Efficiently Mining Homomorphic Patterns from Large Data Trees

Xiaoying Wu [#] and Dimitri Theodoratos [*]

[#] *State Key Lab. of Software Engineering, Wuhan University, China*
xiaoying.wu@whu.edu.cn

[*] *New Jersey Institute of Technology, USA*
dth@cs.njit.edu

Abstract. Finding interesting tree patterns hidden in large datasets is an central topic in data mining with many practical applications. Unfortunately, previous contributions have focused almost exclusively on mining induced patterns from a set of small trees. The problem of mining homomorphic patterns from a large data tree has been neglected. This is mainly due to the challenging unbounded redundancy that homomorphic tree patterns can display. However, mining homomorphic patterns allows for discovering large patterns which cannot be extracted when mining induced or embedded patterns. Large patterns better characterize big trees which are important for many modern applications in particular with the explosion of big data.
In this paper, we address the problem of mining frequent homomorphic tree patterns from a single large tree. We propose a novel approach that extracts non-redundant maximal homomorphic patterns. Our approach employs an incremental frequency computation method that avoids the costly enumeration of all pattern matchings required by previous approaches. Matching information of already computed patterns is materialized as bitmaps a technique that not only minimizes the memory consumption but also the CPU time. We conduct detailed experiments to test the performance and scalability of our approach. The experimental evaluation shows that our approach mines larger patterns and extracts maximal homomorphic patterns from real datasets outperforming state-of-the-art embedded tree mining algorithms applied to a large data tree.

1 Introduction

Extracting frequent tree patterns which are hidden in data trees is central for analyzing data and is a base step for other data mining processes including association rule mining, clustering and classification. Trees have emerged in recent years as the standard format for representing, exporting, exchanging and integrating data on the web (e.g., XML and JSON). Tree data are adopted in various application areas and systems such as business process management, NoSQL databases, key-value stores, scientific workflows, computational biology and genome analysis.

Because of its practical importance, tree mining has been extensively studied [2, 10, 4, 14, 15, 7, 6, 11]. The approaches to tree mining can be basically characterized by two parameters: (a) the type of morphism used to map the tree patterns to the data structure, and (b) the type of mined tree data.

Mining homomorphic tree patterns. The morphism determines how a pattern is mapped to the data tree. The morphism definition depends also on the type of pattern con-

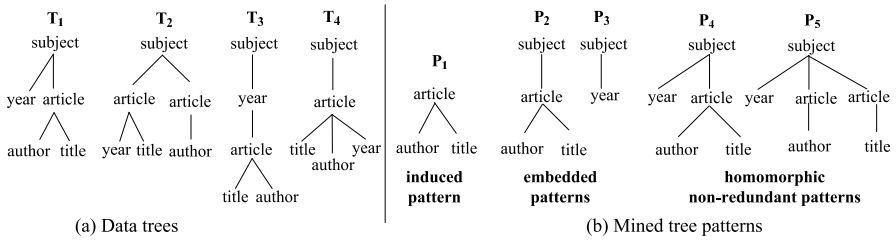


Fig. 1: Different types of mined tree patterns occurring in three of the four data trees.

sidered. In the literature two types of tree patterns have been studied: patterns whose edges represent parent-child relationships (*child* edges) and patterns whose edges represent ancestor-descendant relationships (*descendant* edges). Over the years, research has evolved from considering isomorphisms for mining patterns with child edges (*induced patterns*) [2, 4] into considering embeddings for mining patterns with descendant edges (*embedded patterns*) [10, 15, 14]. Because of the descendant edges, embeddings are able to extract patterns "hidden" (or embedded) deep within large trees which might be missed by the induced definition [14]. Nevertheless, embeddings are restricted because: (a) they are injective (one-to-one), and (b) they cannot map two sibling nodes in a pattern to two nodes on the same path in the data tree. On the other hand, homomorphisms are powerful morphisms that do not have those two restrictions of embeddings. We term patterns with descendant edges, mined through homomorphisms, *homomorphic* patterns. Formal definitions are provided in Section 2. As homomorphisms are more relaxed than embeddings, the mined homomorphic patterns are a superset of the mined embedded patterns.

Fig. 1(a) shows four data trees corresponding to different schemas to be integrated through the mining of large tree patterns. The frequency threshold is set to three. Fig. 1(b) shows induced mined tree patterns, embedded patterns and non-redundant homomorphic patterns. Fig. 1(b) includes the largest patterns that can be mined in each category. As one can see, the shown embedded patterns are not induced patterns, and the shown homomorphic patterns are neither embedded nor induced patterns. Further, the homomorphic patterns are larger than all the other patterns.

Large patterns are more useful in describing data. Mining tasks usually attach much greater importance to patterns that are larger in size, e.g., longer sequences are usually of more significant meaning than shorter ones in bioinfomatics [17]. As mentioned in [16] large patterns have become increasingly important in many modern applications.

Therefore, homomorphisms and homomorphic patterns display a number of advantages. First, they allow the extraction of patterns that cannot be extracted by embedded patterns. Second, extracted homomorphic patterns can be larger than embedded patterns. Finally, homomorphisms can be computed more efficiently than embeddings. Indeed, the problem of checking the existence of a homomorphism of an unordered tree pattern to a data tree is polynomial [9] while the corresponding problem for an embedding is NP-complete [8].

Mining patterns from a large data tree. The type of mined data can be a collection of small trees [2, 10, 4, 14, 15] or a single large tree. Surprisingly, the problem of mining tree patterns from a single large tree has only very recently been touched even though

a plethora of interesting datasets from different areas are in the form of a single large tree. Examples include encyclopedia databases like Wikipedia, bibliographic databases like PubMed, scientific and experimental result databases like UniprotKB, and biological datasets like phylogenetic trees. These datasets grow constantly with the addition of new data. Big data applications seek to extract information from large datasets. However, mining a single large data tree is more complex than mining a set of small data trees. In fact, the former setting is more general than the latter, since a collection of small trees can be modelled as a single large tree rooted at a virtual unlabeled node. Existing algorithms for mining embedded patterns from a collection of small trees [14] cannot scale well when the size of the data tree increases. Our experiments show that these algorithms cannot scale beyond some hundreds of nodes in a data tree with low frequency thresholds.

The problem. Unfortunately, previous work has focused almost exclusively on mining induced and embedded patterns from a set of small trees. The issue of mining *homomorphic patterns* from a *single large data tree* has been neglected.

The challenges. Mining homomorphic tree patterns is a challenging task. Homomorphic tree patterns are difficult to handle as they may contain redundant nodes. If their structure is not appropriately constrained, the number of frequent patterns (and therefore the number of candidate patterns that need to be generated) can be infinite.

