Data Sci. Eng. (2016) 1:203–218
DOI 10.1007/s41019-016-0028-7

CrossMark

Homomorphic Pattern Mining from a Single Large Data Tree

Xiaoying Wu[1] · **Dimitri Theodoratos**[2]

Received: 6 July 2016 / Accepted: 19 December 2016 / Published online: 10 January 2017

Abstract Finding interesting tree patterns hidden in large datasets is a 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. Our contribution also includes an optimization technique which can further reduce the search space of homomorphic patterns. We conducted 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 and synthetic 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, 3, 5, 6, 8–11, 14–19, 25–27]. 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 considered. In the literature, two types of tree patterns have been studied: patterns whose edges represent parent-child

The research of Wu was supported by the National Natural Science Foundation of China under Grant Nos. 61202035 and 61272110.

✉ Dimitri Theodoratos
dth@njit.edu

Xiaoying Wu
xiaoying.wu@whu.edu.cn

[1] State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China

[2] New Jersey Institute of Technology, Newark, NJ, USA

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, 5] to considering embeddings for mining patterns with descendant edges (*embedded patterns*) [17, 26, 27]. 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 pattern definition [26]. Nevertheless, embeddings are still 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 Sect. 2. As homomorphisms are more relaxed than embeddings, the mined homomorphic patterns are a superset of the mined embedded patterns.

Figure 1a 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. Figure 1b shows induced mined tree patterns, embedded patterns and non-redundant homomorphic patterns. Figure 1b 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 bioinformatics [29]. As mentioned in [28], 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 [13], while the corresponding problem for an embedding is NP-complete [12].

Mining patterns from a large data tree The type of mined data can be a collection of small trees [2, 5, 17, 26, 27] 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 modeled as a single large tree rooted at a virtual unlabeled node. Existing algorithms for mining embedded patterns from a collection of small trees [26] 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.

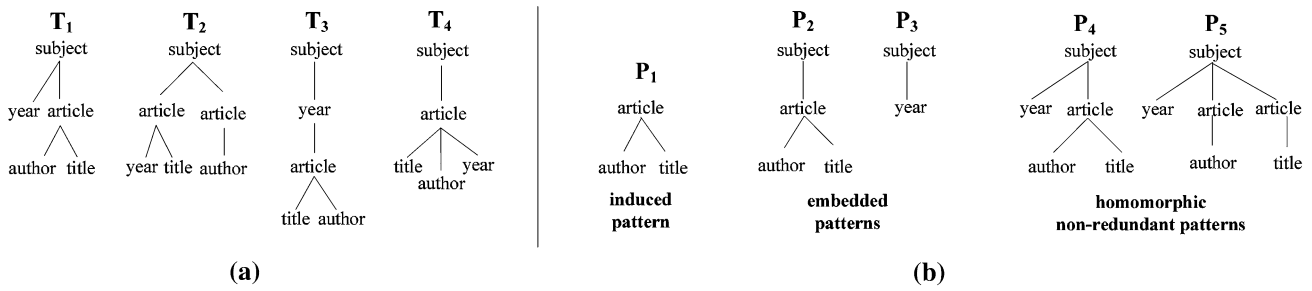


Fig. 1 Different types of mined tree patterns occurring in three of the four data trees. **a** Data trees, **b** mined tree patterns

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 [5]. 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. 1b 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.
- 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.
- We design an efficient algorithm to discover all frequent maximal homomorphic tree patterns. Our algorithm wisely prunes the search space by generating and considering patterns that are maximal and frequent or can contribute to the generation of maximal frequent patterns. It also exploits an optimization technique which relies on pattern ordering to further reduce the space of homomorphic patterns.
- 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.
- 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, (c) our algorithm consumes only a *small fraction of the memory space* and *scales smoothly* when the size of the dataset increases, and(d) the optimization technique substantially improves the time performance of the algorithm.

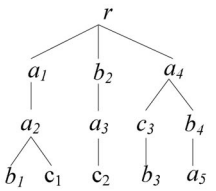
Paper outline The next section introduces various related concepts and formally defines the problem. Section 3 presents our algorithm that discovers all frequent maximal homomorphic tree patterns. Our comparative experimental results are presented and analyzed in Sect. 4. Related work is reviewed in Sect. 5. Section 6 concludes and suggests future work.

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. Figure 2a, 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 whether a tree pattern is included in a tree.



(a)

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

(b)

pattern	occurrences	occur. lists	bitmaps	sup.
P: A (B C)	$\{a_1b_1c_1, a_2b_1c_1, a_4b_3c_3, a_4b_4c_3\}$	$L_A=\{a_1,a_2,a_4\}$ $L_B=\{b_1,b_3,b_4\}$ $L_C=\{c_1,c_3\}$	L_A=11010 L_B=1011 L_C=101	2

(c)

Fig. 2 A tree T, its inverted lists and occurrence info. of pattern P on T. **a** A tree T, **b** inverted lists, **c** occurrence information for pattern P on tree T

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 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 Fig. 2c, the second and third columns give the occurrence relation and the node occurrence lists, respectively, of the pattern P on the tree T of Fig. 2a.

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 Fig. 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 Fig. 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 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 Fig. 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.

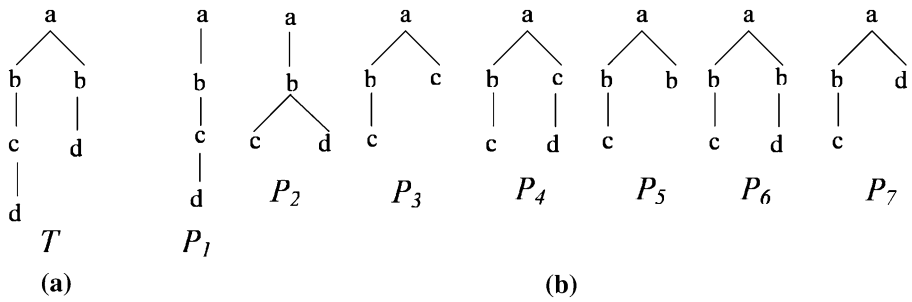


Fig. 3 A data tree and homomorphic patterns. **a** A data tree T, **b** Homomorphic patterns on T

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 homomorphic 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 [26, 27] 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 [7].

