Frequent Tree Mining with Selection Constraints

Jeroen De Knijf

Utrecht University, Institute of Information and Computing Sciences
PO Box 80.089, 3508 TB Utrecht
The Netherlands
jknijf@cs.uu.nl

Abstract. Data that can conceptually be viewed as tree structures occurs extensively in domains such as bio-informatics, web logs, XML databases and computer networks. One important problem in mining tree structured data is to find all frequent subtrees. Due to combinatorial explosion, the number of frequent subtrees grows exponentially with the size of the trees. This causes severe problems with the completion time of the mining algorithm and the huge amount of potentially uninteresting patterns. In related areas such as frequent itemset mining and mining sequential patterns, the use of constraints has resulted in a considerable speedup of the mining application. Furthermore, the discovered patterns are focused on the users interest, which reduces the effort for the user to investigate the output of the mining process. In this paper we define selection constraints, both on the structure and on the labels of trees. We show how to efficiently compute all ordered and unordered induced closed subtrees that satisfy the selection constraints. We illustrate the use of these constraints within the application of web log mining. Finally, the effect of applying selection constraints on a synthetic data set and a real data set is evaluated. Our approach leads to a considerable speedup of the mining algorithm.

1 Introduction

Frequent treemining has become an important and popular problem in the field of knowledge discovery and data mining. The main reasons for the increase in interest is the growing amount of semi-structured data (e.g. XML databases) and the urge to exploit and mine these databases. Furthermore, the availability of treemining algorithms such as [2, 13, 16] to exploit these databases, without losing information on the structure of the data, has increased the interest of the research community. Briefly, given a set of tree data, the problem is to find all subtrees that satisfy the minimum support constraint, that is all subtrees must occur in at least $n\%$ of the data records. Applications of frequent treemining include the following:

- Web log mining: frequent access trees from a database of web logs, where each record corresponds to the entire forward access of a user, are explored in [16].
- Classification and clustering: the work presented in [17] uses a frequent treemining algorithm to extract frequent substructures from XML data, and subsequently classifies the data according to its structure.

– Database indexing: in [15] a frequent treemining algorithm is used to extract frequent tree query patterns from a large collection of XML queries. The answers to the frequent tree query patterns are then stored and indexed for faster retrieval.

In this work we introduce selection constraints on the labels and the structure of trees. We present an efficient mining algorithm (SCTreeminer) that combines closed frequent treemining with the exploitation of user-defined selection constraints to prune the search space. Experimental evaluation of SCTreeminer is performed on both a synthetic and a real data set, and we compared the run time for the constraint based algorithm with a post processing approach.

This paper is organized as follows: in the next section we describe related work and the motivation for a constraint based treemining approach. In section 3 we discuss the basic concepts required for frequent treemining. Section 4 describes selection constraints in detail, their properties, and problems related to mining with these constraints. In this section we also describe the SCTreeminer algorithm. In section 5 we experimentally evaluate our algorithm and compare the performance with a post pruning approach. In the last section we draw conclusions and indicate further research directions.

2 Motivation and Related Work

The design of effective algorithms for mining frequent subtrees has been the subject of several studies in recent years, see for example [2, 4, 5, 9, 12, 13, 16]. These algorithms differ in the type of trees handled (free trees, rooted unordered trees, rooted ordered trees) and the kind of tree matching relation used (induced, embedded, incorporated). A drawback of these approaches is the lack of user controlled focus in the pattern mining process, the only control mechanism the user has being the minimum support threshold. This results in:

1. High computational cost. Given a database of trees, the computation cost for treemining algorithms is fixed for a given minimum support threshold. For users that have a clear idea of which kind of patterns are interesting, the mining algorithm spends computation time on patterns that are of no interest to the user.
2. Huge number of potentially useless results. Since the number of frequent subtrees grows exponentially with the size of the tree, the user has to put a lot of effort into finding interesting results.

There are two basic approaches to address these problems: condensed representations and user-defined constraints. Chi et al. [6] introduce an algorithm to find all closed and maximal frequent subtrees. This approach results in a considerable speedup and smaller output size. For example, on the synthetic data set described in section 5 which consists of $10,000$ records the number of frequent (induced) subtrees is more then $118,000,000$, while the number of closed subtrees is slightly above $164,000$, using a minimum support of 1%. The computation time is reduced by a similar factor as the output size.

Another approach to tackle these problems is to provide users with a constraint specification language, in which they can express a "family" of patterns in which they are interested [7, 8]. If these constraints can be pushed deep into the mining process—as

opposed to post-pruning—this results in a considerable speedup of the mining algorithm in addition to a reduction of the output.