Even if homomorphic patterns are successfully constrained to be non-redundant, their number can be much larger than that of frequent embedded patterns from the same data tree. In order for the mining algorithm to be efficient, new, much faster techniques for computing the support of the candidate homomorphic tree patterns need to be devised.

The support of patterns in the single large data tree setting cannot be anymore the number of trees that contain the pattern as is the case in the multiple small trees setting. A new way to define pattern support in the new setting is needed which enjoys useful monotonic characteristics.

Typically, one can deal with a large number of frequent patterns, by computing only maximal frequent patterns. In the context of induced tree patterns, a pattern is maximal if there is no frequent superpattern [4]. A non-maximal pattern is not returned to the user as there is a larger, more specific pattern, which is frequent. However, in the context of homomorphic patterns, which involve descendant edges, the concept of superpattern is not sufficient for capturing the specificity of a pattern. A tree pattern can be more specific (and informative) without being a superpattern. For instance, the homomorphic pattern P_4 of Fig. 1(b) is more specific than the homomorphic pattern P_5 without being a superpattern of P_5. Therefore, a new sophisticated definition for maximal patterns is required which takes into account both the particularities of the homomorphic patterns and the single large tree setting.

Contribution. In this paper, we address the problem of mining maximal homomorphic unordered tree patterns from a single large data tree. Our main contributions are:

- We define the problem of extracting homomorphic and maximal homomorphic unordered tree patterns with descendant relationships from a single large data tree. This problem departs from previous ones which focus on mining induced or embedded tree patterns from a set of small data trees (Section 2).

- We constrain the extracted homomorphic patterns to be non-redundant in order to avoid dealing with an infinite number of frequent patterns of unbounded size. In order to define maximal patterns, we introduce a strict partial order on patterns characterizing specificity. A pattern which is more specific provides more information on the data tree (Section 2).
- We design an efficient algorithm to discover all frequent maximal homomorphic tree patterns. Our algorithm wisely prunes the search space by generating and considering only patterns that are maximal and frequent or can contribute to the generation of maximal frequent patterns (Section 3).
- Our algorithm employs an incremental frequency computation method that avoids the costly enumeration of all pattern matchings required by previous approaches. An originality of our method is that matching information of already computed patterns is materialized as bitmaps. Exploiting bitmaps not only minimizes the memory consumption but also reduces CPU costs (Section 3).
- We run extensive experiments to evaluate the performance and scalability of our approach on real datasets. The experimental results show that: (a) the mined maximal homomorphic tree patterns are *larger* on the average than maximal embedded tree patterns on the same datasets, (b) our approach mines homomorphic maximal patterns up to *several orders of magnitude faster* than state-of-the-art algorithms mining embedded tree patterns when applied to a large data tree, and (c) our algorithm consumes only a *small fraction of the memory space* and *scales smoothly* when the size of the dataset increases (Section 4).

2 Preliminaries and Problem Definition

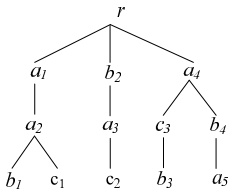
Trees and inverted lists. We consider rooted labeled trees, where each tree has a distinguished root node and a labeling function lb mapping nodes to labels. A tree is called *ordered* if it has a predefined left-to-right ordering among the children of each node. Otherwise, it is *unordered*. The *size* of a tree is defined as the number of its nodes. In this paper, unless otherwise specified, a tree pattern is a rooted, labeled, unordered tree.

For every label a in an input data tree T, we construct an inverted list L_a of the data nodes with label a ordered by their pre-order appearance in T. Fig. 2(a) and (b) shows a data tree and inverted lists of its labels.

Tree morphisms. There are two types of tree patterns: patterns whose edges represent child relationships (child edges) and patterns whose edges represent descendant relationships (descendant edges). In the literature of tree pattern mining, different types of morphisms are employed to determine if a tree pattern is included in a tree.

Given a pattern P and a tree T, a *homomorphism* from P to T is a function m mapping nodes of P to nodes of T, such that: (1) for any node $x \in P$, $lb(x) = lb(m(x))$; and (2) for any edge $(x, y) \in P$, if (x, y) is a child edge, $(m(x), m(y))$ is an edge of T, while if (x, y) is a descendant edge, $m(x)$ is an ancestor of $m(y)$ in T.

Previous contributions have constrained the homomorphisms considered for tree mining in different ways. Let P be a pattern with descendant edges. An *embedding* from P to T is an injective function m mapping nodes of P to nodes of T, such that: (1) for any node $x \in P$, $lb(x) = lb(m(x))$; and (2) (x, y) is an edge in P *iff* $m(x)$ is



(a) A tree T

$L_a = \{a_1, a_2, a_3, a_4, a_5\}$
$L_b = \{b_1, b_2, b_3, b_4\}$
$L_c = \{c_1, c_2, c_3\}$
$L_r = \{r\}$

(b) Inverted lists

pattern	pattern occurrences	occurence lists	bitmaps	support
P: A (B C)	A B C a_1 b_1 c_1 a_2 b_1 c_1 a_4 b_3 c_3 a_4 b_4 c_3	L_A={a_1,a_2,a_4} L_B={b_1,b_3,b_4} L_C={c_1,c_3}	L_A=11010 L_B=1011 L_C=101	2

(c) Occurrence information for pattern P on tree T

Fig. 2: A tree T, its inverted lists, and occurrence info. of pattern P on T.

an ancestor of $m(y)$ in T. Clearly, an embedding is also a homomorphism. Notice that, in contrast to a homomorphism, an embedding cannot map two siblings of P to two nodes on the same path in T. Patterns with descendant edges mined using embeddings are called *embedded* patterns. We call patterns with descendant edges mined using homomorphisms *homomorphic* patterns. In this paper, we consider mining homomorphic patterns. The set of frequent embedded patterns on a data tree T is a subset of the set of frequent homomorphic patterns on T since embeddings are restricted homomorphisms.

Pattern nodes occurrence lists. We identify an occurrence of P on T by a tuple indexed by the nodes of P whose values are the images of the corresponding nodes in P under a homomorphism of P to T. The set of occurrences of P under all possible homomorphisms of P to T is a relation OC whose schema is the set of nodes of P. If X is a node in P labeled by label a, the *occurrence list of* X on T is a sublist L_X of the inverted list L_a containing only those nodes that occur in the column for X in OC.

As an example, in Figure 2(c), the second and third columns give the occurrence relation and the node occurrence lists, respectively, of the pattern P on the tree T of Figure 2(a).

Support. We adopt for the support of tree patterns root frequency: the support of a pattern P on a data tree T is the number of distinct images (nodes in T) of the root of P under all homomorphisms of P to T. In other words, the *support* of P on T is the size of the occurrence list of the root of P on T.

A pattern S is *frequent* if its support is no less than a user defined threshold *minsup*. We denote by F_k the set of all frequent patterns of size k, also known as a *k-pattern*.