3.1.1 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 subpattern 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 *rml*s. 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 (*cousin expansion*), the other is obtained by making y a child node of x in P_x^i (*child expansion*). 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 Fig. 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.

3.1.2 Checking Pattern Redundancy

The pattern generation process may produce candidates which are redundant (defined in Sect. 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 whether 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 called *computeHoms* which, given patterns P and Q, compactly encodes all the homomorphisms from P to Q in polynomial time and space. This algorithm enhances the previous one presented in [13] which checks whether there exists a homomorphism from one tree pattern to another, while achieving the same time and space complexity.

Algorithm computeHoms Algorithm *computeHoms* is presented in Fig. 4. It deploys a standard dynamic programming technique for computing a Boolean matrix $\mathcal{M}(p,$

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

1. Initialize a boolean matrix $\mathcal{C}(p, q)$, $p \in$ nodes(P), $q \in$ nodes(Q);
2. **if** (BottomUpTraversal($\mathcal{C}$)) **then**
3. $\quad\mathcal{M}$:= TopDownTraversal($\mathcal{C}$);
4. **else**
5. $\quad$there is no homomorphism from P to Q;

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

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

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

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

Fig. 4 Algorithm computeHoms

q), $p \in$ 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 *BottomUpTraversal*); 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 path from the pattern root to that node (Function *TopDownTraversal*). Without loss of generality, we assume that both P and Q have a virtual root r. We now describe the algorithm in more detail.

The algorithm first performs a bottom-up traversal of P and Q (Function *BottomUpTraversal*) to compute a Boolean matrix $\mathcal{C}$. Entry $\mathcal{C}(p, q)$ is true if there exists a homomorphism from the subpattern rooted at p to the subpattern rooted at q. To eliminate redundant computations, the bottom-up traversal also computes a second matrix $\mathcal{D}$. Entry $\mathcal{D}(p, q)$ is true if there exists a homomorphism from the subpattern rooted at p to some subpattern of Q whose root is either q or a descendant of q.

If *BottomUpTraversal* returns true, the algorithm proceeds to perform a top-down traversal of P and Q (Function *TopDownTraversal*) to compute a Boolean matrix $\mathcal{P}$. Entry $\mathcal{P}(p, q)$ is true if $\mathcal{C}(p, q)$ (computed by the bottom-up traversal) is true and there exists a homomorphism from the prefix path of p to the prefix path of q. As with the bottom-up traversal, a second matrix $\mathcal{A}$ is computed. Entry $\mathcal{A}(p, q)$ is true if there exists a homomorphism from the prefix path of p to some prefix path of either q or an ancestor of q.

Proposition 1 *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 computeHoms* on *P and Q*.

The proof of Proposition 1 is straightforward by the definition of pattern homomorphisms and the construction process of Boolean matrix $\mathcal{M}$.

We now analyze the complexity of Algorithm *computeHoms*. The entry $\mathcal{D}(u, v)$ of function *BottomUpTraversal* is checked once for every pair of nodes ($u \in children(p)$, $v \in children(q)$) (line 4). The entry $\mathcal{D}(p, v)$ is checked once for every pair of nodes ($p \in P$, $v \in children(q)$) (line 5). Therefore, the total number of times these two entries are checked is no more than $|P| \times |Q|$.

The entry $\mathcal{A}$(parent of p, parent of q) in line 4 and the entry $\mathcal{A}(p$, parent of $q)$ in line 5 of function *TopDownTraversal* are checked once for every pair of nodes ($p \in P$, $q \in Q$). The total number of times these two entries are checked is no more than $|P| \times |Q|$. Therefore, the time and memory complexities of Algorithm *computeHoms* are both $O(|P| \times |Q|)$.

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 Fig. 3b 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 not expandable, every expansion of it is redundant. Therefore, only expandable patterns in a class are considered for expansion.

3.1.3 Expandable Pattern Refining

The number of expandable patterns enumerated by the equivalence class expansion process can still be very large, particularly when the frequent patterns to find have both a high depth and a high branching factor. In order to further reduce the number of generated patterns, we present below a pattern refining method which exploits properties of the equivalence class-based pattern expansion. We observe that the specificity relation $\prec$ induces a linear order on

patterns in a given equivalence class whose rightmost leaf nodes have the same label: for any two patterns P_x^i and P_x^j in the equivalence class $[P]$, $P_x^i \prec P_x^j$ if $i > j$. Clearly, the occurrence set of $P_x^i(x)$ is a subset of the occurrence set of $P_x^j(x)$, for $i > j$.

Let $P_1, P_2, \ldots, P_n$ stand for a sequence of n expandable patterns satisfying the above linear order. Each pattern in the sequence has a rightmost leaf node x. For a pattern P_k, if the occurrence set of $P_k(x)$ is the same as the union of the occurrence sets of $P_i(x)$, $i = 1, \ldots, k-1$, then the set of occurrences of P_k is the same as the union of occurrence sets of P_i's. In this case, it is not useful to further expand P_k, since it is refined by a set of more specific patterns. We call P_k a *refinable* pattern. In Fig. 7, pattern Q_2 is refined by Q_1.

In order to efficiently identify refinable patterns, we keep the patterns P_x^i in each class sorted by the node label x primarily and by the position p (in descending order) secondarily. Figure 6 shows patterns of a class in sorted order. Given a sorted pattern list; the equivalence class expansion process considers ordered pairs of patterns in the class for expansion. This way, the candidate generation process outputs a new class list which is also sorted based on this order, and no explicit sorting is needed.

