TRIPS and TIDES: New Algorithms for Tree Mining

Shirish Tatikonda[1], Srinivasan Parthasarathy [*1], and Tahsin Kurc[2]
Department of Computer Science and Engineering [1],
Department of Biomedical Informatics[2]
The Ohio State University, Columbus, OH 43210, USA

ABSTRACT

Recent research in data mining has progressed from mining frequent itemsets to more general and structured patterns like trees and graphs. In this paper, we address the problem of frequent subtree mining that has proven to be viable in a wide range of applications such as bioinformatics, XML processing, computational linguistics, and web usage mining. We propose novel algorithms to mine frequent subtrees from a database of rooted trees. We evaluate the use of two popular sequential encodings of trees to systematically generate and evaluate the candidate patterns. The proposed approach is very generic and can be used to mine embedded or induced subtrees that can be labeled, unlabeled, ordered, unordered, or edge-labeled. Our algorithms are highly cache-conscious in nature because of the compact and simple array-based data structures we use. Typically, $L1$ and $L2$ hit rates above 99% are observed. Experimental evaluation showed that our algorithms can achieve up to several orders of magnitude speedup on real datasets when compared to state-of-the-art tree mining algorithms.

Categories and Subject Descriptors

H.2 [**DATABASE MANAGEMENT**]: Database Applications—*Data Mining*

General Terms

Algorithms

Keywords

Tree mining, Prufer sequences, Depth first order codes, Frequent patterns, Embedding lists

1. INTRODUCTION

Data mining or knowledge discovery deals with finding interesting patterns or information that is hidden in large datasets. Recently, researchers have started proposing techniques for analyzing structured and semi-structured datasets. Such datasets can often be represented as graphs or trees. This has led to the development of numerous graph mining and tree mining algorithms in the literature[23, 9, 11, 24, 15, 17, 20]. In this article we focus on the development of efficient algorithms for mining trees.

Mining for frequent subtrees has been shown to be useful in various applications. For example, mining user browsing and access patterns on the world wide web can benefit from frequent subtree mining [24, 5]. Also, researchers [15, 27] have applied frequent subtree algorithms for solving bioinformatic tasks like finding the similarity of phylogenetic trees. We have demonstrated that mining embedded sub-trees can be helpful in estimating selectivity of XML twig queries and in summarizing the XML documents [21]. Several researchers have developed association rule mining algorithms, and structure classifiers for XML documents based on their subtree structures [16, 25]. Design of network multicast routing algorithms can also benefit from frequent subtree discovery [6]. Wang *et al* [22] have also shown that, in the context of movie documents, mining for typical structures (trees) can reveal substantial information about the data.

In this work, we develop a novel method for mining induced or embedded subtrees from various types of rooted trees (ordered, unordered, labeled, unlabeled, edge-labeled etc.) [24, 10, 17, 20]. The proposed approach relies on a sequential encoding of the database trees. Specifically we examine the use of Prüfer sequence and Depth First order Sequence (DFS) based encodings for this purpose. Our approach exploits the structure of the sequence encoding to generate candidate subtrees efficiently and to compute the support of said candidates in the database easily. It leverages embedding lists in generating a set of complete and non-redundant candidate patterns. One of the major advantages of our approach is that the sequential encoding allows us to operate on an array based representation of trees. This results in very efficient and cache-conscious algorithms that can suitably leverage the features of modern and emerging architectures [7].

Our experimental results show that the proposed approach is significantly faster than competing strategies on both real and synthetic datasets (up to 355 times speedup). A more in-depth analysis revealed that the proposed approach is extremely cache conscious (L1 hit rate: 99.6, L2 hit rate: 99.94, Cycles Per Instruction (CPI): 0.72) when compared to TreeMiner [24] (L1 hit rate: 96.59, L2 hit rate: 99.96, CPI: 1.2). We find that, on the real and synthetic datasets

*Email: srini@cse.ohio-state.edu

CIKM'06, November 5–11, 2006, Arlington, Virginia, USA.

we evaluated, the Prüfer sequence based encoding shows a marginal yet consistent performance improvement over the DFS based encoding schemes.

Specifically, we make the following contributions in this article: **First**, we propose a new algorithm to mine frequent subtrees based on sequence encoding strategies (Prüfer and DFS codes), that are very efficient on real datasets. **Second**, we develop a novel methodology for generating complete and non-redundant candidate subtrees that is based on sequential encodings of trees. **Third**, we show that the proposed algorithms are generic in nature and can be adopted to mine different types of trees.

2. RELATED WORK

There exist several algorithms in the literature that focus on mining various types of trees such as *free* trees, *rooted* trees, and *ordered* trees. Chi *et al* [3] present an excellent overview on frequent tree mining. One of the first algorithms in this area, *TreeMiner*, was proposed by Zaki [24]. TreeMiner mines for embedded subtrees from a forest (set) of rooted, ordered, and labeled trees. Influenced by the design of Eclat [26], TreeMiner represents trees in vertical format and uses scoping to prune the search space and efficiently mine for frequent sub-trees. A limitation of this method is that it uses pointer-based dynamic data structures and uses a lot of memory as we demonstrate in our experimental evaluation.

Wang *et al* [20] have recently proposed two algorithms, *Chopper* and *XSpanner* to mine frequent embedded subtrees. Chopper recasts subtree mining into sequence mining and uses PrefixSpan [12] to compute the set of frequent sequences. These frequent sequences correspond to candidate subtrees that are evaluated against the database and those sub-trees that are infrequent are pruned away. A limitation of this method, as pointed out by the authors, is that it will only be effective if the set of candidate patterns (from the sequence mining step) contains few false positives. Unfortunately in many real datasets this is not the case. XSpanner falls under the category of algorithms, which generate frequent patterns without *explicit* candidate generation, drawing inspiration from FPgrowth [8]. It recursively projects the database and generates frequent subtrees. A potential problem with this approach is that the recursive projection may again lead to a lot of pointer chasing and poor cache behavior.

There exist various other algorithms which deal with a slightly different problem. Very recently, Tan *et al* [17] have proposed *IMB3-Miner* that mines embedded subtrees using *Tree Model Guided* enumeration. This algorithm uses *occurrence match support* instead of traditional transaction-based support (definition that is used in frequent itemset mining). Asai *et al* [1] have proposed an algorithm *Freqt* that enumerates frequent ordered induced subtrees in a set of ordered trees. CMTreeMiner [4] proposed by Chi *et al* discovers all closed and maximal frequent subtrees from a set of rooted unordered trees. Termier *et al* have proposed two algorithms viz. Dryade and DryadeParent [18, 19] to mine closed frequent subtrees. They assume that no two siblings of a tree node can have the same labels. This assumption makes the complex subtree mining very simple and it might not be applicable in many real-world scenarios. Nijssen *et al* [10, 11] have proposed an efficient graph mining algorithm, *Gaston*. *Gaston* (and also other well known graph mining algorithms like *g-Span*) can mine for special instances of graphs i.e., paths and trees. *Gaston* mines for free trees; moreover, it can only mine induced subtrees. Extension of such approaches for mining embedded subtrees is not trivial.

Sequential representations have been used by many researchers in several areas. For example, Prüfer sequences are used by Rao *et al* in the context of tree indexing [14]. They have proposed a holistic method, PRIX, to efficiently index XML documents by transforming the tree isomorphism problem into subsequence matching. Basagni *et al* [2] have used prüfer sequences to efficiently encode the Steiner trees resulting from multicast groups, in their dynamic source multicast protocol.