Constraining patterns. When homomorphisms are considered, it is possible that an infinite number of frequent patterns of unrestricted size can be extracted from a dataset. In order to exclude this possibility, we consider and define next non-redundant patterns. We say that two patterns P_1 and P_2 are *equivalent*, if there exists a homomorphism from P_1 to P_2 and vice-versa. A node X in a pattern P is *redundant* if the subpattern obtained from P by deleting X and all its descendants is equivalent to P. For example, the rightmost node C of P_3 and the rightmost node B of P_5 in Figure 3 are redundant. Adding redundant nodes to a pattern can generate an infinite number of frequent equivalent patterns which have the same support. These patterns are not useful as they do not provide additional information on the data tree. A pattern is *non-redundant* if it does not have redundant nodes. In Figure 3, patterns P_3 and P_5 are redundant while the rest of the patterns are non-redundant. Non-redundant patterns correspond to minimal tree-pattern queries [1] in tree databases. Their number is finite. We discuss later how to

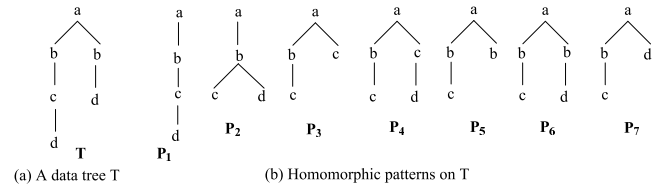


Fig. 3: A data tree and homomorphic patterns.

efficiently check patterns for redundancy by identifying redundant nodes. We set forth to extract only frequent patterns which are non-redundant but in the process of finding frequent non-redundant patterns we might generate also some redundant patterns.

Maximal patterns. In order to define maximal homomorphic frequent patterns, we introduce a specificity relation on patterns: A pattern P_1 is *more specific* than a pattern P_2 (and P_2 is *less specific* than P_1) iff there is a homomorphism from P_2 to P_1 but not from P_1 to P_2. If a pattern P_1 is more specific than a pattern P_2, we write $P_1 \prec P_2$. For instance, in Figure 3, $P_1 \prec P_i,\ i = 2, \ldots, 7$, and $P_2 \prec P_6$. Similarly, in Fig. 1, $P_2 \prec P_1$, $P_5 \prec P_3$, $P_4 \prec P_2$ and $P_4 \prec P_5$. Note that P_4 is more specific than P_5 even though it is smaller in size than P_5. Clearly, $\prec$ is a strict partial order. If $P_1 \prec P_2$, P_1 conveys more information on the dataset than P_2.

A frequent pattern P is *maximal* if there is no other frequent pattern P_1, such that $P_1 \prec P$. For instance, in Fig. 1, all the patterns shown are frequent homomorphic patterns and P_4 is the only maximal pattern.

Problem statement. Given a large tree T and a minimum support threshold *minsup*, our goal is to mine all maximal homomorphic frequent patterns from T.

3 Proposed Approach

Our approach for mining embedded tree patterns from a large tree iterates between the candidate generation phase and the support counting phase. In the first phase, we use a systematic way to generate candidate patterns that are potentially frequent. In the second phase, we develop an efficient method to compute the support of candidate patterns.

3.1 Candidate Generation

To generate candidate patterns, we adapt in this section the equivalence class-based pattern generation method proposed in [15, 14] so that it can address pattern redundancy and maximality. A candidate pattern may have multiple alternative isomorphic representations. To minimize the redundant generation of the isomorphic representations of the same pattern, we employ a canonical form for tree patterns [5].

Equivalence Class-based Pattern Generation. Let P be a pattern of size k-1. Each node of P is identified by its *depth-first position* in the tree, determined through a depth-first traversal of P, by sequentially assigning numbers to the first visit of the node. The *rightmost leaf* of P, denoted *rml*, is the node with the highest depth-first position. The *immediate prefix* of P is the sub-pattern of P obtained by deleting the *rml* from P. The *equivalence class* of P is the set of all the patterns of size k that have P as their

immediate prefix. We denote the equivalence class of P as $[P]$. Any two members of $[P]$ differ only in their rmls. We use the notation P_x^i to denote the k-pattern formed by adding a child node labeled by x to the node with position i in P as the rml.

Given an equivalence class $[P]$, we obtain its successor classes by expanding patterns in $[P]$. Specifically, candidates are generated by *joining* each pattern $P_x^i \in [P]$ with any other pattern P_y^j in $[P]$, including itself, to produce the patterns of the equivalence class $[P_x^i]$. We denote the above join operation by $P_x^i \otimes P_y^j$. There are two possible outcomes for each $P_x^i \otimes P_y^j$: one is obtained by making y a sibling node of x in P_x^i, the other is obtained by making y a child node of x in P_x^i. We call patterns P_x^i and P_y^j the *left-parent* and *right-parent* of a join outcome, respectively.

As an example, in Figure 3, patterns P_1, P_2, P_3, P_5, and P_7 are members of class $[a/b/c]$; P_4 is a join outcome of $P_3 \otimes P_7$, obtained by making the rml d of P_7 a child of the rml c of P_3.

Checking Pattern redundancy The pattern generation process may produce candidates which are redundant (defined in Section 2). We discuss below how to efficiently check pattern redundancy by identifying redundant nodes. We exploit a result of [1] which states that: a node X of a pattern P is redundant iff there exists a homomorphism h from P to itself such that $h(X) \neq X$. A brute-force method for checking if a pattern is redundant computes all the possible homomorphisms from P to itself. Unfortunately, the number of the homomorphisms can be exponential on the size of P. Therefore, we have designed an algorithm which, given two patterns P and Q, compactly represents all the homomorphisms from P to Q in polynomial time and space (see Appendix). Our algorithm enhances the one presented in [9] which checks if there exists a homomorphism from one tree pattern to another while achieving the same time and space complexity. Its detailed description is omitted here in the interest of space.

During the candidate generation, we cannot however simply discard candidates that are redundant, since they may be needed for generating non-redundant patterns. For instance, the pattern P_5 shown in Figure 3(b), is redundant, but it is needed (as the left operand in a join operation with P_7) to generate the non-redundant pattern P_6 shown in the same figure. Clearly, we want to avoid as much as possible generating patterns that are redundant. In order to do so, we introduce the notion of *expandable* pattern.

Definition 1 (Expandable Pattern). *A pattern P is* expandable, *if it does not have a redundant node X such that: (1) X is not on the rightmost path of P, or (2) X is on the rightmost path of P and L_X is equal to $L_{X_1} \cup \ldots \cup L_{X_k}$, where $X_1, \ldots, X_k$ are the images of node X under a homomorphism from P to itself.*

Based on Definition 1, if a pattern is non-expandable, every expansion of it is redundant. Therefore, only expandable patterns in a class are considered for expansion.