In the implementation, we scan patterns of a given class in descending order. For each pattern P under consideration, we check whether it has a preceding pattern Q, such that the rightmost leaf nodes of P and Q have the same label and the same occurrence list. If it is the case, P is a refinable pattern and is excluded from further expansion. The process is summarized in Procedure *CheckClassElements* shown in Fig. 5. Our experiments show that the pattern pruning technique can effectively reduce the pattern search space.

3.1.4 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.

Procedure CheckClassElements(Class $[P]$)
1. **for** (each $P_x^i \in [P]$ in descending order) **do**
2. check if P_x^i is expandable; {Ref. Definition 1 and Algorithm $computeHoms$ of Fig. 4}
3. **if** (P_x^i is not expandable and contains a redundant node not on the rightmost path) **then**
4. remove P_x^i from $[P]$;
5. check if P_x^i is refinable; {Ref. Section 3.1.3}
6. **if** (P_x^i is refinable) **then**
7. remove P_x^i from $[P]$;

Fig. 5 Procedure CheckClassElements

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 (Sect. 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* [4]), 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 tree (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

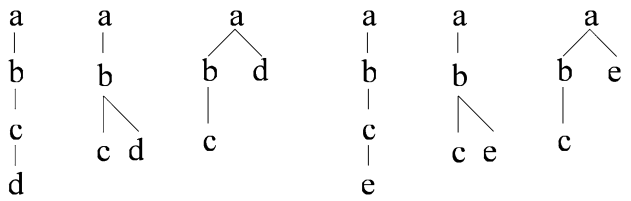


Fig. 6 Sorted patterns in class [*a* / *b* / *c*]

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 [12]. As a consequence, a state-of-the-art unordered embedded pattern mining algorithm *Sleuth* [26] 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 2 *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 2, 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 2, 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 [4]. In contrast, the state-of-the-art methods for mining embedded patterns [26, 27] 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 bit 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 2c 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 [20, 21, 23, 24] for materializing tree pattern views and for efficiently answering queries using materialized views.

Example 1 Figure 7 shows an example of the incremental method for computing the support of Q_1 and Q_2, the two outcomes of $P_1 \otimes P_2$ on the data tree T of Fig. 2a. We assume *minsup* is one. Each node of the patterns P_1 and P_2 is associated with its occurrence list together with the corresponding bitmap vector. To compute Q_1 and Q_2, the bitmaps of P_1 and P_2 are ANDed and the resulting bitmaps are attached to nodes of Q_1 and Q_2. These bitmaps are used as input to compute Q_1 and Q_2 using *TwigStack*. The bitmap output associated with each pattern node indicates the occurrence list of that node on T. Note that pattern Q_2 is refined by Q_1 and thus will not be further expanded.

3.3 The Tree Pattern Mining Algorithm

We present now our homomorphic tree pattern mining algorithm called *HomTreeMiner* (Fig. 8). 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

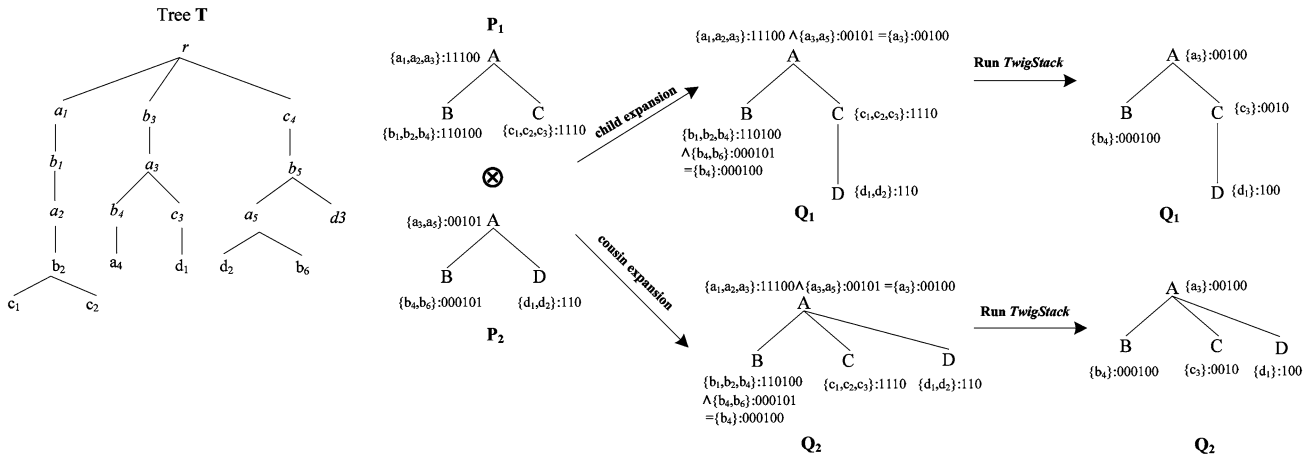


Fig. 7 An example of incremental support computation for the outcomes of $P_1 \otimes P_2$ on tree T

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. Q := the child expansion outcome of $P_x^i \otimes P_y^j$;
6. add Q to $[P_x^i]$ if Q is frequent;
7. Q := the cousin expansion outcome of $P_x^i \otimes P_y^j$;
8. add Q to $[P_x^i]$ if Q is frequent;
9. CheckClassElements($[P_x^i]$); {Ref. Fig. 5}
10. add P_x^i to $\mathcal{M}$ if none of the elements of $[P_x^i]$ is in canonical form;
11. $MineHomPatterns([P_x^i], \mathcal{M})$;

Fig. 8 Homomorphic tree pattern mining algorithm

$lb(Y)$, 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. Then, it computes the support of the child expansion and the cousin expansion outcomes in that order and adds them to $[P_x^i]$ if they are frequent (Lines 1–8). Once all P_y^j have been processed, Procedure *CheckClassElements* of Fig. 5 is invoked on class $[P_x^i]$ (Line 9). Subsequently, the algorithm checks whether P_x^i is a locally maximal pattern. If so, P_x^i is added to the maximal pattern set $\mathcal{M}$ (Line 10). Then, the new class $[P_x^i]$ is recursively explored in a depth-first manner (Line 11). 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 Sect. 3.1 is run to identify maximal patterns among the locally maximal ones and the results are returned to the user (Lines 5–6).