3. PRELIMINARIES

Graphs and Trees: A *graph*, $G = (V, E)$ consists of set of vertices (or nodes) V, and set of edges $E \subseteq V$ X V. A graph is called as *labeled graph* if its vertices and/or edges are labeled. Graphs with no labels are referred to as *unlabeled graphs*. A *path* in the graph is a sequence of vertices, $v_1 v_2 ... v_n$ such that $(v_i, v_{i+1}) \in E, i = 1, 2, ..., n-1$. A graph is said to be *connected* if for any two vertices v_i and $v_j, i \neq j$, there exists a path between them. A *Cycle* of length n in the graph corresponds to a path whose start and end vertices are the same, such as $v_1 v_2 ... v_n v_1$. A *tree* is a connected graph that has no cycles. A *rooted tree* is a tree in which a vertex is distinguished from other vertices that is known as the *root* of the tree. Consider the path from root v_r to some vertex v_x in a rooted tree, $v_r v_1 ... v_m v_x$. All vertices on the path from v_r to v_x are *ancestors* of v_x and v_x is a *descendant* to those vertices. v_x's immediate ancestor, v_m is the *parent* of v_x and v_x is a *child* of v_m. A node with no children is known as a *leaf* node and all other non-leaf nodes are referred to as *internal* nodes. Two nodes with the same parent are *siblings* to each other. A tree is said to be *ordered* if some order is imposed on each set of siblings. If there is no order specified among children of a node, then the tree is *unordered*. Henceforth, "tree" refers to a rooted ordered tree unless otherwise stated.

Subtrees: Tree $S = (V_i, E_i)$ is said to be an *induced* subtree of $T = (V, E)$ if S is connected, $V_i \subseteq V$, and $E_i \subseteq E$. In other words, $\forall e = (v_p, v_c) \in E_i$, v_p is the parent of v_c in T. Such a tree can be obtained by deleting vertices and edges from T. Tree $S = (V_e, E_e)$ is said to be an *embedded* subtree of $T = (V, E)$ if S is connected, $V_e \subseteq V$, and $\forall e = (v_a, v_d) \in E_e$, v_a is the ancestor of v_d in T. For a given subtree S, each occurrence of S in T is known as the *embedding* of S in T. By embedding we mean the set of vertices of T, which are matched with vertices in S.

Tree Traversal: Given a tree, nodes in the tree can be traversed in many different ways. Two methods of interest are *post-order traversal* and *pre-order traversal*. In *post-order traversal*, a node is traversed or visited only *after all* of its children are visited. In *pre-order traversal*, a node is visited *before any of* its children are visited. The node with the post-order traversal number (PON) 1 is referred to as the *left-most* leaf. The node with the largest pre-order traversal number is known as the *right-most* leaf. The unique path from the root to the left-most leaf is known as the *Left Most Path* (LMP) and the unique path from the root to the right-most leaf is referred as the *Right Most Path* (RMP). Pre-order traversal on the tree explores nodes in depth-first

manner. We hence refer to the sequence generated by pre-order traversal as *Depth First order Sequence* (DFS).

Prüfer Sequences: Prüfer sequences were first used by Heinz Prüfer to prove Cayley's formula in 1918 [13]. Prüfer sequences provide a bijection between the set of labeled trees on n vertices and the set of sequences of length $n-2$ on the labels 1 to n. A simple iterative algorithm can be used to construct the prüfer sequence of a tree with n vertices. The algorithm starts with an empty sequence. At each step, the leaf with the smallest label is removed and its parent is appended to an already constructed partial prüfer sequence. This process is repeated until only two vertices remain, i.e., it is repeated for $n-2$ iterations. In the resulting sequence, $(p_1, p_2, ..., p_{n-2})$, p_i is the parent of a node with the i^{th} smallest label. We extend this construction by repeating the procedure for n iterations to produce a sequence of length n. When the last vertex is removed we leave the corresponding entry in the prüfer sequence empty, denoted by "-". Interestingly, for a given sequence S of length $n-2$ on the labels 1 to n, there is a unique labeled tree whose Prüfer sequence is S. This can be proved fairly easily by induction on n.

In the above construction, labels are assumed to be unique i.e., there can exists at most one vertex with any given label. In practice, this is rarely true as multiple nodes in a tree can share the same label. Therefore, a unique labeling system is needed for our purpose with which prüfer sequences can be constructed. For a given tree in the database, we use the post-order traversal numbers of vertices as the unique set of labels over which the prüfer sequence is constructed. We refer to the prüfer sequence constructed from post-order traversal numbers as the *Numbered Prüfer Sequence (NPS)*. Furthermore, the *Label Sequence (LS)* of a tree is given by the sequence of labels of leaf nodes, which are deleted at each step. Precisely, NPS denotes the prüfer sequence represented using post-order numbers, and LS denotes the labels of leaf nodes which are deleted. Both NPS and LS are ordered by the post-order number (PON). We jointly refer to these sequences as our *Consolidated Prüfer Sequence* $CPS(T) = (NPS, LS)(T)$. Figure 1 (b) shows the example prüfer sequences for two trees, T_1 and T_2.

LEMMA 3.1. *(NPS, LS) uniquely represents a rooted, labeled tree.*

PROOF. We prove this informally by presenting the intuition. NPS is constructed using post-order traversal numbers, which are unique in the tree. NPS thus determines the topology of the tree, which can be constructed by following the steps of prüfer sequence construction, in reverse order. Once the topology is determined, LS gives the labels of nodes in the order given by post-order traversal. Hence, NPS and LS *uniquely* represents a rooted labeled tree in which multiple nodes can share the same label. □

Problem Statement: Let $D = \{T_1, T_2, ..., T_n\}$ denote a database of rooted trees and the support of a subtree S with respect to a tree T is given by:

$$sup(S,T) = \begin{cases} 1, & \text{if } S \text{ is a subtree of } T \\ 0, & \text{otherwise} \end{cases}$$

We define the support of S with respect to a database D as $sup(S,D) = \frac{\Sigma_{T_i \in D} sup(S,T_i)}{|D|}$. S is considered as a *frequent* pattern if $sup(S,D)$ is greater than or equal to a user-defined threshold *minsup*. Otherwise, S is said to be an *infrequent* pattern. Given a database of trees D and the minimum support threshold *minsup*, the goal is to mine all frequent embedded or induced subtrees i.e., $\{S / sup(S,D) \geq minsup\}$. If D contains rooted ordered (or unordered) trees, the goal of the subtree mining task is to mine for all frequent (embedded or induced) ordered (or unordered) subtrees. The definition of support can also be devised such that it takes the number of embeddings of S in T into account [17].

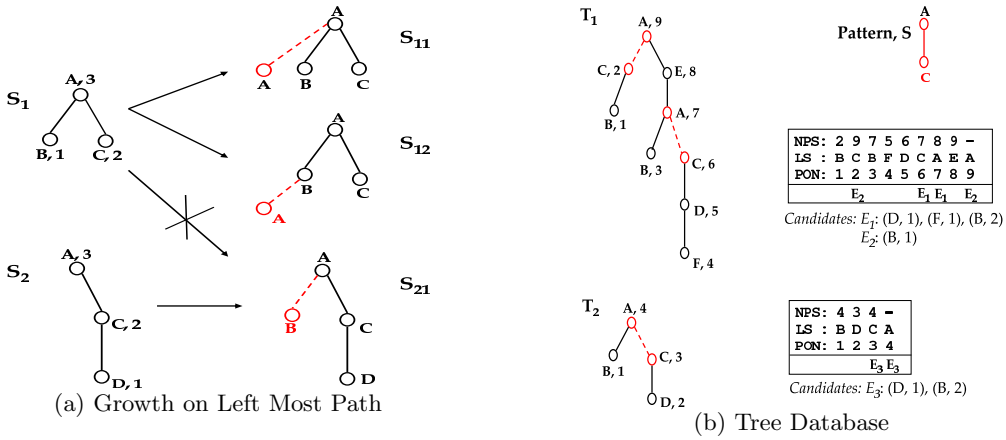
Broadly, tree mining algorithms can be categorized into two types, *Apriori-based* and *Pattern-growth* approaches. Candidate generation and support counting are the two key steps in each algorithm. In Apriori-based algorithms, candidates are generated level-wise. At level i, all candidate subtrees of length i are generated using frequent subtrees found in level $i-1$. Pattern-growth algorithms partition the search space into *equivalence classes*. They start with a node (a seed pattern) and generate all the frequent subtrees with that node as the root. The set of all subtrees resulting from a seed pattern is referred to as an *equivalence class*. In support counting step, generated candidate subtrees are evaluated for frequency. The challenge in candidate generation is to traverse the exponential search space efficiently. All algorithms leverage the anti-monotone property of frequent patterns in efficiently pruning certain branches in the search space. According to this property, no super pattern of an infrequent pattern can be frequent. The challenge in support counting is posed by the isomorphism problem especially in the case of embedded trees. Pattern-growth approaches are, in general, shown to be efficient compared to traditional apriori-based approaches.

