-node* point to this new child node. The details of the algorithm can be found in [1, 5]. Figure 3(a) shows the aggregate tree or base WAP-tree of the example database $D = \{abac, abcac, babac, abacc\}$ from [2, 5].

In this paper, the aggregate tree is extended to make the root node represent the empty symbol ϵ. The count of the root node is the total number of sequences in the database. It is obvious that the count of any node in the aggregate tree is equal to or greater than the sum of the counts of its children.

The difference between the count of a node and the sum of the counts of its children, is the number of sequences in D that end with the symbol of the node. For example, node c:2 has only one child c:1 in the aggregate tree in Figure 3(a). Therefore, there is 2-1=1 sequence in D that starts with a and ends with this c. That is the sequence $abac$.

Integral to all the pattern-growth mining algorithms abstracted in Figure 1 are the tasks of (1) counting the support of a symbol in the database (i.e. $Sup_D(a)$ for symbol a) and (2) identifying the representation of the corresponding projection database (i.e. D_a for symbol a). If the database is represented by an aggregate tree, these tasks amount to finding the so-called first-occurrences of the symbol in the tree. In particular, a node in an aggregate tree is a *first-occurrence* of symbol a if it is labeled with a and none of its ancestors has the same label. For example, the aggregate tree in Figure 3(a) has two first-occurrences of a: a:3 in the left subtree and a:1 in the right subtree. All the other nodes labeled with a are not first-occurrences, because they are descendants of the first-occurrences of a.

The count of a first-occurrence of a node with label a is the number of sequences in D that share the common a-prefix (defined in Section 2), represented by the path from the root node to this first-occurrence. For example, b:3 is a first-occurrence of b in the aggregate tree in Figure 3(a). The count 3 indicates that there are three sequences in D that share the common b-prefix ab represented by the path from the root node to b:3. These sequences are $a\underline{b}ac$, $a\underline{b}cac$ and $a\underline{b}acc$. Therefore, the sum of the counts of all the first-occurrences of a symbol $a \in \Sigma$ is the number of sequences in D that contain at least one occurrence of a (i.e. the support of a in D, $Sup_D(a)$).

Also, the subtrees rooted at the children of a first-occurrence of symbol a, represent all the non-empty a-projections of the sequences sharing this common a-prefix. For example, the two subtrees rooted at the children of the first occurrence b:3 in Figure 3(a) represent the three non-empty b-projections: ac, cac and acc. Therefore, the a-projection database of D, D_a, can be represented by the subtrees rooted at the children of all the first-occurrences of a, plus possible empty sequences. Since the purpose of having a-projection database D_a is to grow frequent patterns by using symbols from Σ

(see the abstract algorithm in Figure 1), it is sufficient to use the subtrees rooted at the children of the first-occurrences of a to represent projection database D_a, ignoring possible empty sequences.

Since first-occurrences play a central role in finding the support of a symbol and its projection database, all pattern-growth mining algorithms based on aggregate trees try to find them efficiently. The PLWAP-tree mining algorithm [2] links *all* occurrences of each symbol in a pre-order traversal, turning the aggregate tree into a PLWAP-tree (Pre-order Linked WAP tree). Figure 3(c) shows the PLWAP-tree from the running example database. To find the first-occurrences of a symbol, the algorithm goes through all the occurrences, following the links and using position codes, to determines the first-occurrences. The PLWAP-tree algorithm uses the original PLWAP-tree for the entire mining and does not rebuild agregate trees for projection databases. Thus, it uses less memory. However, in order to find the first-occurrences of a symbol, it has to go through all its occurrences, including those who are not part of the projection database. The FLWAP-tree algorithm [3], on the other hand, links only the first occurrences of each symbol. Figure 3(b) shows the FLWAP-tree (First-Occurrence Linked WAP tree) of the same example database. However, the FLWAP-tree algorithm rebuilds every projection database, and thus, uses a lot of memory. Both the PLWAP-tree [2] and FLWAP-tree algorithms, [3] are based on the concept of the linked tree: nodes of the same symbols are linked together somehow.