Before expanding a class $[P]$, we make sure that P is expandable and is in canonical form (line 2 in *MineHomPatterns*). Our approach is independent of any particular canonical form; it can work with any systematic way of choosing a representative from isomorphic representations of the given pattern, such as those presented in [7, 26]. Efficient methods for checking canonicity can also be drawn from [7, 26].

Complexity The total cost for generating a new class $[P_x^i]$ is $O(n^2 \times |T| \times |P|)$, where n is the number of elements of $[P]$. In terms of memory consumption, observe that the algorithm only needs to load in memory the classes along a path in a depth-first traversal of the search space. In fact, it only needs to store in memory occurrence lists for two classes at a time: the current class $[P]$ and a new class $[P_x^i]$. Since occurrence lists of each pattern in a class are materialized as bitmaps, the memory footprint of the algorithm is very small. This is verified by our experimental results presented in Sect. 4.

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 evaluate the

Table 1 Dataset statistics

Dataset	Tot. #nodes	#labels	Max/avg depth	#paths
Treebank	2,437,666	250	36/8.4	1,392,231
XMark	83,533	74	12/5.6	60,853
CSlogs	772,188	13,355	86/4.4	59,691 (#trees)

effect of the pattern refining technique described in Sect. 3.1.3, we consider also a basic version of *HomTreeMiner* that does not employ that optimization in its mining process. That basic version was introduced in [22] and is called *HomTMBasic* in the following paragraphs.

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 8 GB memory running JVM 1.7.0 on Windows 7 Professional. The Java virtual machine memory size was set to the default 4 GB.

Datasets We have ran experiments on three real and benchmark datasets with different structural properties. Their main characteristics are summarized in Table 1.

Treebank[1] is a real XML dataset derived from computation linguistics. It models the syntactic structure of English text and provides a hierarchical representation of the sentences in the text by breaking them into syntactic units based on part of speech. The dataset is deep and comprises highly recursive and irregular structures.

XMark[2] is an XML benchmark dataset generated using the data generator with *factor* = 0.05. It is deep and has many regular structural patterns. It includes very few recursive elements.

CSlogs[3] is a real dataset and is composed of users' access trees to the CS department Web site at RPI. The dataset contains 59,691 trees that cover a total of 13,355 unique web pages. The average size of each tree is 12.94.

4.1 Algorithm Performance

We compare the performance of *HomTreeMiner* with two unordered embedded tree mining algorithms *Sleuth* [26] and *EmbTreeMiner* [19]. *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 [12]. As our experiments show, the cost of this post-processing step is in general not significant compared to the frequent pattern mining cost.

To allow *Sleuth*—which is slower—to 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.

Candidate pattern generated Figs. 9c, 10c and 11c compare the total candidates generated by *sleuth*, *EmbTreeMiner*, *HomTMBasic* and *HomTreeMiner*, respectively, under different support thresholds on the Treebank, XMark and CSlogs datasets.

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. On Treebank, for instance, *HomTreeMiner* computes 17 times more candidates than *EmbTreeMiner* at *minsup* = 30 k. Since Treebank contains many deep, highly recursive paths, the search space of homomorphic patterns becomes substantially large at low support levels. Like Treebank, XMark has many deep paths, and therefore, the search space of homomorphic patterns becomes large at low support levels. For example, on XMark at *minsup* = 700, *HomTreeMiner* generates about 2.23 times more candidates than *EmbTreeMiner*. The number of candidates generated by *HomTreeMiner* and *EmbTreeMiner* is comparable on CSlogs. The difference in the number of candidates generated by *sleuth* and *EmbTreeMiner* is not noticeable on all the testing cases.

We notice that *HomTMBasic* generates substantially more candidates than *HomTreeMiner* for low support levels. For instance, on XMark at *minsup* = 650, *HomTMBasic* generates about 9 times more candidates than *HomTreeMiner*. On CSlogs at *minsup* = 250, *HomTMBasic* generates about 5 times more candidates than *HomTreeMiner*. This indicates that the pattern refining technique enables *HomTreeMiner* to reduce substantially the search space when it is applicable.

Execution time We measure the total elapsed time for producing maximal frequent patterns at different support

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

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

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

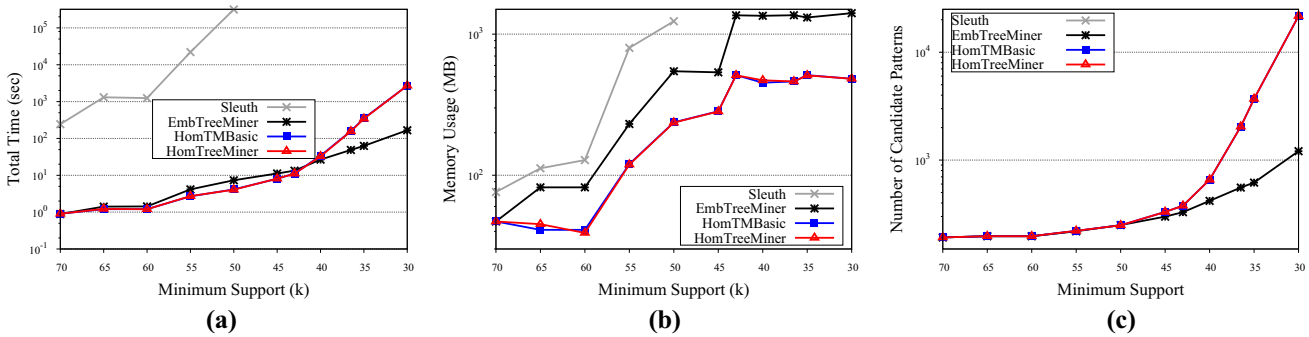


Fig. 9 Performance comparison on a fraction of treebank. **a** Run time versus support. **b** Memory usage. **c** Candidate patterns