4. PROPOSED APPROACH

Our scheme is based on the pattern-growth approach. It first transforms the database trees into sequences. Specific properties of these sequences are then exploited to efficiently generate the candidate subtrees. For a given subtree S, the sequences of all the trees in which S occurs are scanned to find the edges with which S can be extended. Each such edge together with S defines a new candidate subtree. Candidate subtrees with support greater than or equal to *minsup* are then processed recursively to generate bigger subtrees. The proposed method for generating candidates is *non-redundant* as the process is guided by database tree sequences (Section 4.1). Our novel method for candidate generation renders support counting step to be simple. Both candidate generation and support counting with respect to S can be done simultaneously in one scan of the sequences in which S occurs. This generic approach can be applied to any type of sequential encodings as long as they exhibit some specific properties (described later). In the following sections we will show how our approach can be applied to prüfer sequences and depth-first order sequences. We first describe our TRee mIning algorithm using Prüfer Sequences (*TRIPS*). We then point out the changes to be made for applying similar approach to depth-first order sequences.

4.1 Candidate Generation

In this section, we present our novel prüfer sequence based candidate generation technique. The pattern-growth approach is employed to systematically generate the candidate patterns (or subtrees). Hereafter, the terms "pattern" and "subtree" are used interchangeably. Assume we want to grow a subtree S with an edge e. Note that, growing S



(a) Growth on Left Most Path (b) Tree Database

Figure 1: LMP-based growth, Database trees & their prüfer sequences

with an edge and growing S with a vertex are synonymous. The number of ways in which e can be added to S are very large. e can be attached to any node in S and for a given node v, e can be at any position in the list of v's children[1]. To avoid such large number of possibilities, we restrict the positions at which e can be attached. Suppose the left most path (LMP) in S is $P = v_r v_1 ... v_m$, where v_r is the root of S and v_m is the left-most node. We allow e to be attached only to those nodes that are on the LMP of S. Furthermore, e is *always* added as the first child. By following such an orderly mechanism we restrict the number of possibilities in which S can be grown with e. It is fairly simple to see that *this mechanism indeed generates every possible subtree in the search space and generates each subtree only once*; this is because the tree mining approach starts from frequent 1-node subtrees and generates (frequent) subtrees of 2 nodes, 3 nodes, etc., recursively (see Section 4.4).

Each candidate pattern generated from S is denoted by the tuple (l, p), where l is the label of the node that is being attached to S and p is the position at which l is attached. In other words, p gives the PON of the node in S to which l is added as the first child. We refer to each such pair (l, p) as an *extension point* of S.

Example: Consider Figure 1 (a), which shows two patterns S_1, S_2 and their possible extensions. The edges, with which S_1 and S_2 are extended, are shown as dashed lines in red color. In S_1, the edge $e = (A, B)$ forms the left most path; hence, the new edge can be attached to any one of the two nodes of e. Therefore, S_{11}, S_{12} are valid extensions from S_1 and their extension points are $(A, 3)$ and $(A, 1)$ respectively. In S_{21}, node B is added to node A, which is on the LMP of S_2. Therefore, S_{21} is a valid extension from S_2 and its extension point is $(B, 3)$. Note that, even though S_{21} contains S_1 as a subtree it cannot be generated from S_1, as node D is not on the LMP of S_1.

It is worth noting that the extension points are defined with respect to a particular embedding of S in T, a tree in the database. Assume S has two embeddings E_1 and E_2 in a database tree T. An extension point of S in T with respect to E_1 might not be a valid extension point of S in T with respect to E_2. This is because the two embeddings might not have any node in common. An extension point with respect to one embedding might not even be connected to a node in another embedding. Even when two embeddings have a common node, the position of the common node might not be the same. Hence, *an extension point must always be defined with respect to a particular embedding.*

We explore the interesting relation between the LMP-based growth mechanism and prüfer sequences to develop a method that can efficiently generate the set of candidate patterns by taking the structure of database trees into account. This relationship is captured in the following lemma:

LEMMA 4.1. *Let S denote a pattern and its sequences are given by $CPS(S) = (NPS, LS)$. Extension point (l, p) of S corresponds to a prefix in the prüfer sequence of the resulting pattern, $R = S \cup (l, p)$.*

PROOF. For the sake of exposition, we only present the proof informally. Since (l, p) is an extension point of S, l is attached to a node at position p (say that node's label is v). By the definition of an extension point, v is on LMP of S and l is added as the first child of v. Therefore, post-order traversal on R visits l before any other node that was present in S. As the prüfer sequence is ordered by PON, l is the first node in the prüfer sequence of R. Hence, extensions on the LMP of S corresponds to the prefixes of R's prüfer sequence. More precisely, $CPS(R)$ can be obtained by prefixing $CPS(S)$ with $(p+1, l)$ i.e., $NPS(R) = (p+1)\ .\ NPS(S)$, and $LS(R) = l\ .\ LS(S)$, where "." symbolizes concatenation. Obviously, $NPS[v]$, $\forall v \in S$ must be incremented by 1 while constructing R's sequence. □

The LMP-based candidate generation is systematic, but it does not deal with the problem of exponential search space efficiently. Though it uses the anti-monotone property of frequent patterns to prune the search space, it can still potentially generate redundant candidate patterns. A pattern is considered *redundant* if its support is zero. Consider the example database and patterns in Figure 1. Both S_{11} and S_{12} are valid LMP-based extensions of S_1, but their support is zero as they do not occur either in T_1 or in T_2. Therefore, generating S_{11} and S_{12} is redundant and should be avoided. On the other hand, S_{21} is not only a valid extension of S_2 but also a subtree of T_1. Thus, S_{21}, whose support is 1, is not a redundant pattern. To avoid false positives like S_{11} and S_{12},

[1]If the trees are unordered then there is exactly one way in which e can be attached to v.

the candidate generation technique should take the topology of database trees into account. Such schema-conscious candidate generation is not new and has been used in several algorithms like Gaston [11] and IMB3-Miner [17]. This can be achieved by using a special data structure called *Embedding Lists*. For a given pattern S and a database tree T, the embedding list stores the information about *all* the embeddings of S in T. This information can be used to quickly find the vertices with which S can be grown such that the *resulting pattern is supported by* T. We use a simple array-based embedding structure (section 4.1.1) that makes our algorithm cache-conscious and facilitates efficient implementation. Since extension point, (l, p) is generated in schema-conscious manner, $S \cup (l, p)$ corresponds to a subtree in *at least* one tree in the database.

THEOREM 4.1. *Consider extending a pattern S whose embedding is E in T, a database tree. Say, v is a node that is on the left of E in $CPS(T)$. If v is connected to a node u that is part of E, then v defines a valid extension for S in T with respect to E.*

PROOF. From the definition of extension point: for a node to be a valid extension of S, it needs to be connected to a node that is on the LMP of E. From Lemma 4.1, the extensions of S corresponds to the prefixes of resulting pattern's prüfer sequence. Hence, the extensions of S with respect to E should present on the left of E in $CPS(T)$. Therefore, v is a *possible* extension for S with respect to E. Since, v is connected to u that is part of E, u *must* be on the LMP of E. If u is not on the LMP of E then v cannot be on the left of E in $CPS(T)$. Since v is connected a node u that is on the LMP of E, v defines a valid extension for S with respect to E. □

Theorem 4.1 plays a critical role in determining the nodes with which S can be extended to generate larger subtrees. For simplicity, consider the case of *induced* subtrees. According to the theorem, it is sufficient to check the nodes which are on the left of E when finding extensions with respect to E. For each node that is on the left of E (say v whose label is l), we need to check if v's parent is part of E or not. Precisely, we need to check whether $LS[NPS[v]]$ is part of E or not. We refer to this check as *connectivity check*. The position at which v is connected to E (i.e., p) can be computed by maintaining a simple counter that is updated while traversing E. Extension point (l, p) thus generated uniquely defines a new subtree. When evaluating a node v, connectivity checks need to be carried out against *each* of the embeddings that is on the right of v in $CPS(T)$. Since extension points are generated by traversing the CPS of trees in D, false positive (or redundant) patterns are avoided.

