Efficiently Computing Frequent Tree-Like Topology Patterns in a Web Environment

Xuemin Lin* Chengfei Liu† Yanchun Zhang‡ Xiaofang Zhou§

Abstract

Data mining recently emerges from a number of real-life applications, such as banking, supermarketing, and interneting. In this paper, we investigate a novel data mining problem derived from the applications of WWW. We will present an efficient algorithm for solving the problem. Our initial implementation suggests that the algorithm is very efficient and scalable in practice.

Keywords: Data Mining Tools, Topological Access Patterns, WWW.

1 Introduction

The internet and World Wide Web (WWW) have grown exponentially in recent years, so that huge amount of information is now available on the internet for user access. Managing, organizing, and efficiently accessing internet resources, therefore, become critical. While a number of internet search engines have already been prototyped and are now available for use, recent success in modern database technologies such as data mining [1, 4], data warehousing and OLAP [10], and web queries [6], brings a great impact on further tuning the performance of a web search engineering. In this paper, we will investigate a novel data mining problem in computing user frequent web access patterns.

Current research in mining user patterns in a WWW environment follows two directions: 1) computing *rule-based* patterns [5, 7], 2) computing *topology-based* patterns [4]. In a rule-based approach, we view the input, user access records, as a table in a relational

*School of Computer Science & Engineering, University of New South Wales, Sydney, NSW 2052, Australia. E-mail: lxue@cse.unsw.edu.au

†School of Computing Science, University of Technology Sydney, Sydney NSW 2009, Australia. E-mail: liu@socs.uts.edu.au

‡Department of Mathematics & Computing, University of Southern Queensland, Toowoomba, QLD 4350, Australia. E-mail: yan@usq.edu.au.

§Department of Computer Science & Electrical Engineering, University of Queensland, Brisbane QLD 4072, Australia. E-mail: zxf@csee.uq.edu.au

database, and output the discovered *association* rules by the algorithms in [1, 9]. In a topological oriented approach, we view a web environment, the linked documents, as a *directed* graph; and view a user access to the web environment as a *walk* [2] in the directed graph along arcs. Then, frequent user walk patterns, restricted to a specific topology, are to be computed. In this paper, we are interested in the problem of mining "frequent tree-like traversal"; and the problem is abbreviated to MFTT.

A tree-like traversal is analogous to a *depth-first search tree* [2] in a directed graph and will be formally defined in the next section. The problem MFTT has a number of real-life applications in a web environment, where an understanding of user access patterns will not only help improve the system design [10] (e.g., provide efficient access between highly corelated objects and better authoring design for pages) but also be able to lead to better marketing policies [4] (e.g. putting advertisements in proper places and better customer/user classification and behavior analysis). For instance, in maintaining hierarchical (graph structured) views [10] for web databases, it is usually impossible to create the complete views due to physical resource limitation. Therefore, an understanding of user access patterns will provide us the useful information about which views may have to be materialized.

To the best of our knowledge, [4] is the first paper to investigate efficient computation of user access patterns restricted to a specific graph topology - *path*. The problem MFPP is more general than the problem in [4]. Further, the algorithm in [4] is not applicable to MFPP. In this paper, we will present a new and efficient algorithm to solve MFTT; and the algorithm is based on an efficient iteration paradigm. Our intitial implementation shows that the algorithm is efficient and scalable in practice.

The rest of the paper will be organized as follows. In next section, we formally define the problem. The third section presents our algorithm as well as the performance discussions. This is followed by the conclusions.

2 Preliminary

A user can access the internet as many times as he (she) likes; and each time can access as many web sites as he (she) wants. Detailed information about web access from the users in a local system may be recorded in a system log. These records will enable the local system to learn the web access patterns from its users and then develop a good system for its users [8]. This is one of immediate applications of our results in this paper.

An access to web from a user may be retrieved from the system log in form of $< UID, (s_1, d_1), (s_2, d_2), ...(s_n, d_n) >$, where domain UID gives the user ID, (s_i, d_i) indicates that the user travels from document s_i to document d_i, and $d_i = s_{i+1}$. By recording the starting point of an access as null, we should be able to identify different accesses from the same user, and different accesses from different users.