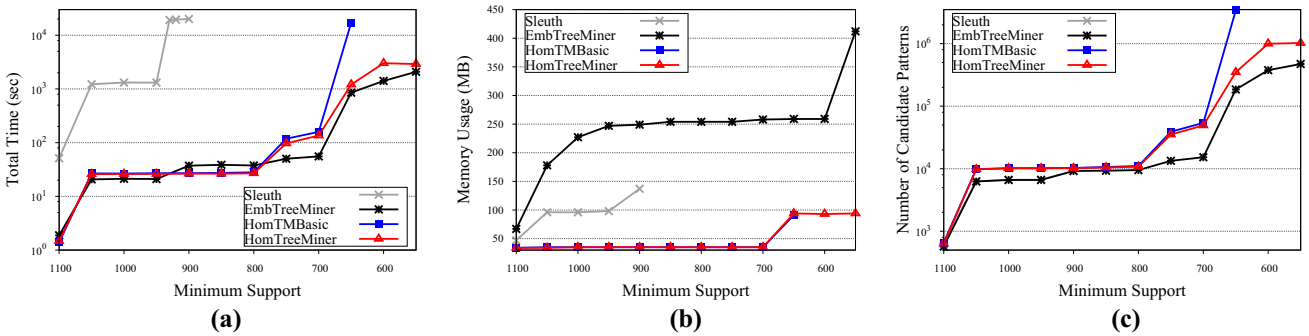


Fig. 10 Performance comparison on XMark. **a** Run time versus support. **b** Memory usage. **c** Candidate patterns

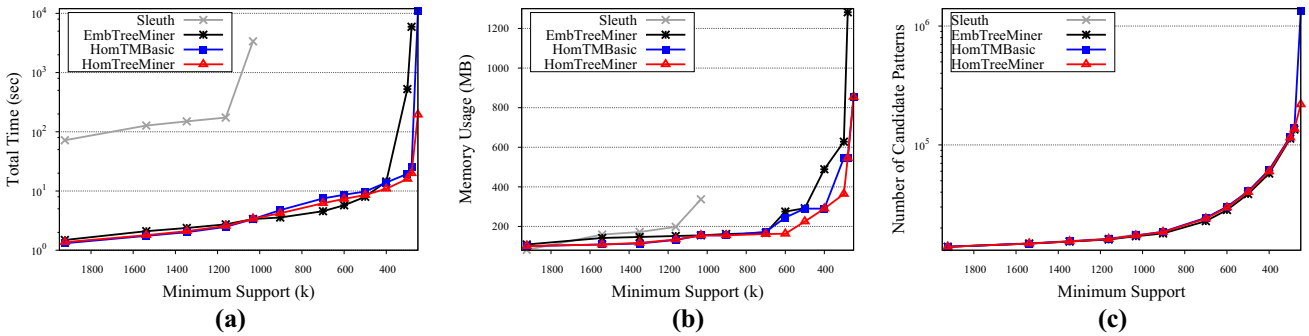


Fig. 11 Performance comparison on CSlogs. **a** Run time versus support. **b** Memory usage. **c** Candidate patterns

thresholds. The total time involves the time to generate candidate patterns, compute pattern support and check maximality of frequent patterns.

Figures 9a, 10a and 11a compare the total elapsed time of the four algorithms under different support thresholds on the Treebank, XMark and CSlogs datasets. Due to prohibitively long times, we stopped testing *Sleuth* when support levels are below certain values on each dataset.

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 (Sect. 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.

The large number of candidate homomorphic patterns can negatively affect the time performance of *HomTreeMiner* at low support levels. For instance, on Treebank, *HomTreeMiner* shows similar or better performance than *EmbTreeMiner* when support levels are above 40 K and both generate a similar number of candidates. When *minsup* decreases below 40 K, the execution time of *HomTreeMiner* increases noticeably faster than that of *EmbTreeMiner* due to the substantially larger number of

candidates evaluated by *HomTreeMiner*. At *minsup* = 30k, in order to evaluate 17 times more candidates, *HomTreeMiner* runs about 15 times slower than *EmbTreeMiner*.

However, even though the number of (candidate and frequent) homomorphic patterns is always larger than the number of embedded patterns, this difference is not so pronounced in shallower datasets like CSlogs. As we can see from Fig. 11a, *HomTreeMiner* can largely outperform *EmbTreeMiner* at low support levels. This is due to its efficient computation of pattern support which does not require the enumeration of pattern occurrences and the embedding checking as is the case with *EmbTreeMiner* [19].

From the results, we observe that *HomTreeMiner* can largely outperform *HomTMBasic*, when it is able to substantially reduce the search space with the refinement technique. For instance, on XMark at *minsup* = 650, *HomTreeMiner* runs more than 13 times faster than *HomTMBasic*.

Memory usage We measured the memory footprint of the four algorithms with varying support thresholds. The results are shown in Figs. 9b, 10b and 11b. We can see that *HomTreeMiner* always has the best memory performance. It consumes substantially less memory than both *Sleuth* and *EmbTreeMiner* in all the test cases. 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. The memory performance of *HomTMBasic* is similar to that of *HomTreeMiner*. The results indicate that the memory performance of a mining algorithm is mainly determined by its pattern support computation.

4.2 Algorithm Scalability

In our final experiment, we studied the scalability of the three algorithms *EmbTreeMiner*, *HomTMBasic* and *HomTreeMiner* as we increase the input data size. We omit the comparison with *sleuth*, since *sleuth* was unable to finish within a reasonable time even on the smallest size of input.