This method can only mine for *induced* subtrees as the connectivity check is performed only for the parent of v. When mining for embedded subtrees, node v is considered as an extension point if *any of its ancestors* is part of the embedding. Hence, connectivity checks need to be carried out for each ancestor of v and against each embedding. As soon as an ancestor that is part of some embedding E is found, the connectivity check for v with respect to E needs to be stopped. Essentially, connectivity check finds parent-child relationships (in case of induced subtrees) and ancestor-descendant relationships (in case of embedded subtrees).

Example: Consider the running example in Figure 1 (b), assume pattern S is being extended to generate the extension points. The embeddings of S in T_1 and T_2 are marked as dashed lines in red color. There are two embeddings E_1 and E_2 in T_1 and one embedding E_3 in T_2. In this example, we denote a node as $[l, n]$, where l is the label of the node and n is its PON. In T_1, search for extension points should be carried out from $[D, 5]$ to $[B, 1]$ (Lemma 4.1). Consider the node $v = [D, 5]$, whose parent is $[C, 6]$. $LS[NPS[v]] = C$ is part of embedding E_1 and hence v is a valid extension to S (Theorem 4.1). v is attached to E_1 at position 1 (the PON of D's parent in S). Hence, $(D, 1)$ defines an extension point for S in T_1 with respect to E_1. Consider the node $[F, 4]$, whose parent $[D, 5]$ is not part of any of the two embeddings. However, F's ancestor $[C, 6]$ is part of E_1. As a result, $(F, 1)$ is a valid extension point. Note that, $(F, 1)$ is not an extension point when mining for induced subtrees because the connectivity check stops with F's parent, $[D, 5]$. Node $[B, 3]$ is attached to E_1 at position 2 (the PON of B's parent in S) generating an extension point $(B, 2)$. Though $[C, 2]$ is part of E_2, it must be evaluated against E_1 as it resides on the left of E_1. Therefore, one node can be part of multiple embeddings but at different positions. Similarly, $[B, 1]$ gives an extension point $(B, 1)$ with respect to E_2. Note that, evaluating $[B, 1]$ involves a connectivity check with respect to both E_1 and E_2, because $[B, 1]$ is to the left of both E_1 and E_2 in $CPS(T_1)$. Since $[B, 1]$ is not connected to E_1, it does not generate any extension point with respect to E_1. Thus, the algorithm generates four extension points, $(D, 1)$, $(F, 1)$, $(B, 2)$, and $(B, 1)$ by scanning T_1. Similarly, the algorithm generates $(D, 1)$ and $(B, 2)$ by processing T_2.

4.1.1 Embedding Lists

We now briefly describe the structure of our array-based embedding lists. As mentioned earlier, this structure stores the information of all the embeddings of a pattern S in the given database tree T. This structure is maintained in a *per tree* basis. Each entry in the embedding list corresponds to a node in T that is matched with a node in S. These entries are of the form (m, ptr) where m is the PON of the matching node and ptr is the pointer to the parent node in the pattern. The matching nodes of the two different vertices are separated by a special entry, $(0, -2)$. Entries corresponding to the nodes, which are matched to the root of a pattern, have their ptr set to a value of -1. This is because they are the initial nodes which were added to the embedding list. Please note that the number of nodes in the last section of the embedding list (between last two separators) is equal to the number of embeddings of the pattern. Each node in the last section together with nodes pointed to directly or indirectly by them defines the LMP of the embedding. Therefore, when a pattern is being extended, the traversal of the embedding list is constrained only to the LMP of the embedding.

Example: Consider again the example database in Figure 1 (b). A pattern and corresponding embedding lists with respect to T_1 and T_2 are shown in Figure 2 (a). The first section of T_1's list (before the first separator) gives the PON numbers of nodes matched to the root of the pattern, A. Since the patterns are grown on LMP, A is extended with C and the corresponding matching nodes are recorded in the second section of the embedding list. For each matching node, a pointer is created to its parent node in the embedding. That is, when $A - C$ is extended with the node B, matching nodes of B point to A.

Such an array-based structure is simple and gels well with the recursive structure of our algorithm. To illustrate the point, assume the pattern $A - C$ has two extension points, B and D. First $A-C$ is extended with B (say, the resulting pattern is P) to generate all the subtrees extending from P. Once the mining with respect to P is completed, $A - C$ needs to be extended with D. At this time, the matching nodes of C in the embedding list can be replaced with the matching nodes of D as C's matchings nodes are no longer required. Hence, our array-based embedding list not only is simple but also facilitates efficient recursive implementation of our algorithm.

The space overhead incurred by embedding lists needs special mention as their maintenance demands a non-trivial amount of memory. The amount of overhead is directly dependent on the distribution of distinct labels (L) over the set of nodes (N) of a given tree (T). If $|L|$ and $|N|$ are comparable in number, then the number of matches in T for a given pattern would be very few. This results in small embedding lists. Instead, if $|L| << |N|$ then the number of matches for a given pattern would be large. Let us consider the worst case scenario where all the $|N|$ nodes in a given tree has the same label (i.e., $|L| = 1$), say v. This tree results in $|N|$ matches for a one-node pattern (every node in the tree is a match). When the pattern has two nodes (edge $v - v$), the number of matches *can be up to* $|N| * |N - 1|$. Add to that, when the dataset contains a very large number of such trees, the overhead incurred in maintaining the embedding lists of all trees increases tremendously. Such extreme but rare scenarios demand algorithms which do not require to maintain embedding lists.

4.2 Support Counting

The candidate generation mechanism makes the support counting step trivial, normally a costly step. This is done by maintaining a hash table H, known as *Support Structure* that stores the counts for each of the extension points found. Whenever a new extension point (l, p) is generated, H is probed to see if (l, p) is already present in H or not. If $(l, p) \in H$, then the count associated with (l, p) is incremented by 1 to reflect its new support. If not, the new entry is added to H with a count of 1. Once all the trees are scanned for extension points, H contains the set of extension points and associated supports. The resulting set of patterns is given by, $R = S \cup \{(l, p)/(l, p) \in H \wedge support((l, p)) \geq minsup\}$. Each pattern in R is considered again by the candidate generation algorithm to generate larger patterns.

Example: For the example database in Figure 1 (b), support structure is shown in Figure 2 (b). When T_1 is scanned for extension points for S, one entry is created for each of the four extension points resulted from T_1. Each entry is created with support 1 representing the occurrence of a new pattern in T_1. When $(D, 1)$ is generated from T_2, count associated with it is incremented by 1 as it is already present in H. Similarly, count associated with the extension point $(B, 2)$ is incremented by 1. If the minimum support is 2 then only two ($(D, 1)$ and $(B, 2)$) of the four extension points would be frequent and will be considered again for candidate generation.

4.3 DFS-based Approach

Though *TRIPS* is based on prüfer sequences, the idea of extending sequences for candidate generation can be applied to other sequence encodings. In this section, we describe such an application by devising Tree mIning algorithm using DEpth first order Sequences (*TIDES*). *TRIPS* and *TIDES* differ only in their candidate generation step. The support counting step is the same for both the algorithms. Method of candidate generation in both the algorithms is quite similar in its spirit except for a few differences. While *TRIPS* restricts the growth points to the Left Most Path of the pattern, *TIDES* restricts them to the Right Most Path (RMP). Therefore, an extension point (l, p) now refers to the addition of a node with label l at a node whose depth first order number is p. Use of RMP instead of LMP is motivated by the relation between RMP and depth first order sequences that is similar to the relation between LMP and prüfer sequences. Hence, similar to Lemma 4.1 and Theorem 4.1, we can prove the following lemmas in the context of depth first order sequences.