Finding Maximal Patterns. One way to compute the maximal patterns is to use a post-processing pruning method. That is, first compute the set S of all frequent homomorphic patterns, and then do the maximality check and eliminate non-maximal patterns by checking the specificity relation on every pair of patterns in S. However, the time complexity of this method is O($|S|^2$). It is, therefore, inefficient since the size of S can be exponentially larger than the number of maximal patterns.

We have developed a better method which can reduce the number of frequent patterns that need to go through the maximality check. During the course of mining frequent patterns, the method locates a subset of frequent patterns called locally maximal patterns. A pattern P is *locally maximal* if it is frequent and there exists no frequent pattern in the class $[P]$. Clearly, a non-locally maximal pattern is not maximal. Then, in order to identify maximal patterns, we check only locally maximal patterns for maximality. Our experiments show that this improvement can dramatically reduce the number of frequent patterns checked for maximality.

3.2 Support Computation

Recall that the support of a pattern P in the input data tree T is defined as the size of the occurrence list L_R of the root R of P on T (Section 2). To compute L_R, a straightforward method is to first compute the relation OC which stores the set of occurrences of P under all possible homomorphisms of P to T and then "project" OC on column R to get L_R. Fortunately, we can do much better using a twig-join approach to compute L_R without enumerating all homomorphisms of P to T. Our approach for support computation is a complete departure from existing approaches.

A holistic twig-join approach. In order to compute L_X, we exploit a holistic twig-join approach (e.g., $TwigStack$ [3]), the state of the art technique for evaluating tree-pattern queries on tree data. Algorithm $TwigStack$ works in two phases. In the first phase, it computes the matches of the individual root-to-leaf paths of the pattern. In the second phase, it merge joins the path matches to compute the results for the pattern. $TwigStack$ ensures that each solution to each individual query root-to-leaf path is guaranteed to be merge-joinable with at least one solution of each of the other root-to-leaf paths in the pattern. Therefore, the algorithm can guarantee worst-case performance *linear* to the size of the data tree inverted lists (the input) and the size of the pattern matches in the data three (the output), i.e., the algorithm is optimal.

By exploiting the above property of $TwigStack$, we can compute the support of P at the first phase of $TwigStack$ when it finds data nodes participating in matches of root-to-leaf paths of P. There is no need to enumerate the occurrences of pattern P on T (i.e., to compute the occurrence relation OC).

The time complexity of the above support computation method is $O(|P| \times |T|)$, where $|P|$ and $|T|$ denote the size of pattern P and of the input data tree T, respectively. Its space complexity is the $min(|T|, |P| \times heigh(T))$. We note that, on the other hand, the problem of computing an unordered embedding from P to T is NP-complete [8]. As a consequence, a state-of-the-art unordered embedded pattern mining algorithm $Sleuth$ [14] computes pattern support in $O(|P| \times |T|^{2|P|})$ time and $O(|P| \times |T|^{|P|})$ space.

Nevertheless, the $TwigStack$-based method can still be expensive for computing the support of a large number of candidates, since it needs to scan fully the inverted lists corresponding to every candidate pattern. We present below an incremental method, which computes the support of a pattern P by leveraging the computation done at its parent patterns in the search space.

Computing occurrence lists incrementally. Let P be a pattern and X be a node in P labeled by a. Using $TwigStack$, P is computed by iterating over the inverted lists corresponding to every pattern node. If there is a sublist, say L_X, of L_a such that P can be computed on T using L_X instead of L_a, we say that node X can be *computed*

using L_X on T. Since L_X is non-strictly smaller than L_a, the computation cost can be reduced. Based on this idea, we propose an incremental method that uses the occurrence lists of the two parent patterns of a given pattern P to compute P.

Let pattern Q be a join outcome of $P_x^i \otimes P_y^j$. By the definition of the join operation, we can easily identify a homomorphism from each parent P_x^i and P_y^j to Q.

Proposition 1. *Let X' be a node in a parent Q' of Q and X be the image of X' under a homomorphism from Q' to Q. The occurrence list L_X of X on T, is a sublist of the occurrence list $L_{X'}$ of X' on T.*

Sublist L_X is the inverted list of data tree nodes that participate in the occurrences of Q to T. By Proposition 1, X can be computed using $L_{X'}$ instead of using the corresponding label inverted list. Further, if X is the image of nodes X_1 and X_2 defined by the homomorphisms from the left and right parent of Q, respectively, we can compute X using the *intersection*, $L_{X_1} \cap L_{X_2}$, of L_{X_1} and L_{X_2} which is the sublist of L_{X_1} and L_{X_2} comprising the nodes that appear in both L_{X_1} and L_{X_2}.

Using Proposition 1, we can compute Q using only the occurrence list sets of its parents. Thus, we only need to store with each frequent pattern its occurrence list set. Our method is space efficient since the occurrence lists can encode in linear space an exponential number of occurrences for the pattern [3]. In contrast, the state-of-the-art methods for mining embedded patterns [15, 14] have to store information about all the occurrences of each given pattern in T.

Occurrence lists as bitmaps. The occurrence list L_X of a pattern node X labeled by a on T can be represented by a bitmap on L_a. this is a bot array of size $|L_a|$ which has a '1' bit at position i iff L_X comprises the tree node at position i of L_a. Then, the occurrence list set of a pattern is the set of bitmaps of the occurrence lists of its nodes. Figure 2(c) shows an example of bitmaps for pattern occurrence lists.

As verified by our experimental evaluation, storing the occurrence lists of multiple patterns as bitmaps results in important space savings. Bitmaps offer CPU cost saving as well by allowing the translation of pattern evaluation to bitwise operations. This bitmap technique is initially introduced and exploited in [12, 13] for materializing tree-pattern views and for efficiently answering queries using materialized views.

3.3 The Tree Pattern Mining Algorithm

We present now our homomorphic tree pattern mining algorithm called *HomTreeMiner* (Fig. 4). The first part of the algorithm computes the sets containing all frequent 1-patterns F_1 (i.e., nodes) and 2-patterns F_2 (lines 1-2). F_1 can be easily obtained by finding inverted lists of T whose size (in terms of number of nodes) is no less than $minsup$. The total time for this step is $O(|T|)$. F_2 is computed by the following procedure: let X/Y denote a 2-pattern formed by two elements X and Y of F_1. The support of X/Y is computed via algorithm $TwigStack$ on the inverted lists $L_{lb(X)}$ and $L_{lb(Y)}$ that are associated with labels $lb(X)$ and $lb(Y)$, respectively. The total time for each 2-pattern candidate is $O(|T|)$.

The main part of the computation is performed by procedure $MineHomPatterns$ which is invoked for every frequent 2-pattern (Lines 3-4). This is a recursive procedure. It tries to join every $P_x^i \in [P]$ with any other element $P_y^j \in [P]$ including P_x^i itself.