For Treebank, we generated ten fragments of increasing size and fixed *minsup* at 4.5%. For XMark, we generated 10 XMark trees by setting *factor* = 0.01, 0.02, ..., 0.1 and fixed *minsup* at 1%. For CSlogs, we generated 7 datasets of different sizes (from 40 k trees and up to 100 k) by randomly choosing trees from the original CSlogs. We fixed *minsup* at 400.

The results show that *HomTreeMiner* has the best time performance on both XMark and CSlogs (Fig. 12b, c); it runs slightly slower than *EmbTreeMiner* on Treebank (Fig. 12a). The reason is that, on both XMark and CSlogs, the number of candidates evaluated by *HomTreeMiner* is similar to that by *EmbTreeMiner*, whereas on Treebank, it needs to evaluate 56% more candidates on average. On CSlogs, the growth of the running time of *EmbTreeMiner* becomes much sharper with datasets containing 80 k trees and up. *EmbTreeMiner* is unable to finish within 5 hours on CSlogs containing 90 k trees and up. *HomTMBasic* has similar time performance with *HomTreeMiner* on both Treebank and XMark. However, on CSlogs of size 90 k and 100 k, *HomTreeMiner* outperforms *HomTMBasic* by a factor of more than 2.5. The reason is that, in these two cases, *HomTMBasic* has to evaluate about 47 k more candidates in total and generates twice as many frequent patterns on average than *HomTreeMiner*.

Figure 13a–c show that *HomTreeMiner* always has the smallest memory footprint. The growth of its memory consumption is much slower than that of *EmbTreeMiner*.

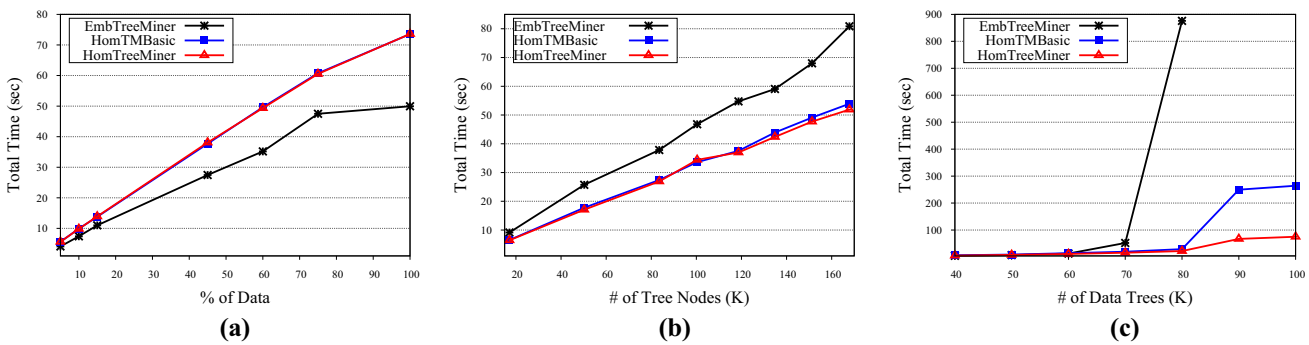


Fig. 12 Run time scalability comparison on the three datasets with increasing size. **a** Treebank (*minsup* = 4.5%). **b** XMark (*minsup* = 1%). **c** CSlogs (*minsup* = 400)

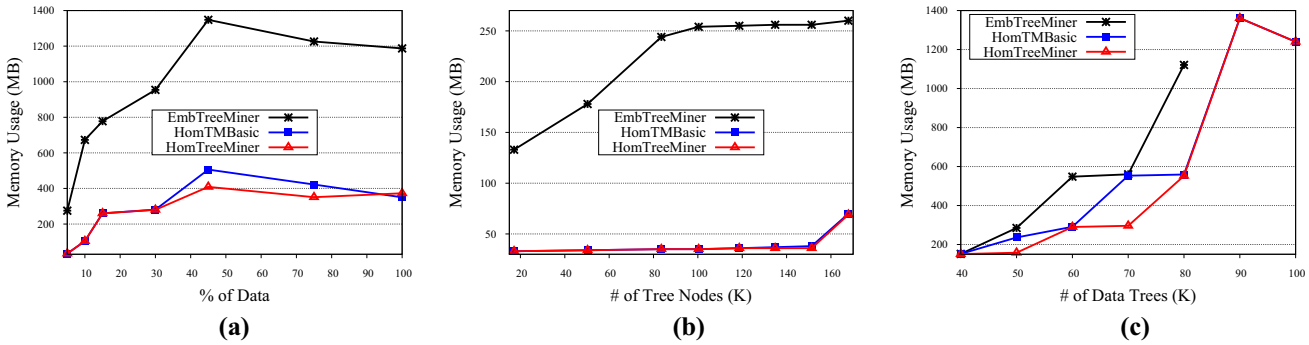


Fig. 13 Memory usage scalability comparison on the three datasets with increasing size. **a** Treebank (*minsup* = 4.5%). **b** XMark (*minsup* = 1%). **c** CSlogs (*minsup* = 400)

Table 2 Statistics for maximal frequent patterns mined from the three datasets

Dataset	Morphism	# freq. patterns	# loc.max patterns	# max. non. red.patterns	%max. over freq. patterns	Average #nodes	Average height	Average fanout	maximum #nodes	#common max.patterns
Treebank	Emb	78	n/a	2 (8)	2.6	0.63	0.375	0.25	3	1
(minsup = 35k)	Hom	521	158	9	1.7	5	2.11	2.11	8	
Treebank	Emb	175	n/a	13 (32)	7.4	1.47	0.66	0.78	5	5
(minsup = 30k)	Hom	2937	915	35	1.2	6.14	2.23	2.57	9	
XMark	Emb	934	n/a	14 (19)	1.5	2.63	1.05	1.58	5	6
(minsup = 800)	Hom	853	26	15	1.76	4.67	1.93	2.6	10	
XMark	Emb	43,441	n/a	27 (54)	0.06	3.33	1	2.09	15	14
(minsup = 550)	Hom	56,160	302	35	0.06	8.74	2.29	5.71	15	
CSlogs	Emb	638	n/a	133 (164)	20.8	2	0.896	1.1	5	119
(minsup = 400)	Hom	816	307	152	18.6	2.53	1.11	1.41	6	
CSlogs	Emb	2192	n/a	250 (375)	11.4	1.68	0.728	0.95	6	192
(minsup = 280)	Hom	1625	676	312	19.2	2.8	1.22	1.57	6	