To impose a partial ordering, in this paper we interpret a user access as a tree traversal in a web environment, which is similar to the depth-first search tree in a directed graph. Therefore, we convert a user access into a *traversal* tree below. Iteratively, add a new vertex d_i into the tree T together with the link (s_i, d_i) if d_i does not occur in T. For example, with respect to the access

$$< (null, A), (A, B), (B, C), (C, B), (B, A), (A, D), (D, E) >,$$

the corresponding traversal tree is illustrated in Figure 1(a). A traversal tree is a *rooted tree* [2]; that is, a tree with notion of parents and children. In this paper, we are interested only in rooted trees; for terminology simplicity, a rooted tree is abbreviated to "tree" in this paper.

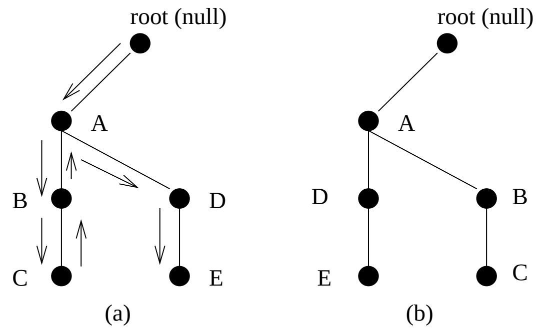

Figure 1: A Traversal Tree

We assume that a database (table) D for user access records has been obtained such that each row (tuple) is a traversal tree with the attribute TID for identification. The problem MFTT can be described as follows. Given a minimum support $s\%$, find the set T of all trees such that:

- for each tree $t \in T$, there are at least $s\%$ of tuples in D that contain t as a subtree, and
- no two trees in T having an inclusion relationship.

Here we call a tree with k vertices (nodes) *k-tree*; and call a tree t *frequent* if there are at least $s\%$ of tuples in D that contain t as a subtree. A frequent tree t is *maximal* if there is no other frequent tree that contains t as a subtree. Note that MFTT is to compute all the maximal frequent trees.

3 An Algorithm for MFTT

A naive way to solve MFTT is to 1) firstly find out all vertices of the trees in the database; and 2) secondly enumerate all subtrees generated from these vertices; and 3) thirdly for each subtree t, count how many tuples in D contain t as a subtree. It should be clear that for n vertices, the number of possible subtrees generated from the n vertices is larger than $n!$. Therefore, this naive approach is too expensive to be applicable in practice, as n may be very large.

Inspired by the results in [1, 4, 9], in this paper we present an efficient algorithm based on an efficient iteration for solving MFTT. Note that a k-tree t is qualified as a frequent tree if all $(k-1)$-subtrees of t are frequent. This gives the foundation to our iteration based algorithm. In our algorithm, instead of enumerating all k-trees for counting purpose, we generate only the k-trees such that their $(k-1)$-subtrees are frequent. In practice, this will dramatically reduce the number of candidates to be considered.

To present our algorithm, we need the following notation.

- L_k denotes the set of frequent k-trees.
- C_k denotes a superset of L_k.

Our algorithm consists of the three steps below:

Algorithm CMFTT

Input: A database D where each tuple represents a tree; and s.
Output: All the maximum frequent subtrees.

Step 1: Read in D to compute $L1$.

Step 2: **for** $k = 2$, $k++$ **do**
{ compute C_k from L_{k-1};
count C_k against D to get L_k; }
Repeat till $L_k = \emptyset$.

Step 3: Compute the maximal trees from $\cup_k L_k$. $\square$

It should be clear that a tree can be represented and stored by a *linked list* data type. Note that a tree may be represented in a number of different ways according to different vertex orderings in the tree. For instance, Figure 1(a) and Figure 1(b) give the two different representations of the same tree. The nun-uniqueness of representation will lead to a slow detection of the equality of two trees. To resolve this, in this paper we represent a tree in a unique way to *preserve* the ordering of vertices. That is, in a tree, the children vertices of each vertex are stored in the linked list according to the increasing ordering of the children. For instance, regarding the tree in Figure 1, the representation in Figure 1(a) is always adopted if $A < B < C < D < E$. In the rest of paper, we assume that the trees mentioned are stored and represented in this way. Clearly, each subtree of a tree also preserves the vertex ordering.

Below, we present the algorithm CMFTT step by step.

3.1 Computing L_1

The first step of the algorithm CMFTT is conceptually trivial. Basically, we read in D and count the occurrence of each vertex v. Then output L_1 that contains frequent 1-tress (i.e. vertices).