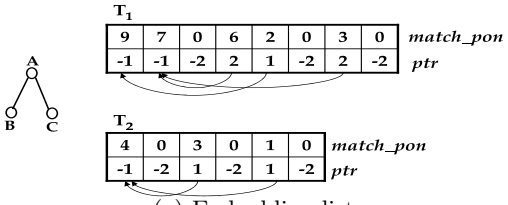
LEMMA 4.2. *Let S denote a pattern with depth first order sequence $DFS(S)$. Extension point (l, p) of S corresponds to a postfix in the depth first order sequence of the resulting pattern, $R = S \cup (l, p)$*

THEOREM 4.2. *Consider extending a pattern S whose embedding is E in T, a database tree. Say, v is a node that is on the right of E in $DFS(T)$. If v is connected to a node u that is part of E, then v defines a valid extension for S in T with respect to E.*

Since the depth first order sequences are based on preorder traversal of trees, growth on the right most path is the same as growing the sequence on the right hand side. The embedding list in *TIDES* keeps track of the nodes, which form the RMP of the pattern at hand.

Comparison of TRIPS and TIDES: *TIDES* can offer certain optimizations, which are difficult to achieve in *TRIPS*. In the candidate generation step, *TRIPS* scans the entire sequence that is to the left of embedding. This involves processing of nodes, which are not in the subtree of the pattern. Consider the example shown in Figure 1 (b) and assume the pattern being extended is an edge $A - D$. In T_1, only $[B, 3]$ and $[A, 4]$ are valid candidates, as they are present in the subtree of the pattern. In *TRIPS*, the connectivity check is performed not only on these two nodes but also on $[B, 1]$ and $[C, 2]$, as they are on the left of embedding of the pattern. *TIDES* can avoid such redundant processing by making use of scope data structures similar to the ones described in [24]. Scope structures enable us to focus only on nodes that are part of the subtree of the pattern. This benefit comes at the cost of *higher space overhead* that is incurred by storing the scope values (two integers) for every match. In other words, every embedding now needs to be associated with the scope. This overhead can be quite high if the number of embeddings of a pattern in the tree is very large (e.g. large trees with very few distinct labels). *TIDES* also has certain drawbacks when compared to *TRIPS*. The connectivity check on a node v involves traversing all the ancestors of v i.e., exploring the path from v to the root of the tree. Such a traversal is straight forward in *TRIPS*, as prüfer sequences are constructed based on parent-child relationships. To derive such relationships efficiently in *TIDES*, explicit pointers to parent nodes must be created for each node in the depth first order sequence.

In a nutshell, our sequence based candidate generation technique can be applied to any type of sequential encod-



(a) Embedding list

Extension point	Support
(D, 1)	2
(F, 1)	1
(B, 2)	2
(B, 1)	1

(b) Hash table for extension points

Figure 2: (a) Example embedding lists (b) Support structure for S in Figure 1-b

ing schemes as long as one can relate the growth in the trees to the growth in the corresponding sequences. This involves building relationships similar to the ones mentioned in Lemma 4.1 and Lemma 4.2. We now consolidate our main contributions in the next section.

4.4 Putting it All Together

Algorithm 1 Subtree Mining Algorithm

Require: $D = \{T_1, T_2, \ldots, T_N\}, minsup$
1: F_1 = readTrees (D)
2: **for all** v in F_1 **do**
3: mineTrees ($NULL$, (v, -1), D)
4: **end for**

In this section, we abstract out the ideas that are inherent in both *TRIPS* and *TIDES* to summarize our subtree mining algorithm (Algorithm 1). During the first scan of D, frequent nodes are identified (F_1). For each node v in F_1, the procedure *mineTrees* (Algorithm 2) is called to recursively mine the equivalence class of v, i.e., all subtrees whose root is v. *mineTrees* is a recursive procedure that applies the candidate generation algorithm on the given pattern, *pat*. At every level of recursion, a new pattern *newpat* is generated by extending *pat* with an extension point (l, pos). *tidlist* provides the projection of database with respect to *pat*. It contains the identifiers of trees in which *pat* occurs as a subtree. When extending *pat*, only the sequences of trees in *tidlist* need to be scanned. For a given pattern and an extension point, mineTrees first identifies the location of (l, pos) in each tree of *tidlist* and updates the corresponding embedding list (lines 3-8 in Algorithm 2). At the same time, *newtidlist* is constructed to include the identifiers of trees in which *newpat* has an embedding (line 6). Once the all embeddings of *newpat* are found, the candidate generation algorithm is applied to find the extension points with respect to *newpat* (lines 10-13). For each *frequent* extension point, mineTrees is called recursively to mine larger patterns (lines 14-18). Note that, Algorithm 2 shows two different loops in lines 3 and 10: one for finding the embedding of *newpat* and the other for extending the embedding. In the actual implementation, these two loops can be combined so that only one scan on the *tidlist* is performed. Thus, our algorithm, for a given pattern, can *generate and find the patterns simultaneously*, in a single scan of prüfer sequence.

As described in section 4.1, our algorithm can easily be adopted for mining both induced and embedded subtrees. Induced subtrees can be mined by adjusting the range of nodes for which the connectivity check is performed. Though the algorithm described deals with labeled trees, it is fairly

Algorithm 2 Mining a given pattern

mineTrees (*pat*, (l, *pos*), tidlist)
1: *newpat* = extend (*pat*, l, *pos*)
2: *newtidlist* = NULL
3: **for all** T in *tidlist* **do**
4: **if** (l, *pos*) is an extension point for *pat* in T **then**
5: update embedding list of T
6: add T to *newtidlist*
7: **end if**
8: **end for**
9: H = NULL
10: **for all** T in *newtidlist* **do**
11: Scan T for extension points of *newpat*
12: Add generated extension points to H
13: **end for**
14: **for all** h in H **do**
15: **if** $h.support \geq minsup$ **then**
16: mineTrees (*newpat*, ($h.l$, $h.pos$), *newtidlist*)
17: **end if**
18: **end for**

easy to see that it is applicable even if the trees which are *not labeled*. Moreover, it can be modified to handle *edge labels* by making simple extensions to the algorithm while maintaining the simplicity in representation. Each pair in $CPS(T)$ i.e., $(NPS[v], LS[v])$ represents an edge in T. If the edges are labeled, a new sequence can be incorporated into CPS to provide the edge label for each pair in the prüfer sequence. Note that, this change in the representation is minimal and we cannot do better than this.

Our algorithm can also be used to mine *unordered trees*. The only change that is required in Algorithm 2 is at line 11, scanning T for extension points. Assume, a node v in the tree T is matched with the pattern's root node. Since the children of v are unordered, connectivity check needs to be performed for every descendant of v (not just the descendants of the left-most child of v). We thus need to start the scan from a position that is immediately to the left of v, while searching for extension points. The modified pattern growth mechanism can potentially yield duplicate patterns. To avoid the generation of duplicates, a strict total order must be enforced on the set of labels, e.g., lexicographic ordering. Furthermore, we have shown that our algorithms can be applied to *any type of sequence encodings* that follow specific properties (last para of section 4.3). Hence, our approach is *generic* with respect to the representation of trees as well as the type of subtrees to be mined.

5. EXPERIMENTAL RESULTS

In this section, we present the experimental evaluation of the proposed approach on both synthetic and real datasets shown in Table 1. We compare the performance of our approach against TreeMiner [24]. All the experiments were performed on an Itanium 2 based system with 4 GB of main memory and a 1.3GHz processor. In the following discussion, dataset sizes are expressed in terms of number of trees, and *minsup* refers to the absolute support rather than the fraction, as defined earlier.

Name	Description
$DS1$	-T 10 -V 100
$DS2$	-T 10 -V 50
$DS3$	-f 15 -d 10 -n 100 -m 100
$DS4$	-f 15 -d 10 -n 1000 -m 100
CSLOGS	59,691 trees
TREEBANK	52,581 trees

Table 1: Datasets

5.1 Synthetic Datasets

In the first set of experiments, we evaluate our *TRIPS* algorithm on four different synthetic datasets. As shown in Table 1, the synthetic datasets $DS1$ and $DS2$ are generated using the PAFI toolkit developed by Kuramochi and Karypis ($PafiGen$)[2]. The datasets $DS3$ and $DS4$ are generated from the tree generator created by Zaki ($TreeGen$)[3]. The table also shows the parameter settings used for creating these datasets. Since $PafiGen$ can create only graphs, we have extracted spanning trees from these graphs and used in our analysis. Please note that the input parameters are just the guidelines to the generator (especially to $TreeGen$). Actual data that is created might not have the exact same statistics as the parameters provided. In the following, notation $D - num$ refers to a dataset D with *num* trees.