Input: inverted lists $\mathcal{L}$ of tree T and $minsup$.
Output: all the frequent maximal patterns $\mathcal{M}$ in T .

1. F_1 := {frequent 1-patterns};
2. F_2 := {classes $[P]_1$ of frequent 2-patterns};
3. **for** (every $[P] \in F_2$) **do**
4. $MineHomPatterns([P], \mathcal{M} = \emptyset)$;
5. run the maximality checking procedure on $\mathcal{M}$;
6. **return** $\mathcal{M}$;

Procedure $MineHomPatterns([P], \mathcal{M})$

1. **for** (each $P_x^i \in [P]$) **do**
2. **if** (P_x^i is in canonical form and is expandable) **then**
3. $[P_x^i] := \emptyset$
4. **for** (each $P_y^j \in [P]$) **do**
5. **for** (each join outcome Q of $P_x^i \otimes P_y^j$) **do**
6. add Q to $[P_x^i]$ if Q is frequent;
7. add P_x^i to $\mathcal{M}$ if none of the members of $[P_x^i]$ is in canonical form;
8. $MineHomPatterns([P_x^i], \mathcal{M})$;

Fig. 4: Homomorphic Tree Pattern Mining Algorithm.

Then, it computes the support of each possible join outcome, and adds them to $[P_x^i]$ if they are frequent (Lines 1-6). Once all P_y^j have been processed, it checks if P_x^i is a locally maximal pattern. If so, P_x^i is added to the maximal pattern set $\mathcal{M}$ (Line 7). Then, the new class $[P_x^i]$ is recursively explored in a depth-first manner (Line 8). The recursive process is repeated until no more frequent patterns can be generated.

Once all the locally maximal patterns have been found, the maximality check procedure described in Sec. 3.1 is run to identify maximal patterns among the locally maximal ones and the results are returned to the user (Lines 5-6).

4 Experimental Evaluation

We implemented our algorithm *HomTreeMiner* and we conducted experiments to: (a) compare the features of the extracted (maximal) homomorphic patterns with those of (maximal) embedded patterns, and (b) study the performance of *HomTreeMiner* in terms of execution time, memory consumption and scalability.

To the best of our knowledge, there is no previous algorithm computing homomorphic patterns from data trees. Therefore, we compared the performance of our algorithm with state-of-the-art algorithms that compute embedded patterns on the same dataset.

Our implementation was coded in Java. All the experiments reported here were performed on a workstation equipped with an Intel Xeon CPU 3565 @3.20 GHz processor with 8GB memory running JVM 1.7.0 on Windows 7 Professional. The Java virtual machine memory size was set to 4GB.

Datasets. We have ran experiments on four real and benchmark datasets (see Appendix). Due to space limitation, we only present results of our experimental study on one real tree dataset called $Treebank$[1] derived from computational linguistics. The

[1] http://www.cis.upenn.edu/∼treebank

dataset is deep and comprises highly recursive and irregular structures. Its statistics are shown below.

Dataset	Tot. #nodes	#labels	Max/Avg depth	#paths
Treebank	2437666	250	36/8.4	1392231

4.1 Agorithm Performance

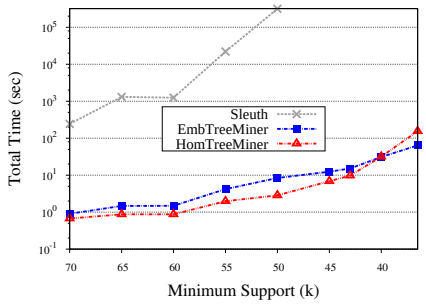
We compare the performance of *HomTreeMiner* with two unordered embedded tree mining algorithms *Sleuth* [14] and *EmbTreeMiner* [11]. *Sleuth* was designed to mine embedded patterns from a set of small trees. In order to allow the comparison in the single large tree setting, we adapted *Sleuth* by having it return as support of a pattern the number of its root occurrences in the data tree. *EmbTreeMiner* is a newer embedded tree mining algorithm which, as *HomTreeMiner*, exploits the twig-join approach and bitmaps to compute pattern support.

To the best of our knowledge, direct mining of maximal embedded patterns has not been studied in the literature. We therefore use post-processing pruning which eliminates non-maximal patterns after computing all frequent embedded patterns. For this task, we implemented the unordered tree inclusion algorithm described in [8]. As our experiments show, the cost of this post-processing step is not significant compared to the frequent pattern mining cost.

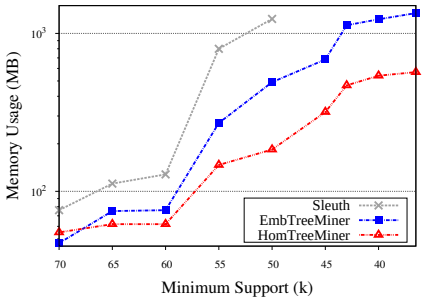
Execution time. We measure the total elapsed time for producing maximal frequent patterns at different support thresholds. The total time involves the time to generate candidate patterns, compute pattern support, and check maximality of frequent patterns. To allow *Sleuth*—which is slower—extract some patterns within a reasonable amount of time, we used a fraction of the Treebank dataset which consists of 35% of the nodes of the original tree. We measured execution times over the entire Treebank dataset in the scalability experiment.

Table 5(d) presents evaluation statistics. As one can see, the search space of a homomorphic pattern mining can be larger than that of embedded pattern mining for low support levels. *HomTreeMiner* computes 3.7 times more candidates and produces 4.25 times more frequent patterns than *EmbTreeMiner* at $minsup$ = 36.5k. Since $Treebank$ contains many deep, highly recursive paths, the search space of homomorphic patterns becomes large at low support levels.

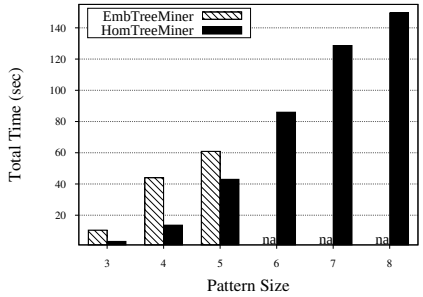
Figure 5(a) presents the total elapsed time of the three algorithms under different support thresholds. Due to prohibitively long times, we stopped testing *Sleuth* on support levels below 50k. We can see that *HomTreeMiner* runs orders of magnitude faster than *Sleuth* especially for low support levels. The rate of increase of the running time for *HomTreeMiner* is slower than that for *Sleuth* as the support level decreases. This is expected, since *HomTreeMiner* computes the support of a homomorphic pattern in time linear to the input data size, whereas this computation is exponential for embedded pattern miners (Sec. 3.2). Furthermore, *Sleuth* has to keep track of all possible embedded occurrences of a candidate to a data tree, and to perform expensive join operations over these occurrences. *HomTreeMiner* shows similar or better performance than *EmbTreeMiner* for support levels above 40K. The large number of candidate homomorphic patterns can negatively affect the time performance of *HomTreeMiner* at low support levels. For example, *HomTreeMiner* is 2.4 times slower than *EmbTreeMiner* in mining