Note that before computing L_1, we may have no information about how many vertices that are present in the database. To speed up the counting computation, we may use a hashing technique to handle it; more details can be found in the next subsection when we present our algorithm for Step 2.

3.2 Efficient Execution of Step 2

Step 2 iteratively computes C_k and L_k from L_{k-1}, where C_k is the set of candidates to be included in L_k. The following theorem is immediate according to the definition of L_k.

Theorem 1 *For a k-tree $t \in L_k$, each $(k-1)$-subtree of t must be in L_{k-1}.*

3.2.1 Computing C_k

Theorem 1 is the base of our approach to compute C_k. We use a join-like operation to compute C_k from L_{k-1}. Clearly, each tree with at least two vertices must have at least two vertices with degree 1. Given a tree, the *breadth-first* search method [2] from the root to the bottom and from the *left* to the *right* will generate the unique vertex sequence; for instance, regarding Figure 1(a), the vertex sequence $(root, A, B, D, C, E)$ will be generated. We call such a vertex sequence *adjoint* vertex sequence of the tree. For a rooted tree t, we denote the first vertex with degree 1 in the adjoint vertex sequence by $t.f$, and denote the last vertex in the adjoint vertex sequence by $t.l$. Note that $t.l$ must be a leave, while $t.f$ may be a root in some cases.

Based on Theorem 1, each k-trees in C_k is composed of a pair of $(k-1)$-trees t_1 and t_2 in L_{k-1}, such that $t_1 - \{t_1.f\} = t_2 - \{t_2.l\}$. Figure 2 and Figure 3 respectively illustrate two representative examples, where $t_1.f$ is either a leave or the root. Note that Figure 3 also shows the necessity of using the first node with degree 1 instead of using the first leave.

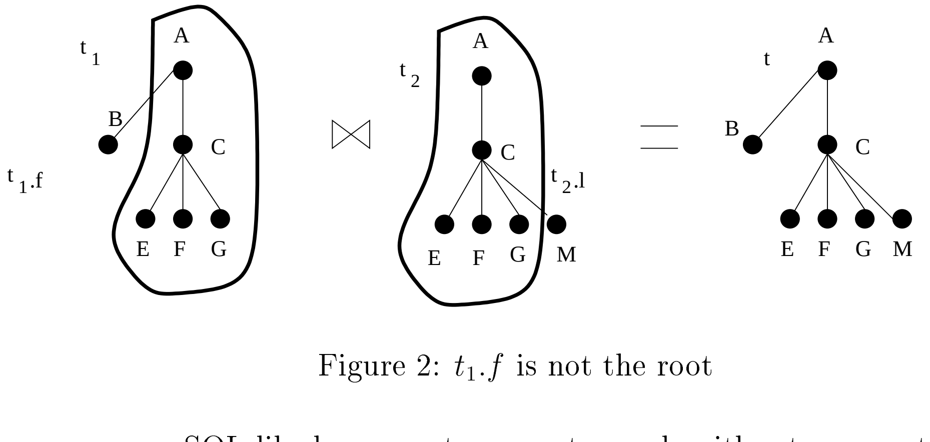

Figure 2: $t_1.f$ is not the root

Below, we use a SQL-like language to present our algorithm to compute C_k.

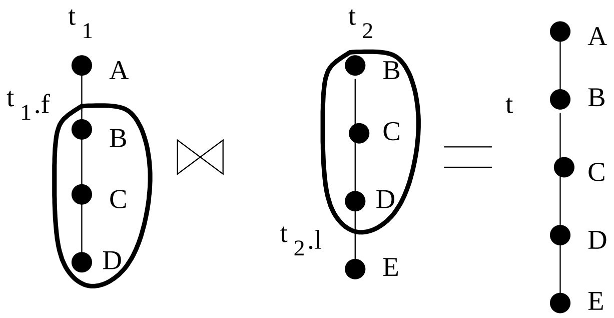

Figure 3: $t_1.f$ is the root

Insert into C_k
select $x.f$, $x - \{x.f\}$, $y.l$
from L_{k-1} x, L_{k-1} y
where $x - \{x.f\} = y - \{x.l\}$.