The performance of *TRIPS* against *TreeMiner* on the synthetic datasets is shown in Figures 3 (a) and (b). *TRIPS* is clearly scalable and continues to perform better as the dataset size increases. It achieves a speedup of 11.6 on $DS1$-$40K$ and a speedup of 8.6 on $DS4$-$100K$. We have observed that the *skewness* of the dataset directly affects the mining time. If the dataset is skewed, most of the frequent subtrees are produced from a small number of equivalence classes (F_1 in line 2 of Algorithm 1). This results in smaller mining times. The trees generated from *PafiGen* are less skewed compared to the trees from *TreeGen*. Therefore, the mining times of the *PafiGen* datasets are large, even when the datasets are small. For example, on the datasets with 40,000 trees, *TRIPS* (*TreeMiner*) spent 790 (8042) seconds in mining trees in $DS1$ whereas it spent only 315 (2711) seconds in mining trees in $DS4$. Higher speedups from the *Pafi* datasets shows that *TRIPS* can handle the difficulty posed by (lesser) skewness better than *TreeMiner*.

Figures 4 (a) and (b) depict the effect of *minsup* on the mining time and the number of frequent patterns found in the $PafiGen$ datasets. Not surprisingly, the execution time and the number of patterns increases as *minsup* decreases. Notably, the performance difference between *TRIPS* and *TreeMiner* continuously goes up with the decrease in *minsup*. On $DS2$-$40K$, as the value of *minsup* decreases from 5 to 1, the number of frequent patterns increases by 339 times. In this case, the execution time of *TreeMiner* increased by 65 times, whereas the execution time of *TRIPS* increased only by 15 times. This is because of two reasons: large number of join operations; large number of false positive patterns. *First*, at low support levels, *TreeMiner* has to perform costly join operations on a very large number of *scope lists*. Though *TRIPS* has to perform more connectivity checks (at low *minsup*), the increase in run time is not very high. The time taken for a connectivity check, in general, is very small because it is performed only on nodes present on LMP. Moreover, for a given pattern the set of nodes for which the connectivity check is performed *does not* depend on the support level. It only depends on the tree size and the matching nodes of the pattern (Theorem 4.1). Whereas in *TreeMiner*, the number of scope lists and hence the number of joins performed is highly dependent on the value of *minsup*. *Second*, *TreeMiner* naively performs the join operation on scope lists and hence it has the potential for generating a large number of false positive patterns. For example on $DS1$-$10K$, at $minsup = 1$, *TreeMiner* produced 1041 million candidates out of which only 173 million (approximately 16%) are frequent. On the other hand, *TRIPS* will *never* generate a redundant candidate pattern as the candidate generation is guided by the topology of the database trees. Thus, the strategies adopted by *TRIPS* enable us to *mine large datasets at very low support levels efficiently*. The performance of *TIDES* is similar to that of *TRIPS* on all synthetic datasets.

5.2 Real Datasets

In the second set of experiments, we evaluated the proposed algorithms on two different real datasets *CSLOGS* and *TREEBANK*. *CSLOGS*[4] contains web logs collected over a month in the Computer Science Department at the Rensselaer Polytechnic Institute. It contains 59,691 user browsing access patterns for 13,361 different web pages. On average, a tree in *CSLOGS* dataset has 12.94 nodes and the largest tree is of 428 nodes. The *TREEBANK* dataset[5] is made up of language treebanks, which have been widely used in computational linguistics. Treebanks are XML documents, which capture the syntactic structure of English text and provide a hierarchical representation of the sentences in the text by breaking them into syntactic units based on part of speech. *TREEBANK* has a total of 52,581 XML documents, which are narrow and have deep recursion of element names. The largest tree in this dataset has 648 nodes and the average number of nodes in a tree is 68.03.

Figure 5 (a) presents the run time comparison of our algorithms against *TreeMiner* on *CSLOGS*. The overall performance trend is similar to the one observed in synthetic datasets, but the speedup achieved is much larger in case of real datasets. At $minsup = 800$, *TreeMiner* spent 8748.5 seconds in mining the frequent patterns, whereas *TRIPS* spent only 24.61 seconds giving a speedup of 355.43. When *minsup* is decreased, time spent by *TreeMiner* increases much more quickly than *TRIPS* and *TIDES*. With the decrease in *minsup* from 1000 to 900, quick rise in mining time in all the algorithms is due to the presence of frequent patterns inside

[2] http://glaros.dtc.umn.edu/gkhome/pafi/overview/

[3] http://www.cs.rpi.edu/~zaki/software/

[4] http://www.cs.rpi.edu/~zaki/software/

[5] http://www.cs.washington.edu/research/xmldatasets/

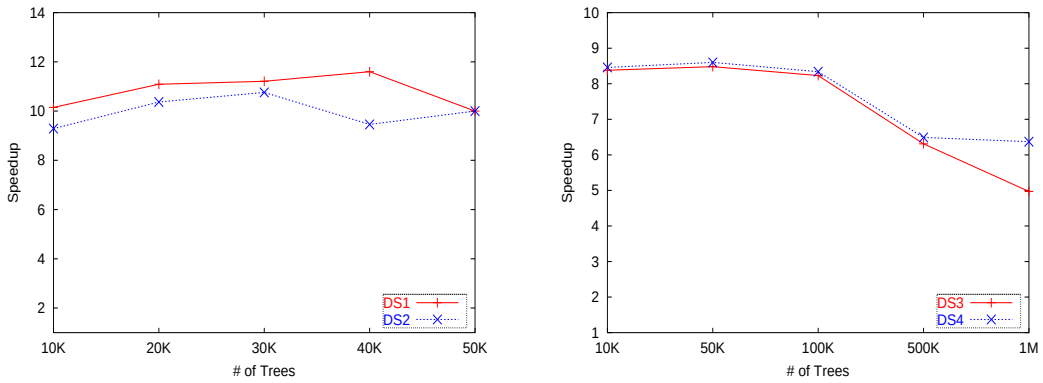


Figure 3: Performance of TRIPS on synthetic datasets (*minsup* = 1) (a) Pafi Trees (b) TreeGen Trees

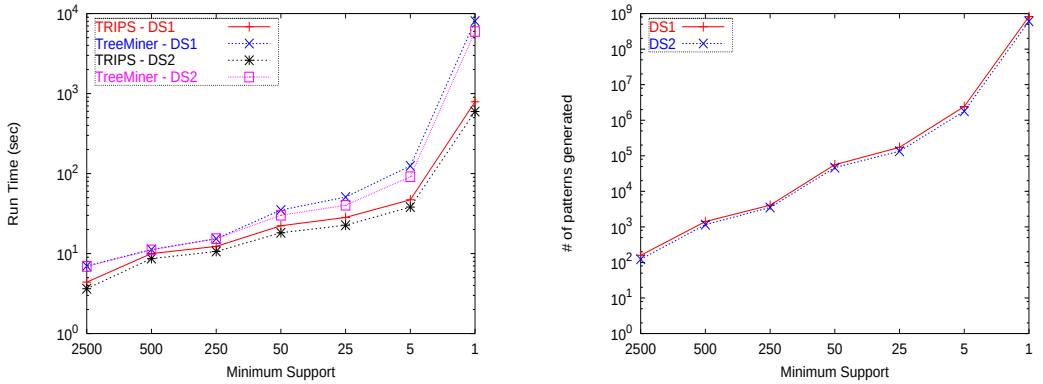


Figure 4: Effect of *minsup* on Pafi datasets with $50K$ trees (a) Run Time (b) No. of patterns generated

very large trees (which were not frequent at *minsup*= 1000). The mining time of *TreeMiner* increased by 300 times, while it is increased only by 6 times in *TRIPS*.