(a) Run time vs. support



(b) Memory usage



(c) Run time vs. pattern size ($minsup$ = 36.5k)

minsup	morphism	# candidate patterns	# frequent patterns	max. size of freq. patterns
45k	Emb	299	23	4
	Hom	331	27	4
40k	Emb	419	41	5
	Hom	658	81	8
36.5k	Emb	557	65	5
	Hom	2044	276	8

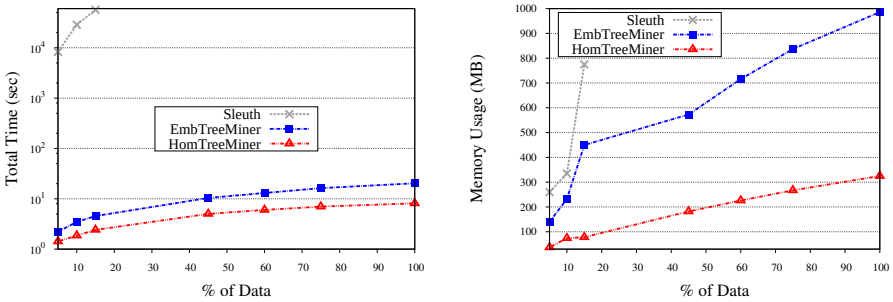
(d) Evaluation statistics

Fig. 5: Performance comparison on a fraction of Treebank.

frequent patterns at $minsup$ = 36.5k. However, even though the number of (candidate and frequent) homomorphic patterns is always larger than embedded patterns, this difference is not so pronounced in shallower datasets[1]. As a consequence, *HomTreeMiner* largely outperforms *EmbTreeMiner* in those cases both at higher and low support levels. This is due to its efficient computation of pattern support which does not require the enumeration of pattern occurrences as is the case with *EmbTreeMiner* [11].

Figure 5(c) presents the runtime *HomTreeMiner* and *EmbTreeMiner* need to compute the frequent patterns of a given size varying the pattern size. As we can see, *HomTreeMiner* is more efficient than *EmbTreeMiner* in computing frequent patterns of the same size even though the homomorphic patterns are more numerous.

Memory Usage. We measured the memory footprint of the three algorithms with varying support thresholds. The results are shown in Figure 5(b). We can see that *HomTreeMiner* has the best memory performance. It consumes substantially less memory than both *Sleuth* and *EmbTreeMiner* in all the test cases, whereas *Sleuth* consumes the largest amount of memory. This is mainly because *Sleuth* needs to enumerate and store in memory all the pattern occurrences for candidates under consideration. In contrast, *HomTreeMiner* avoids storing pattern occurrences by storing only bitmaps of occurrence lists which are usually of insignificant size. Although *EmbTreeMiner* does not store pattern occurrences, it still has to generate pattern occurrences as intermediate results the size of which can be substantial at low support levels.



(a) Run time (b) Memory usage

Fig. 6: Scalability comparison on Treebank with increasing size ($minsup$ = 5.5%).

dataset	morphism	# freq. patterns	# loc.max patterns	# max. patterns	%max. over freq. patterns	average #nodes	average height	maximum #nodes	#common max.patterns
Treebank (minsup=45k)	Emb	23	n/a	6	26.1	2.8	1.3	4	3
	Hom	27	10	5	16.1	2.8	1.4	4	
Treebank (minsup=40k)	Emb	41	n/a	9	22	3.2	1.4	5	4
	Hom	81	43	8	9.8	3.4	1.6	5	
Treebank (minsup=36.5k)	Emb	65	n/a	13	20	4.0	1.7	5	1
	Hom	276	90	11	3.9	5.3	2.0	7	

Table 1: Statistics for maximal frequent patterns mined from Treebank.

Scalability. In our final experiment, we studied the scalability of the three algorithms as we increase the input data size. We generated ten fragments of the Treebank dataset of increasing size and fixed $minsup$ at 5.5%.

Figure 6(a) shows that *HomTreeMiner* has the best time performance. It exhibits good linear scalability as we increase the input data size. The growth of the running time of *sleuth* is much sharper. *HomTreeMiner* outperforms *Sleuth* by several orders of magnitude. It also outperforms *EmbTreeMiner* by a factor of more than 2 on average.

Figure 6(b) shows that *HomTreeMiner* always has the smallest memory footprint. The growth of its memory consumption is much slower than that of both *sleuth* and *EmbTreeMiner*.

4.2 Comparison of mined maximal homomorphic and embedded patterns

We computed different statistics on frequent and maximal frequent patterns mined by *HomTreeMiner* and *EmbTreeMiner* from Treebank varying the support; the results are summarized in Figure 5(d) and Table 1. We can make the following observations.

First, *HomTreeMiner* is able to discover larger patterns than *EmbTreeMiner* for the same support level. As one can see in Figure 5(d) and Table 1, the maximum size of frequent homomorphic patterns and the maximum size and average number of nodes and height of maximum frequent homomorphic patterns is never smaller than that of the embedded patterns for the same support level.

Second, the number of maximal homomorphic patterns is never larger than the number of maximal embedded patterns for the same support (Column 5 of Table 1). Further, the number of homomorphic and embedded frequent patterns is substantially reduced if only maximal patterns are selected (Column 6 of Table 1). However the effect is larger on homomorphic patterns as the number of frequent homomorphic patterns is usually larger than that of embedded patterns for the same support level (Column 3 of Table 1).

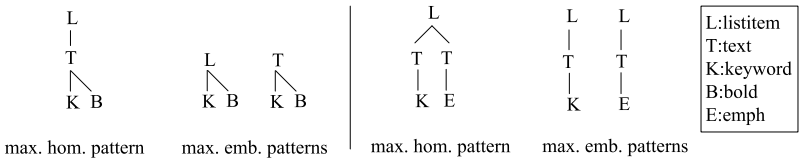


Fig. 7: Examples of maximal patterns mined from XMark at the same support level.

Third, by further looking at the mined maximal patterns we find that the embedded maximal patterns at a certain support level can be partitioned into sets which correspond one-to-one to the maximal homomorphic patterns at the same support level so that all the embedded patterns in a set are less specific than the corresponding homomorphic pattern. Figure 7 shows two pairs of embedded maximal patterns each from the same set in the partition and the corresponding maximal homomorphic pattern. The patterns are extracted from the XMark dataset[2]. Therefore, for a number of applications, maximal homomorphic patterns can offer more information in a more compact way.