In this paper, the concept of the linked tree is thrown out and is replaced with a forest of *first-occurrence subtrees* as the basic data structure for projection database representation. Given a symbol a, each subtree rooted at a first-occurrence of a is called a *first-occurrence subtree* of a. The forest of *first-occurrence subtrees* of a symbol is simply a list of pointers to the first-occurrences of a in the aggregate tree. Figure 4 shows the forest of first-occurrence subtrees of a using the example database D in Figure 3(a). The root

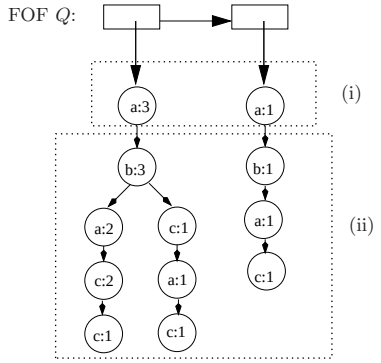


Figure 4: FOF for D_a

nodes of the first-occurrence subtrees form part (i). The sum of the counts of these root nodes provides the support of a in database D (i.e. $Sup_D(a)$). The subtrees rooted at the children of the nodes in part (i) form part (ii) which represent the projection database D_a. Note that all the nodes in both part (i) and part (ii) already exist in the original aggregate tree of the database to be mined. The memory cost of the forest of first-occurrences subtrees is simply the list of pointers. The data type of the forest of first-occurrences subtrees is called FOF. Figure 5 shows the general FOF data structure used in the new mining algorithm.

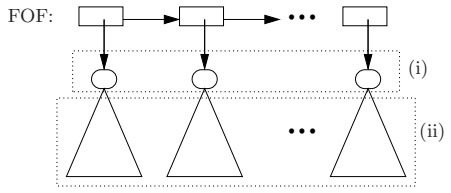


Figure 5: FOF Abstract Data Structure

The initial aggregate tree in Figure 3(a) is regarded as a first-occurrence subtree of empty symbol ϵ. Its root node is the only first-occurrence of ϵ in D and its count is the support of ϵ (i.e. $Sup_D(\epsilon)$), which is equal to the number of sequences in D. Thus, the FOF data structure containing the original aggregate tree as the only first-occurrence subtree is shown in Figure 6.

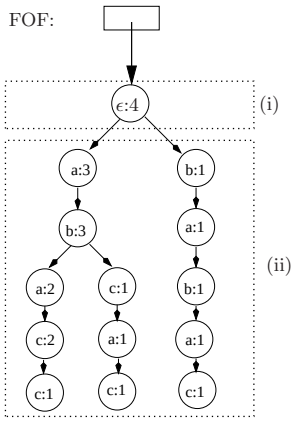


Figure 6: Initial FOF for D

3.2 FOF Mining Algorithm

Figure 8 presents the FOF mining algorithm. The data structure of the nodes in the aggregate tree is depicted in Figure 7. Each node contains: a label representing a symbol of the node, and an integer for the count. The tree is linked by the pointers in each node to the first child node (firstChild) and the next sibling (nextSibling). In the *main*

```
struct node {
   symbol label;
   int count;
   node * firstChild;
   node * nextSibling;
};
```

Figure 7: Node Data Structure

function in Figure 8, the aggregate tree was built in line 3 and incorporated into the initial FOF structure *initialFOF* in line 4. This initial FOF structure is like the one in Figure 6. It is passed on to the pattern growth mining function

FOFMine() in line 5 to find all non-empty frequent sequential patterns.

The global variable Q of type FOF in Figure 8, is used to pass a new FOF structure every time the *FOFMine()* function is called.

Function *FOFMine()* is our pattern-growth mining function implementing the abstract mining algorithm in Figure 1. Argument T of type FOF is the FOF structure whose part (ii), the subtrees rooted at the children of the nodes of part (i), represent the current database to be mined. Part (i) of T represents the first occurrences of the previous symbol in the previous call and it is not used in the current call. Lines 3-6 build the FOF structure of Q for the projection database of the current database. The **if** statement at line 7, corresponds to line 1 of the abstract algorithm in Figure 1. It sums up the counts of the root nodes of all first-occurrence subtrees (i.e. the nodes in part (i)) in Q, to see if it is above η. If it is above η, Q is passed on to the next recursive call of *FOFMine()*, to mine the projection database stored in part (ii) of Q.

The FOF structure for the projection database is established in Q, by repeatedly calling function *FindFirstOccurrences()* (line 5) for the first child of the root nodes of each first-occurrence subtrees (part (i)) in T. It effectively searches for first-occurrences of the symbol in part (ii) of T. Line 2 of function *FindFirstOccurrences()* appends the first-occurrence subtree to Q once it is found. If not found, the function continues the depth-first search (shown in lines 3-10).