4.3 Comparison of Mined Maximal Homomorphic and Embedded Patterns

We computed different statistics on frequent and maximal frequent patterns mined by *HomTreeMiner* and *EmbTreeMiner* from the three datasets varying the support; the results are summarized in Table 2. For the comparison, we considered only maximal embedded patterns that contain no redundant nodes. We show the total number of maximal embedded patterns in parenthesis in Column 5. 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 Table 2, the maximum size of frequent homomorphic patterns and the maximum size and average number of nodes, height and fanout of maximum frequent homomorphic patterns is never smaller (substantially larger in many cases) than that of the embedded patterns for the same support level.

Second, the number of homomorphic and embedded frequent patterns is substantially reduced if only maximal patterns are selected (Column 6 of Table 2). 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 2).

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 14 shows, for each of the three datasets, examples of embedded maximal patterns each from the same set in the partition and the corresponding maximal homomorphic pattern. Therefore, for a number of applications, maximal homomorphic patterns can offer more information in a more compact way.

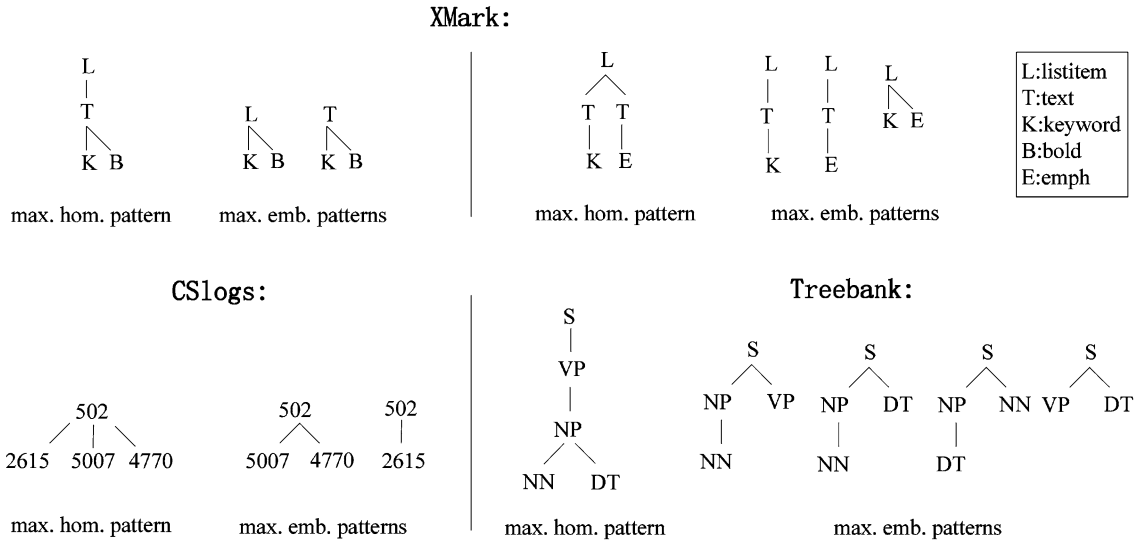


Fig. 14 Examples of maximal patterns mined from the three datasets

5 Related Work

We now discuss how our work relates to the 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 [2, 3, 5, 6, 8–11, 14–19, 25–27], only few mine unordered embedded patterns [9, 17, 26].

Mining unordered embedded patterns TreeFinder [17] 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.

WTIMiner [9] transfers the frequent tree pattern mining to itemset mining. It first finds all the frequent itemsets, and then for each itemset found, it scans the database to count all the corresponding tree patterns. Although *WTIMiner* is complete, it is inefficient since the structural information is lost while mining for frequent itemsets. Further, the overhead for processing false positives may potentially reduce the performance.

Sleuth [26] extends the ordered embedded pattern mining algorithm *TreeMiner* [27]. 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.

Mining maximal and closed induced patterns There exist algorithms [5, 18, 25] which focus on mining closed and maximal (induced) patterns. A frequent pattern P is closed if none of P's proper superpatterns has the same support as that P has; P is maximal if none of P's proper superpatterns is frequent. The number of both maximal and closed patterns is usually much smaller, yet represents the same information as that of *all* frequent patterns. We below briefly mention about these algorithms.

CMTreeMiner [5] mines both closed and maximal frequent patterns from a set of small trees. Their method relies on a concept called *blanket*. The blanket of a pattern provides the set of immediate super patterns that are frequent. By comparing the occurrences of a given pattern with the occurrences of its blanket patterns, the algorithm determines whether the original pattern is closed or not. It uses pruning and heuristic techniques to reduce the search space. *CMTreeMiner* is the first algorithm which directly mines closed and maximal patterns without first generating all frequent patterns. However, it mines only for induced patterns; extending it to embedded patterns is not trivial.

PathJoin [25] finds maximal unordered induced patterns from a set of small trees. *PathJoin* assumes that no two siblings in data trees have the same label. It first discovers the set of maximal frequent paths and then it finds the tree patterns by joining the paths. After obtaining all frequent patterns, *PathJoin* keeps only maximal patterns by using a post-processing pruning, which eliminates those that are not maximal. Such a strategy will suffer from a significant overhead if the number of false positive paths is very high.