Figure 5 (b) shows the amount of virtual memory (Resident Set Size) used by the algorithms. *TIDES* memory consumption is slightly higher compared to *TRIPS* because of the extra pointers to the parent nodes (for traversing RMP efficiently). If *TIDES* employs the scope values for each embedding, the difference in memory usage would be much higher. At low support levels the amount of memory consumed by our algorithms is much lower compared to *TreeMiner*. At *minsup* = 800, *TreeMiner* used 2505 MB of memory and *TRIPS* completed the mining process by using just 60.84 MB of memory. Furthermore, the memory space consumed by *TreeMiner* raises exponentially with decrease in *minsup*. We suspect that this is because of the trees with large number of nodes and very few distinct labels. As mentioned in section 4.1.1, in such trees the number of embeddings for a given pattern would be high. This results in large scope lists (in *TreeMiner*) and large embedding lists (in our algorithms). Since an entry in embedding list is a lot simpler and smaller than an entry in scope list, *TreeMiner* experiences a quick rise in memory usage. At *minsup*=700, *TreeMiner* was aborted after 54 hours with memory usage more than 4 GB (*TRIPS*: 300.27 seconds & 335.97 MB). At very high support values (for example, 1200 in the figure), both *TRIPS* and *TIDES* use slightly more memory compared to *TreeMiner* because of the embedding lists.

Similar conclusions can be drawn from experiments conducted on the *TREEBANK* dataset (Figure 6). The time spent in mining and the amount of virtual memory used increased very sharply when using *TreeMiner*. For our algorithms, increase in execution time and virtual memory is slower when compared to *TreeMiner*, but is faster when compared to the increase observed on *CSLOGS* (Figures 5 & 6). This can be attributed to the narrow and deep structure of the XML trees in the *TREEBANK* dataset. Even in this case, our algorithms exhibit excellent performance in terms of both execution time and memory consumption. At a support level of $40,000$, *TreeMiner* ran for 7.3 hours consuming 2.2 GB of virtual memory. *TRIPS*, on the other hand, spent only 266 seconds with memory usage of 221 MB, resulting in a *speedup of* 98.93. Please note that for the experiment at *minsup* = $30,000$, *TreeMiner* was ran on a Itanium 2 based system with 12 GB of main memory. This system has been used only for this experiment.

Finally, a note on the performance differences between *TRIPS* and *TIDES*: as shown in Figures 5 & 6, *TRIPS* performs marginally better than *TIDES*. In terms of memory consumption, *TIDES* uses more memory than *TRIPS* for the reasons presented in the above discussion. We ascribe the difference in execution times to the structure of database trees and frequent patterns. Tree and frequent pattern structures govern the number of connectivity checks performed and affect the execution time. The difference in run times is higher on the *CSLOGS* dataset than the

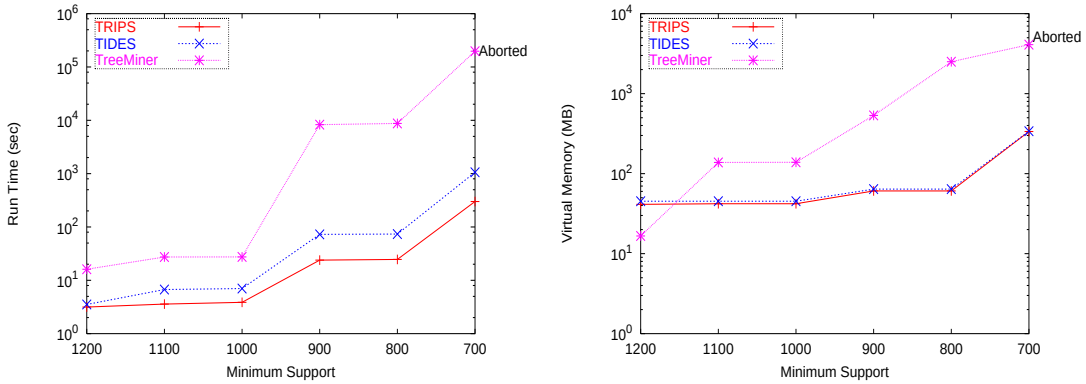


Figure 5: Performance on CSLOGS dataset (a) Run Time (b) Virtual Memory

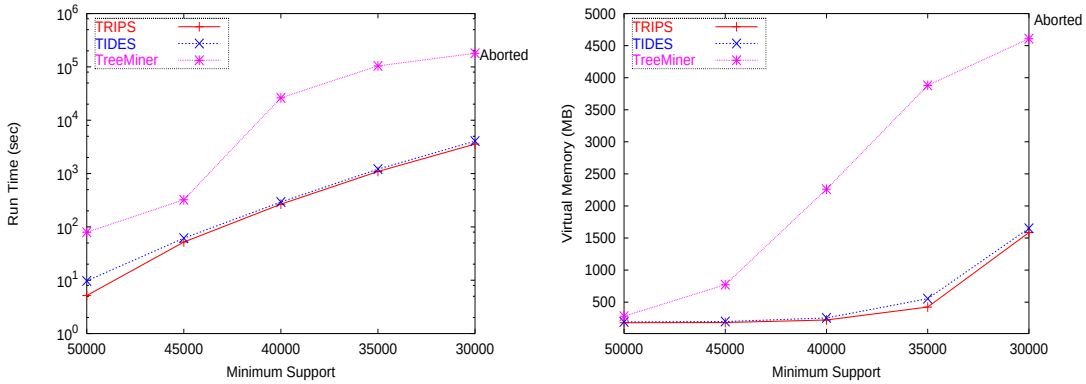


Figure 6: Performance on TREEBANK dataset (a) Run Time (b) Virtual Memory

TREEBANK dataset. However, in all cases both *TIDES* and *TRIPS* perform several orders of magnitude better than *TreeMiner*.

5.3 Performance Analysis

We use the *Intel VTune Performance Analyzers*[6] to perform cache performance analysis of our algorithms. This tool profiles the program execution at the level of source code and provides performance characteristics for each function in the implementation. Results presented in this section are obtained using the *TREEBANK* dataset at $minsup = 45000$. Both *TRIPS* and *TIDES* exhibit excellent cache performance with ($L1$ hit rate, $L2$ hit rate, CPI) of (99.6, 99.94, 0.72) and (99.7, 99.96, 0.7), respectively. Whereas, *TreeMiner* produced hit rates and CPI of (96.59, 99.96, 1.2). We observed that *TreeMiner* has slightly lower $L1$ hit rate. It also suffers from memory management issues. 49% of *TreeMiner*'s time is spent in the library, *libc-2.3.4.so* which contains several memory management routines like malloc() and free(). *TRIPS* and *TIDES*, on the other hand, spent only 2.3% and 3.7% of time in that library. The simple array-based data structures (sequence encodings and embedding lists) we use facilitate efficient memory management and make our algorithms cache-conscious. On the other hand, *TreeMiner*'s pointer-based dynamic structures make the memory management a difficult task. This problem worsens with decreasing values of *minsup* because of the exponential increase in memory size that is occupied by scope lists.

In all the three algorithms, candidate generation is the most expensive step. *TreeMiner* spent about 75 seconds (22.8% of run time) in scope-list joins (functions *check_ins*, *check_outs*, and *compare*). Both *TRIPS* and *TIDES* spent only about 31 seconds (63.7% of execution time) in traversing the embedding list and performing the connectivity checks. High percentage of time spent in user-level code depicts the better CPU utilization by *TRIPS* and *TIDES*. Similar results are observed on synthetic datasets, Figure 7. For example, CPU utilization of TRIPS (TreeMiner) is 99.3% (90.0%) and 99.3% (72.5%) on $DS1-50K$ and $DS4-1M$, respectively.

As part of our analysis, we also compared our approach against XSpanner [20], using the binary provided by the authors. Please note that the following results are obtained on an Intel P4 based system with a 2.4GHz processor and 4 GB of main memory. While we expected our approaches to outperform XSpanner, we also expected XSpanner to outperform TreeMiner as demonstrated in [20]. The results we obtained are surprising in that XSpanner achieves performance that is typically much worse than that of TreeMiner for many datasets. One possible explanation for this behavior is the fact that the memory footprint can be very large for such pattern-growth projection based approaches, potentially resulting in memory thrashing and large I/O costs [7].