5 Related Work

We now discuss, how our work relates to existing literature. The problem of mining tree patterns from a set of small trees has been studied since the last decade. Among the many proposed algorithms, only few mine unordered embedded patterns [10, 14].

$TreeFinder$ [10] is the first unordered embedded tree pattern mining algorithm. It is a two-step algorithm. In the first step, it clusters the input trees by the co-occurrence of labels pairs. In the second step, it computes maximal trees that are common to all the trees of each cluster. A known limitation of $TreeFinder$ is that it tends to miss many frequent patterns and is computationally expensive.

$Sleuth$ [14] extends the ordered embedded pattern mining algorithm $TreeMiner$ [15]. Unlike $TreeFinder$, $Sleuth$ uses the equivalence class pattern expansion method to generate candidates. To avoid repeated invocation of tree inclusion checking, $Sleuth$ maintains a list of embedded occurrences with each pattern. It defines also a quadratic join operation over pattern occurrence lists to compute support for candidates. The join operation becomes inefficient when the size of pattern occurrence lists is large. Our approach relies on an incremental stack-based approach that exploits bitmaps to efficiently compute the support in time linear to the size of input data.

The work on mining tree patterns in a single large tree/graph setting has so far been very limited. The only known papers are [7, 6] which focus on mining tree patterns with only child edges from a single graph, and [11] which leverages homomorphisms to mine embedded tree patterns from a single tree. To the best of our knowledge, our work is the first one for mining homomorphic tree patterns with descendant edges from a single large tree.

6 Conclusion

In this paper we have addressed the problem of mining maximal frequent homomorphic tree patterns from a single large tree. We have provided a novel definition of maximal

[2] http://monetdb.cwi.nl/xml/

homomorphic patterns which takes into account homomorphisms, pattern specificity and the single tree setting. We have designed an efficient algorithm that discovers all frequent non-redundant maximal homomorphic tree patterns. Our approach employs an incremental stack-based frequency computation method that avoids the costly enumeration of all pattern occurrences required by previous approaches. An originality of our method is that matching information of already computed patterns is materialized as bitmaps, which greatly reduces both memory consumption and computation costs. We have conducted extensive experiments to compare our approach with tree mining algorithms that mine embedded patterns when applied to a large data tree. Our results show that maximal homomorphic patterns are fewer and larger than maximal embedded tree patterns. Further, our algorithm is as fast as the state-of-the art algorithm mining embedded trees from a single tree while outperforming it in terms of memory consumption and scalability.

We are currently working on incorporating user-specified constraints to the proposed approach to enable constraint-based homomorphic pattern mining.

References

1. S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *SIGMOD Conference*, 2001.
2. T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *SDM*, 2002.
3. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *SIGMOD*, 2002.
4. Y. Chi, Y. Xia, Y. Yang, and R. R. Muntz. Mining closed and maximal frequent subtrees from databases of labeled rooted trees. *IEEE Trans. Knowl. Data Eng.*, 17(2), 2005.
5. Y. Chi, Y. Yang, and R. R. Muntz. Canonical forms for labelled trees and their applications in frequent subtree mining. *Knowl. Inf. Syst.*, 8(2), 2005.
6. A. Dries and S. Nijssen. Mining patterns in networks using homomorphism. In *SDM*, 2012.
7. B. Goethals, E. Hoekx, and J. V. den Bussche. Mining tree queries in a graph. In *KDD*, 2005.
8. P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM J. Comput.*, 24(2):340–356, 1995.
9. G. Miklau and D. Suciu. Containment and equivalence for a fragment of xpath. *J. ACM*, 51(1):2–45, 2004.
10. A. Termier, M.-C. Rousset, and M. Sebag. Treefinder: a first step towards xml data mining. In *ICDM*, 2002.
11. X. Wu and D. Theodoratos. Leveraging homomorphisms and bitmaps to enable the mining of embedded patterns from large data trees. In *DASFAA*, 2015.
12. X. Wu, D. Theodoratos, and W. H. Wang. Answering XML queries using materialized views revisited. In *CIKM*, 2009.
13. X. Wu, D. Theodoratos, W. H. Wang, and T. Sellis. Optimizing XML queries: Bitmapped materialized views vs. indexes. *Inf. Syst.*, 38(6):863–884, 2013.
14. M. J. Zaki. Efficiently mining frequent embedded unordered trees. *Fundam. Inform.*, 66(1-2), 2005.
15. M. J. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Trans. Knowl. Data Eng.*, 17(8), 2005.
16. F. Zhu, Q. Qu, D. Lo, X. Yan, J. Han, and P. S. Yu. Mining top-k large structural patterns in a massive network. *PVLDB*, 4(11), 2011.
17. F. Zhu, X. Yan, J. Han, P. S. Yu, and H. Cheng. Mining colossal frequent patterns by core pattern fusion. In *ICDE*, pages 706–715, 2007.

APPENDIX

6.1 Algorithm compHomos.

The algorithm deploys a standard dynamic programming method, computing a Boolean matrix $\mathcal{M}(p, q)$, for $p \in$ nodes(P), $q \in$ nodes(Q), such that $\mathcal{M}(p, q)$ is true if: (1) there exists a homomorphism from the subpattern rooted at p to the subpattern rooted at q (Function $BottomUp$-$Traversal$); and (2) there exists a homomorphism from the prefix path of p to the prefix path of q, where *prefix path* of a node is the subpattern from the pattern root to that node without branches (Function $TopDownTraversal$). The time and memory complexities of Algorithm $compHomos$ are both $O(|P| \times |Q|)$.

Proposition 2. *There exists a homomorphism from pattern P to pattern Q that maps node $p \in P$ to node $q \in Q$ iff entry $\mathcal{M}(p,q)$ is true, where $\mathcal{M}$ is the Boolean matrix computed by Algorithm $compHomos$ on P and Q.*

The proof of Proposition 2 is straightforward by the definition of pattern homomorphisms and the construction process of Boolean matrix $\mathcal{M}$.

Input: two patterns P and Q.
Output: a Boolean matrix $\mathcal{M}$ that encodes all the homomorphisms from P to Q.