To make the best use of Theorem 1, we can further prune some unnecessary elements included in C_k:

for each element $t \in C_k$ **do**
  for each $(k-1)$-subtrees t' of t **do**
    if $(t' \notin L_{k-1})$ **then**
      delete t from C_k. $\square$

Note that to efficiently implement the computation of C_k, we use a similar paradigm to sort-merge join to combine the above two processes together. We first sort L_{k-1} according to the lexicographical ordering of the corresponding adjoint vertex sequences and then implement the above join operation and pruning process.

The special case for computing C_k is when $k = 2$; in this case, $C_2 = L_1 \times L_1$.

3.2.2 Counting C_k

Once C_k is obtained, the algorithm CMFTT carries out a counting process to record the number of occurrences of each element (k-tree) in C_k. To speed up the counting process, it is desirable to employ a hashing technique as with the computation of *association rules* [1]. The specification of adjoint vertex sequence makes it feasible. Below, we present our counting process.

The elements (k-trees) in C_k are stored in a *hash-tree* that consists of k layers. A node of the hash-tree either contains a list of k-trees in C_k (a *leaf* node) or a hash table (an *internal* node). In an internal node, each bucket of the hash table points to another node. The root of the hash tree is defined to be depth 1. An internal node at depth d points to nodes at depth $d+1$.

Note that the hash-tree for C_k is built based on adjoint vertex sequence with k vertices

$(x_1, x_2, ..., x_k)$. The hash-tree is created in a way such that the root stores a hash table that hashes the first vertex of each adjoint vertex sequence in C_k; while the children of the root store the corresponding hash tables that hash the second vertex in the adjoint vertex sequence, and so fourth. Therefore, the hash tree for C_k has k-layers. For example, assume that the database contains 6 vertices $\{A, B, C, D, E, F\}$, where their orders follow the corresponding alphabetical ordering and the order of A is 1. There are four 2-trees $\{B \to C, C \to B, F \to E, E \to F\}$ in C_2. Suppose that the hash function is $order(x)\ mode$ 3, where $x \in \{A, B, C, D, E, F\}$. The hash-tree is illustrated in Figure 4.

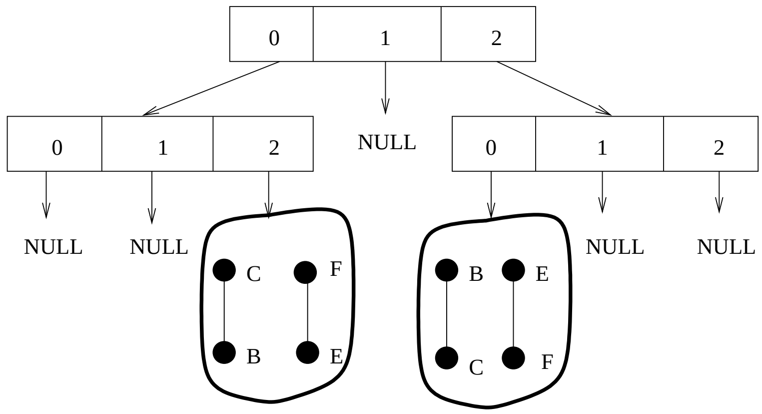

Figure 4: A hashing example

After the hash tree for C_k is built, each element (k-tree) in C_k will be counted for its occurrence in the database D. This can be done by the following steps.

S1: Read in each tuple t in D.

S2: Decompose t into a number of k-subtrees t_i.

S3: Probe each subtree t_i to the hash tree to see if t_i is an element of C_k. If t_i is in C_k, then the count of t_i is increased by 1.

In our implementation of S2 and S3, we start with decomposing the adjoint vertex sequence $(x_1, x_2, ..., x_n)$ of t into the set of sub-sequences with length k. Then, probe each subsequence $(x_{i_1}, x_{i_2}, ..., x_{i_k})$ into the hash tree in such a way that x_{i_1} is hashed against the hash table in the root. If it is hashed into an empty bucket, then the subsequence is omitted; otherwise x_{i_2} is hashed by the corresponding node of the hash tree, and so on. Eventually, a vertex subsequence with length k is either hashed into a leave node or omitted on the half way. If a vertex subsequence $(x_{i_1}, x_{i_2}, ..., x_{i_k})$ is probed into the leaf node, then it will be recovered as a subtree, by referencing the tree t, to be compared with the k-trees stored in the leaf node to count C_k. Note that a vertex subsequence does not always correspond to an adjoint vertex sequence of a subtree of t; and if it cannot be recovered as a subtree then it should be omitted. For instance, the vertex subsequence (A, C, E) does not correspond to a subtree of the tree in Figure 1(a).