[6] `http://www.intel.com/cd/software/products/asmo-na/eng/vtune/index.htm`

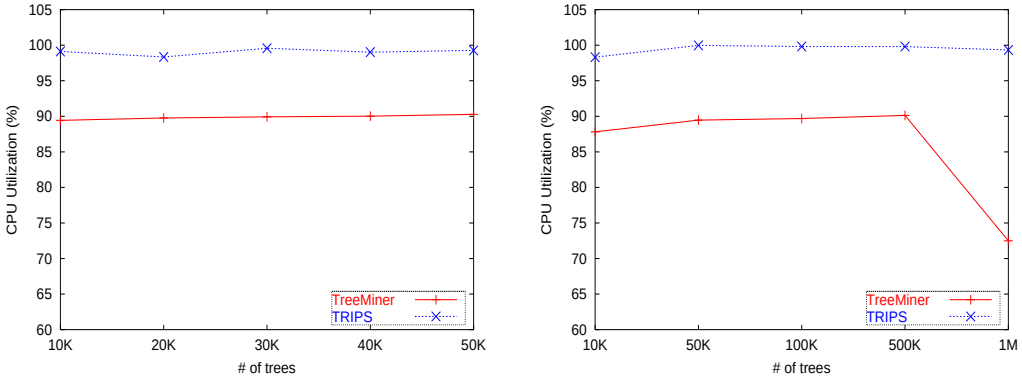


Figure 7: CPU Utilization on synthetic datasets (a) Pafi Trees ($DS1$) (b) TreeGen Trees ($DS4$)

Additionally, XSpanner is likely to suffer from poor cache performance due to the complexity of the pseudo-projection step. For example, on $DS1$ XSpanner took 1166 seconds whereas TreeMiner has taken only 38.31 seconds. On $DS3$, the mining time of XSpanner and *TreeMiner* are 245 and 3.1 seconds, respectively. On all the datasets, *TRIPS* ran faster than TreeMiner even on this architecture ($DS1 : 24$ seconds and $DS3 : 1.1$ seconds). We observed similar results on both $DS2$ and $DS4$. All the results were obtained from datasets with $50,000$ trees and at $minsup = 25$. Only on some toy datasets with 5 to 10 trees, the run times of XSpanner are comparable to *TreeMiner*. We could not evaluate XSpanner on the two real datasets because XSpanner binary expects the number of distinct labels to be less than $10,000$. In the *CSLOGS* and *TREEBANK* datasets, the maximum label number is $13,361$ and $1,387,266$, respectively.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed new algorithms for mining frequent induced or embedded subtrees from rooted trees. Novel sequential encoding based strategies are proposed, which facilitate the fast generation of complete and non-redundant set of candidate subtrees. We evaluated the proposed algorithms against *TreeMiner* on various synthetic and real datasets. We show that our approach achieves several orders of magnitude improvement on real datasets, when compared to state-of-the-art algorithms. Specifically, we obtained speedup of 355 and 98 on the *CSLOGS* and *TREEBANK* datasets, respectively. Our algorithms make use of simple array-based data structures and show excellent cache and memory performance. We also have demonstrated that our techniques are generic and can easily be adopted to mine various types of trees i.e., ordered, unordered, labeled, unlabeled, edge-labeled, etc.

In the future, we would like to extend the proposed approaches to devise parallel algorithms for tree mining. We would like to develop algorithms which do not employ embedding lists as the overhead incurred by them can potentially be prohibitive. We also want to characterize different tree mining algorithms in terms of their cache behavior and the type of datasets on which they perform well.

Acknowledgments. This work is supported by NSF grants CAREER-IIS-0347662, RI-CNS-0403342, and NGS-CNS-0406386. We would like to thank authors of TreeMiner and XSpanner for providing us the source code or binary. We also thank Amol Ghoting, Greg Buehrer, and Qian Zhu for the insightful discussions and valuable suggestions.

7. REFERENCES

[1] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. *Proceedings the 2nd SIAM International Conference on Data Mining (SDM2002)*, pages 158–174, 2002.

[2] S. Basagni, I. Chlamtac, et al. Location aware, dependable multicast for mobile ad hoc networks. *Computer Networks*, 36(5):659–670, 2001.

[3] Y. Chi, S. Nijssen, R. Muntz, and J. Kok. Frequent Subtree Mining-An Overview. *Fundamenta Informaticae*, 2005.

[4] Y. Chi, Y. Yang, Y. Xia, and R.R. Muntz. CMTreeMiner: Mining Both Closed and Maximal Frequent Subtrees. *The Eighth Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD04)*, 2004.

[5] R. Cooley, B. Mobasher, and J. Srivastava. Web Mining: Information and Pattern Discovery on the World Wide Web. *Proceedings of the 9th IEEE International Conference on Tools with Artificial Intelligence (ICTAI97)*, 1(2.1), 1997.

[6] J.H.R. Cui, J.R. Kim, D.R. Maggiorini, K.R. Boussetta, and M.R. Gerla. Aggregated Multicast - A Comparative Study. *Cluster Computing*, 8(1):15–26, 2005.

[7] A. Ghoting, G. Buehrer, and S. Parthasarathy et al. Cache conscious frequent pattern mining on a modern processor. In *Proceedings of the 31st international conference on very large databases (VLDB)*, 2005.

[8] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *ACM SIGMOD Record*, 2000.

[9] M. Kuramochi and G. Karypis. Frequent subgraph discovery. *Proceedings IEEE International Conference on Data Mining, ICDM 2001.*, pages 313–320, 2001.

[10] S. Nijssen and J.N. Kok. Efficient discovery of frequent unordered trees. *First International Workshop on Mining Graphs, Trees and Sequences*, pages 55–64, 2003.

[11] S. Nijssen and J.N. Kok. A quickstart in frequent structure mining can make a difference. *Proceedings of the 2004 ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 647–652, 2004.

[12] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.C. Hsu. PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. *Proceedings. 17th International Conference on Data Engineering*, 2001.

[13] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv für Mathematik und Physik*, 27:742–744, 1918.

[14] P. Rao and B. Moon. PRIX: indexing and querying XML using prufer sequences. *Data Engineering, 2004. Proceedings. 20th International Conference on*, pages 288–299, 2004.

[15] U. Ruckert and S. Kramer. Frequent free tree discovery in graph data. *Proceedings of the 2004 ACM symposium on Applied computing*, pages 564–570, 2004.

[16] H. Tan, T.S. Dillon, L. Feng, E. Chang, and F. Hadzic. X3-Miner: Mining Patterns from XML Database. *Proceedings Data Mining. Skiathos, Greece*, 2005.

[17] H. Tan, T.S. Dillon, F. Hadzic, E. Chang, and L. Feng. IMB3-Miner: Mining Induced/Embedded Subtrees by Constraining the Level of Embedding. *The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2006.

[18] A. Termier, M. C. Rousset, and M. Sebag. DRYADE: A New Approach for Discovering Closed Frequent Trees in Heterogeneous Tree Databases. *Proceedings of Fourth IEEE International Conference on Data Mining*, 2004.

[19] A. Termier, M.C. Rousset, M. Sebag, K. Ohara, T. Washio, and H. Motoda. Efficient mining of high branching factor attribute trees. *Proceedings of Fifth IEEE International Conference on Data Mining, 2005*, pages 785–788, 2005.

[20] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi. Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining. *The Eighth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD04)*, 2004.

[21] C. Wang, S. Parthasarathy, and R. Jin. A Decomposition-Based Probabilistic Framework for Estimating the Selectivity of XML Twig Queries. *International Conference on Extending Database Technology*, 2006.

[22] K. Wang and H. Liu. Discovering typical structures of documents: a road map approach. *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, 1998.

[23] X. Yan and J. Han. gSpan: graph-based substructure pattern mining. *Proceedings of IEEE International Conference on Data Mining (ICDM)*, pages 721–724, 2002.

[24] M.J. Zaki. Efficiently mining frequent trees in a forest. *Proceedings of the eighth ACM SIGKDD conference on Knowledge discovery and data mining*, 2002.

[25] M.J. Zaki and C.C. Aggarwal. XRules: an effective structural classifier for XML data. *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 316–325, 2003.

[26] M.J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, et al. New algorithms for fast discovery of association rules. *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, pages 283–296, 1997.

[27] S. Zhang and J. T. L. Wang. Mining Frequent Agreement Subtrees in Phylogenetic Databases. *Proceedings of the 6th SIAM International Conference on Data Mining (SDM 2006)*, pages 222–233, 2006.