1. **if** (BottomUpTraversal(Matrix $\mathcal{C}$)) **then**
2. $\mathcal{M}$:= TopDownTraversal(Matrix $\mathcal{C}$);
3. **else**
4. there is no homomorphism from P to Q;

Function BottomUpTraversal(Matrix $\mathcal{C}$)

1. Initialize a boolean matrices $\mathcal{D}(p, q)$ with $p \in$ nodes(P), $q \in$ nodes(Q);
2. **for** (q of Q's nodes in the bottom-up order) **do**
3. **for** (p of P's nodes in the bottom-up order) **do**
4. $\mathcal{C}(p, q) := (lb(q) = lb(q)) \wedge \bigwedge_{u \in children(p)} (\bigvee_{v \in children(q)} \mathcal{D}(u, v))$;
5. $\mathcal{D}(p, q) := \mathcal{C}(p, q) \vee \bigvee_{v \in children(q)} \mathcal{D}(p, v)$;
6. **return** $\mathcal{D}$(root(p), root(q));

Function TopDownTraversal(Matrix $\mathcal{C}$)

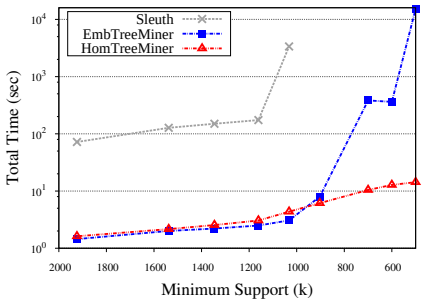
1. Initialize two boolean matrices $\mathcal{P}(p, q)$ and $\mathcal{A}(p, q)$ with $p \in$ nodes(P), $q \in$ nodes(Q);
2. **for** (q of Q's nodes in the top-down order) **do**
3. **for** (p of P's nodes in the top-down order) **do**
4. $\mathcal{P}(p, q) := (\mathcal{C}(p, q)) \wedge \mathcal{A}$(parent of p, parent of q);
5. $\mathcal{A}(p, q) := \mathcal{P}(p, q) \vee \mathcal{A}(p$, parent of q);
6. **return** $\mathcal{P}$;

Fig. 8: Algorithm compHomos

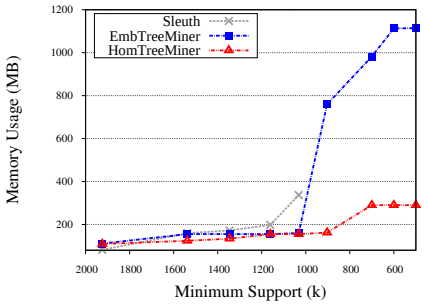
6.2 Experimental Evaluation Plots.

Dataset	Tot. #nodes	#labels	Max/Avg depth	#paths
Treebank	2437666	250	36/8.4	1392231
CSlogs	772188	13355	86/4.4	59691 (#trees)
DBLP	3332130	35	6/3	3000839
XMark	83533	74	12/5.6	60853

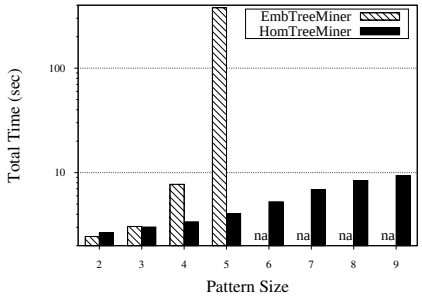
Table 2: Dataset statistics.



(a) Run time vs. support



(b) Memory usage

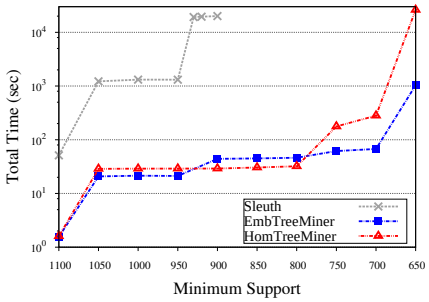


(c) Run time vs. pattern size ($minsup$ = 700)

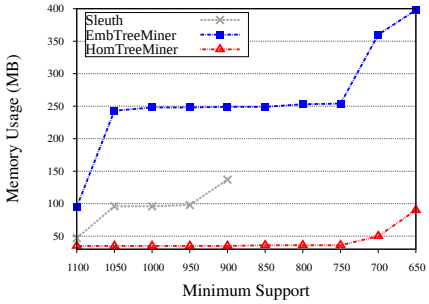
minsup	morphism	tot. candidates	tot. freq. patterns	max. size of freq. patterns
900	Emb	17992	141	5
	Hom	18994	258	8
700	Emb	22836	212	5
	Hom	25238	474	9
500	Emb	38626	396	6
	Hom	42019	751	9

(d) Evaluation statistics

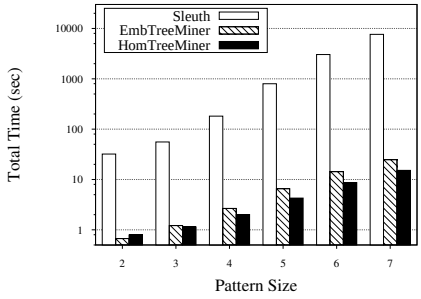
Fig. 9: Performance comparison on CSlogs.



(a) Run time vs. support



(b) Memory usage

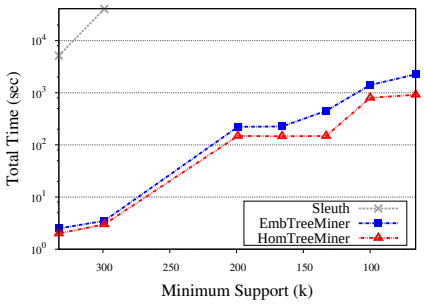


(c) Run time vs. pattern size ($minsup$ = 900)

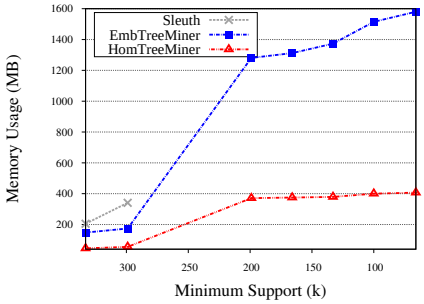
minsup	morphism	tot. candidates	tot. freq. patterns	max. size of freq. patterns
900	Emb	9187	877	10
	Hom	10291	786	10
750	Emb	13359	1259	11
	Hom	38986	2803	11
650	Emb	184757	17117	15
	Hom	3408851	190728	19

(d) Evaluation statistics

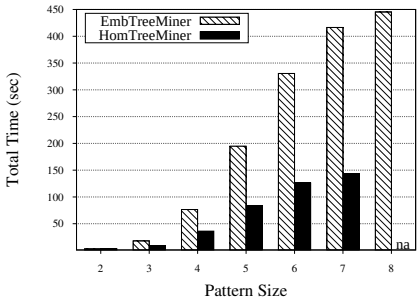
Fig. 10: Performance comparison on XMark.



(a) Run time vs. support



(b) Memory usage



(c) Run time vs. pattern size ($minsup$ = 133k)

Support	morphism	tot. candidates	tot. freq. patterns	Max. size of freq.
133k	Emb	883	103	7
	Hom	711	71	7
100k	Emb	4195	428	10
	Hom	4643	396	9
66k	Emb	5631	572	10
	Hom	5407	460	9

(d) Evaluation statistics

Fig. 11: Performance comparison on DBLP.