As an example, the algorithm is traced using the running example database with $\eta = 3$. The initial FOF structure for the original database is shown in Figure 6. This is the FOF structure passed as T to function *FOFMine()*. There is one node N in part (i) of this FOF structure. Starting from the first child of this N node (node a:3), the function builds the FOF structure of Q with two first-occurrence subtrees and this is shown in Figure 4. This Q will be passed on to the next level call of *FOFMine()*, because the sum of the counts of the root nodes of the subtrees (node a:3 and a:1) in Q is $3+1=4$, which is greater than η. Therefore, the algorithm has found frequent pattern a. Then, this Q will be passed to the recursive call of *FOFMine()* (line 10), which will build the new Q for the first occurrence subtrees of symbol a as shown in Figure 9. This mining process continues until all frequent sequential patterns have been discovered.

```
FOF Q;

function main () {
1:  FOF initialFOF ← newFOF();
2:  node root;
3:  Construct initial aggregate tree starting from root;
4:  initialFOF.append(root);
5:  F ← FOFMine(ϵ, initialFOF, η);
}

function FOFMine(pattern q, FOF T, int η) {
1:  F ← ∅;
2:  for each symbol a in Σ do
3:     Q ← newFOF();
4:     for each root node N of the subtrees in T do
5:        FindFirstOccurrences(a, N.firstChild);
6:     endfor
7:     if (the sum of the count of the root nodes
8:           of all the subtrees in Q) ≥ η) then
9:        F ← F ∪ {q · a};
10:       F ← F ∪ FOFMine(q · a, Q, η);
11:    endif
12:    delete Q;
13: endfor
14: return F;
}

procedure FindFirstOccurrences(symbol a, node N) {
1:  if (N.label = a) then
2:     Q.append(N);
3:  else
4:     if (N.firstChild ≠ NULL) then
5:        FindFirstOccurrences(a, N.firstChild);
6:     endif
7:  endif
8:  if (N.nextSibling ≠ NULL) then
9:     FindFirstOccurrences(a, N.nextSibling);
10: endif
}
```

Figure 8: FOF Algorithm

4. EXPERIMENTAL RESULTS

Now that the FOF algorithm has been fully disseminated, one remaining discussion is left of importance: the evaluation and comparison of the FOF algorithm to other well-known frequent sequential pattern mining algorithms. As mentioned before, the two notable types of frequent sequential pattern mining algorithms are the apriori-based and pattern-growth. Because our algorithm is a member of the latter type, we chose to compare our algorithm with the other well-known pattern-growth algorithms. In our evaluation, the mining time and the memory usage of the FOF algorithm were compared to those of the PLWAP-tree and FLWAP-tree algorithms. Source code for the PLWAP-tree was obtained from the authors' web site, and the FLWAP-tree's source code was obtained through its authors. Each algorithm was run on the same random datasets generated by the IBM data generator, and were executed on an E6600 Intel Core 2 Duo 2.40GHz system with 4 GB of RAM running SuSe Linux version 9.3. Several parameters can be supplied in order to produce datasets of varying characteristics using the generator and they include:

- N is the number of unique symbols in Σ (i.e. $|\Sigma|$).
- D is the number of sequences to be generated.
- C is the average length of the sequences generated.

In order to accurately measure memory utilization of the various algorithms, instrumentation code was added to each. Let $T = \{t_i, \cdots, t_n\}$ be the set of n data types for which the peak memory usage of the dynamic heap variables needs to be measured. Let the size of data type t_i $(1 \leq i \leq n)$ be s_i. For each data type $t_i \in T$ $(1 \leq i \leq n)$, an integer c_i is injected into the code for the count of the number of variables of type t_i that have been allocated. Also, let P represent the total peak heap memory usage. Whenever in the mining program a variable of type t_i is allocated, the code $c_i \leftarrow c_i + 1; P \leftarrow max(\sum_{i=1}^{n} c_i \cdot s_i, P)$ is inserted. Conversely, whenever a variable of type t_i is deallocated, the code $c_i \leftarrow c_i - 1$ is inserted to decrement c_i. Thus, at the end of execution, variable P will hold the total peak memory usage.