DryadeParent [18] mines closed induced patterns from a set of small trees. Observing that the performances of

existing algorithms are dramatically affected by the branching factor of the tree patterns to find, *DryadeParent* makes the assumption that no two siblings in the data trees can have the same label (similar to *PathJoin*). The method first computes a set of tiles, which are closed frequent patterns of depth 1. Then, it develops a hooking strategy that reconstructs the closed frequent patterns from these tiles. Similar to *PathJoin*, *DryadeParent* is designed based on the assumption that no two siblings in the data trees can have the same label. While this simplifies the problem, it limits the usage of the method in real applications.

The work on mining tree patterns in a single large tree or graph setting has so far been very limited. The only known papers are [8, 10, 11] which focus on mining tree patterns with only child edges from a single graph and [19] 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 efficiently mining (maximal) homomorphic tree patterns with descendant edges from a single large tree.

A preliminary version of algorithm *HomTreeMiner* was presented in [22]. The algorithm described in the present paper extends the previous version with an optimization technique which exploits the specificity relation to prune the space of candidate homomorphic patterns. The performance of the new version of *HomTreeMiner* is compared with that of its old version in the experimental section.

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 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. An optimization technique further prunes the search space of candidate patterns. 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.

Several applications are interested in extracting not all the frequent patterns, but only those that comply with a number of restrictions. We are currently working on incorporating user-specified constraints to the proposed approach to enable constraint-based homomorphic pattern mining.

References

1. Amer-Yahia S, Cho S, Lakshmanan LVS, and Srivastava D (2001) Minimization of tree pattern queries. In: SIGMOD, pp 497–508
2. Asai T, Abe K, Kawasoe S, Arimura H, Sakamoto H, and Arikawa S (2002) Efficient substructure discovery from large semi-structured data. In: SDM, pp 158–174
3. Asai T, Arimura H, Uno T, Nakano S-I (2003) Discovering frequent substructures in large unordered trees. In: Discovery, Science, pp 47–61
4. Bruno N, Koudas N, and Srivastava D (2002) Holistic twig joins: optimal XML pattern matching. In: SIGMOD, pp 310–321
5. Chi Y, Xia Y, Yang Y, Muntz RR (2005) Mining closed and maximal frequent subtrees from databases of labeled rooted trees. IEEE Trans Knowl Data Eng 17(2):190–202
6. Chi Y, Yang Y, and Muntz RR (2004) Hybridtreeminer: an efficient algorithm for mining frequent rooted trees and free trees using canonical form. In: SSDBM, pp 11–20
7. Chi Y, Yang Y, Muntz RR (2005) Canonical forms for labelled trees and their applications in frequent subtree mining. Knowl Inf Syst 8(2):203–234
8. Dries A, Nijssen S (2012) Mining patterns in networks using homomorphism. In: SDM, pp 260–271
9. Feng Z, Hsu W, and Lee M-L (2005) Efficient pattern discovery for semistructured data. In: ICTAI, pp 294–301
10. Goethals B, Hoekx E, and den Bussche JV (2005) Mining tree queries in a graph. In: KDD, pp 61–69
11. Kibriya AM, Ramon J (2013) Nearly exact mining of frequent trees in large networks. Data Min Knowl Discov 27(3):478–504
12. Kilpeläinen P, Mannila H (1995) Ordered and unordered tree inclusion. SIAM J Comput 24(2):340–356
13. Miklau G, Suciu D (2004) Containment and equivalence for a fragment of xpath. J ACM 51(1):2–45
14. Nijssen S, Kok JN (2004) A quickstart in frequent structure mining can make a difference. In: KDD, pp 647–652
15. Tan H, Hadzic F, Dillon TS, Chang E, Feng L (2008) Tree model guided candidate generation for mining frequent subtrees from xml documents. TKDD 2(2):1–43
16. Tatikonda S, Parthasarathy S, Kurç TM (2006) Trips and tides: new algorithms for tree mining. In: CIKM, pp 455–464
17. Termier A, Rousset M-C, Sebag M (2002) Treefinder: a first step towards xml data mining. In: ICDM, pp 450–457

18. Termier A, Rousset M-C, Sebag M, Ohara K, Washio T, Motoda H (2008) Dryadeparent, an efficient and robust closed attribute tree mining algorithm. IEEE Trans Knowl Data Eng 20(3):300–320
19. Wu X, Theodoratos D (2015) Leveraging homomorphisms and bitmaps to enable the mining of embedded patterns from large data trees. In: DASFAA, pp 3–20
20. Wu X, Theodoratos D (2016) Template-based bitmap view selection for optimizing queries over tree data. Int J Coop Inf Syst 25(3):1–28
21. Wu X, Theodoratos D, Kementsietsidis A (2015) Configuring bitmap materialized views for optimizing XML queries. World Wide Web 18(3):607–632
22. Wu X, Theodoratos D, Peng Z (2016) Efficiently mining homomorphic patterns from large data trees. In: DASFAA, pp 180–196
23. Wu X, Theodoratos D, Wang WH (2009) Answering XML queries using materialized views revisited. In: CIKM, pp 475–484
24. Wu X, Theodoratos D, Wang WH, Sellis T (2013) Optimizing XML queries: bitmapped materialized views vs. indexes. Inf Syst 38(6):863–884
25. Xiao Y, Yao J-F, Li Z, Dunham MH (2003) Efficient data mining for maximal frequent subtrees. In: ICDM, pp 379–386
26. Zaki MJ (2005) Efficiently mining frequent embedded unordered trees. Fundam Inform 66(1–2):33–52
27. Zaki MJ (2005) Efficiently mining frequent trees in a forest: algorithms and applications. IEEE Trans Knowl Data Eng 17(8):1021–1035
28. Zhu F, Qu Q, Lo D, Yan X, Han J, Yu PS (2011) Mining top-k large structural patterns in a massive network. PVLDB 4(11):807–818
29. Zhu F, Yan X, Han J, Yu PS, Cheng H (2007) Mining colossal frequent patterns by core pattern fusion. In: ICDE, pp 706–715