To further speed up the counting process, in our implementation we combine the decomposition of a tree to subtrees with the process of probing the hash tree together to prune a subsequence as earlier as possible. For instance, with respect to the adjoint vertex sequence $(x_1, x_2, ...x_n)$, if we find that (x_2, x_3) either falls to an empty hash bucket or there is no edge connecting them in the tree, then all the subsequences with (x_2, x_3) as the prefix should not be considered.

Moreover, as the database may be too large to reside in the main memory all the time during the computation, we read in database each time when we count C_k. After C_k has been counted, clearly we need to scan C_k only once to get L_k. In summary, the computation of L_k by counting C_k is described as follows.

Counting C_k

Input: The database D and C_k.
Output: L_k

Create a hash tree H on C_k;
Read in D;
for each $t \in D$ **do**
{ count C_k against t; }
Scan C_k to obtain L_k. □

3.3 Computing Maximal Frequent Trees

This step is based on the following theorem that can be immediately verified.

Theorem 2 *If an element t in L_i is contained by an element in L_j for $j > i$, then t must be contained by an element in L_{i+1}.*

This theorem implies that we can implement the 3rd step of the algorithm CMFTT iteratively between each pair of L_{i-1} and L_i from $i = 2$ to $i = N$ where $N = \max\{k : L_k \text{ is the set of frequent } k\text{-trees}\}$. That is, for $2 \leq i \leq N$, remove elements t from L_{i-1} if t is contained as a subtree of an element in L_i.

3.4 Discussion

It should be clear that the algorithm CMFTT runs in polynomial time in each iteration. The dominant computation costs in each iteration are the reading costs of D from hard disk to the main memory, and the counting costs. Our preliminary implementation suggest that this algorithm runs fast in practice. We initially implement the algorithm on a pentum-pro workstation with main memory size 128 MB running the Windows NT server 4.0 System and using Javaview. For a database where there are 200,000 tuples on average, each tuple is a tree with 20 vertices on average, and totally there are more than 1000 vertices, the algorithm takes just a few minutes to complete. This performance is very comparable to the computation of association rules [1], although MFTT appears more complicated.

4 Conclusions

In this paper, we specified a novel data mining problem MFTT from real application in WWW. We also provide a framework for efficiently solving MFTT. Currently, we continue our implementation by applying various new techniques in mining association rules [3, 9] into CMFTT. As a possible future study, we will investigate the computation of more complicated topological patterns, as well as in a dynamic database.

References

[1] R. Agrawal and R. Srikant, Fast Algorithm for Mining Association Rules in Large Database, *Proceedings of the 20th International Conference on Very Large Data Bases*, 478-499, 1994.

[2] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*, Macmillan, 1977.

[3] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, Dynamic Itemstet Counting and Implication Rules for Market Basket Data, *Proceedings of 1997 ACM SIGMOD*, 225-264, 1997.

[4] M.-S. Chen, J. S. Park, and P. S. Yu, Efficient Data Mining for Path Traversal Patterns, *IEEE Transactions on Knowledge and Data Engineering*, 10(2), 209-221, 1998.

[5] D. W. Cheung, B. Kao, and J. Lee, Discovering User Access Patterns on the World-Wide Web, *1st Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD-97)*, 1997.

[6] H. Garcia-Molina, W. Labio, and R. Yerneni, Capability-Sensitive Query Processing on Internet Sources, *ICDE'99*, 50-59, 1999.

[7] B. Mobasher, N. Jain, E.-H. Han, and J. Srivastava, Web Mining, Pattern Discovery from World Wide Web Transactions, *Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'97)*, 1997.

[8] D. Ngu and X. Wu, SiteHelper: A Localized Agent that Helps Incremental Exploration of the World Wide Web, *6th International WWW Conference*, 1996.

[9] J. S. Park, M.-S. Chen, and P. S. Yu, An effective Hash Based Algorithm for Mining Association Rules, *ACM SIGMOD*, 175-186, 1995.

[10] Y. Zhuge and H. Garcia-Molina, Graph Structed Views and Their Incremental Maintance, *ICDE'98*, 116-125, 1998.