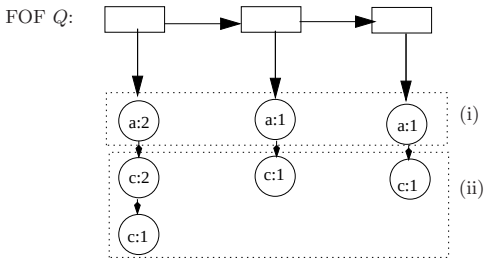
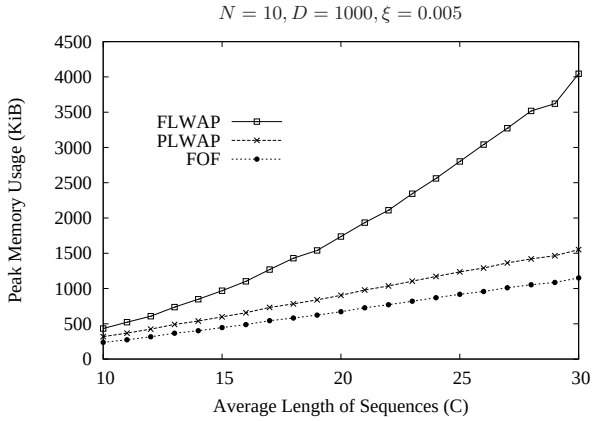
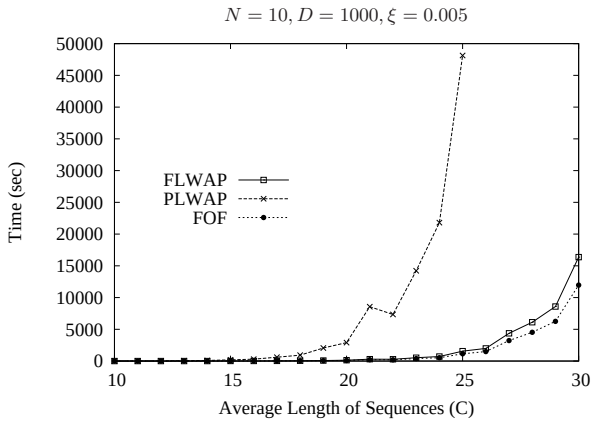


Figure 9: FOF for $(D_a)_a$

Figure 10(a) presents a memory usage comparison among the various algorithms, where $N = 10, D = 1000, \xi = 0.005$, and C is varied from 10 to 30. The FOF algorithm is shown to outperform the PLWAP-tree algorithm and more significantly the FLWAP-tree algorithm. The large difference in peak memory usage between the FLWAP-tree and the FOF algorithms, is mostly due to the creation of new projection databases in the FLWAP-tree algorithm, which consumes more memory space. Also, the FOF algorithm outperforms even the PLWAP-tree algorithm, even though it does not create additional projection databases. This is most likely due to the increased size of tree nodes in the PLWAP-tree (i.e. for added link pointers and position codes).



(a) Memory Usage Comparison



(b) Execution Time Comparison

Figure 10: Performance Comparison

The execution times of the differing algorithms was also tested. To accurately measure execution time, code was added to the original implementations, which calculates total execution time. Figure 10(b) shows an execution time comparison, where $N = 10, D = 1000, \xi = 0.005$, and C is varied from 10 to 30. Again, the FOF algorithm is shown to outperform the FLWAP-tree and the PLWAP-tree in terms of execution time. This time however, the FOF algorithm outperformed the PLWAP-tree algorithm in a more significant manner. This is due to the fact, that much time is spent traversing the PLWAP-tree in search of first-occurrences.

5. CONCLUSIONS

We have presented a pattern-growth frequent sequential pattern mining algorithm using a forest of first-occurrence subtrees as the basic data structure for databases (including projection databases). Our algorithm uses a simple list of pointers to the first-occurrence subtrees and a simple algorithm to find first-occurrences. The performance evaluation reveals that this simple implementation of pattern-growth mining, outperform both the PLWAP-tree and FLWAP-tree mining algorithms, in both mining time and memory usage. In particular, our FOF mining algorithm is significantly faster than the PLWAP-tree mining algorithm and uses significantly less memory than the FLWAP-tree mining algorithm.

6. REFERENCES

[1] Myra Spiliopoulou and Lukas C. Faulstich. WUM: A tool for web utilization analysis. In *Proceedings of EDBT Workshop Web DB'98*. Springer Verlag, LNCS 1590, 1998.

[2] Christie I. Ezeife and Yi Lu. Mining web log sequential patterns with position coded pre-order linked wap-tree. *International Journal of Data Mining and Knowledge Discovery*, 10:5–38, 2005.

[3] Peiyi Tang, Markus P. Turkia, and Kyle A. Gallivan. Mining web access patterns with first-occurrence linked WAP-trees. In *Proceedings of the 16th International Conference on Software Engineering and Data Engineering (SEDE'07)*, pages 247–252, Las Vegas, USA, July 2007.

[4] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the International Conference on Extending Database Technology*, pages 3–17, 1996.

[5] Jian Pei, Jiawei Han, Behzad Mortazavi-asl, and Hua Zhu. Mining access patterns efficiently from web logs. In *Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'00)*, pages 396–407. Lecture Notes in Computer Science, Vol. 1805, 2000.

[6] Jiawei Han, Jian Pei, and Yiwen Yin. Mining frequent patterns without candidate generation. In *Proceedings of the ACM SIGMOD International on Management of Data*, pages 1–12. ACM Press, 2000.