In this paper we combine both approaches. Mining closed trees does not give the user more control of the mining process, but it compresses the output such that all frequent subtrees are derivable from it. On the other hand the output of constraint query patterns can be further compressed by presenting only the closed patterns that satisfy the constraints without any loss of information.

3 Preliminaries

In this section we provide the basic concepts and notation that is used in the remainder of this paper. A labeled rooted tree $T = \{V, E, \Sigma, L, v_0\}$ consists of a vertex set V, an edge set E, an alphabet Σ for vertex labels and a labeling function $L : V \rightarrow \Sigma$ that assigns labels to vertices. The special node v_0 is called the root. If $(u, v) \in E$ then u is the parent of v and v is the child of u. For a node v, any node u on the path from the root node to v is called an ancestor of v. If u is an ancestor of v then v is called a descendant of u. If in addition the tree is also ordered there is a binary relation '$\leq$' $\subset V^2$ that represents an ordering among siblings. The size of a tree is defined as the number of vertices; we refer to a tree of size k as a k-tree.

Given two labeled rooted trees T_1 and T_2 we call T_2 an *induced* subtree of T_1 and T_1 an *induced* supertree of T_2, denoted by $T_2 \preceq T_1$, if there exists an injective matching function Φ of V_{T_2} into V_{T_1} satisfying the following conditions for any $v, v_1, v_2 \in V_{T_2}$:

- Φ preserves the parent relation: $(v_1, v_2) \in E_{T_2}$ iff $(\Phi(v_1), \Phi(v_2)) \in E_{T_1}$.
- Φ preserves the labels: $L_{T_2}(v) = L_{T_1}(\Phi(v))$.

If in addition T_1 and T_2 are ordered, the mapping Φ should also preserve the order among the siblings; that is, if $v_1 \leq_{T_2} v_2$ then $\Phi(v_1) \leq_{T_1} \Phi(v_2)$.

In this work we refer to induced subtrees simply as subtrees. Let D denote a database where each transaction $d \in D$ is a labeled rooted tree. For a given pattern tree t, which is also a labeled rooted tree, we say t occurs in a transaction d if t is a subtree of d. Let $\phi_d(t)$ denote the distinct occurrences of t in d, i.e. $\phi_d(t)$ is the set $\{\{\Phi^1(v_1), \ldots, \Phi^1(v_k)\}, \ldots, \{\Phi^n(v_1), \ldots, \Phi^n(v_k\}\}$ where $(v_1, \ldots, v_k) \in V_t$ and $|t| = k$; with $\Phi^1, \ldots, \Phi^n$ the distinct matching functions from t into d. With $\phi(t)$ we denote the union of all occurrences in the database of t: $\phi(t) = \cup_{d \in D} \phi_d(t)$. Let $\psi_d(t) = 1$ if $|\phi_d(t)| > 0$ and 0 otherwise. The support of a pattern tree t in the database D is then defined as $\psi(t) = \sum_{d \in D} \psi_d(t)$, that is the number of records in which it occurs one or more times. A pattern tree t is called frequent if $\psi(t)$ is greater than or equal to a user defined minimum support (minsup) value. The goal of frequent treemining is to find all frequent trees in a given database. Frequent treemining algorithms make use of the apriori property, which states that any subtree of a frequent tree is also frequent and any supertree of an infrequent tree is also infrequent.

To mine all frequent trees that satisfy the selection constraints we use some properties of closed trees: a tree t is closed if all supertrees of t have lower support. The blanket of t, denoted by B_t, is defined as the set of frequent supertrees of t that can be constructed from t by adding one vertex. Using the definition of blanket, we can

rewrite the definition of closed trees in terms of the blanket of a tree: a tree t is closed iff $\forall t' \in B_t : \psi(t') < \psi(t)$. For two trees t_1 and t_2, we call t_1 occurrence matched with t_2, if there is an extension t_3 of t_1, with one or more nodes of t_2, such that t_3 is a supertree of t_2 and for every occurrence of t_1 there is a distinct occurrence of t_3. More formally, t_1 is occurrence matched with t_2, with $t_2 \not\preceq t_1$, if there exists a tree t_3 with $t_1 \preceq t_3 \wedge t_2 \preceq t_3$ and for every transaction $d \in D$ in which t_1 occurs we have that

$$\forall X \in \phi_d(t_1) \exists Y \in \phi_d(t_2) : X \cup Y \in \phi_d(t_3).$$

If t_1 is occurrence matched with t_2 then t_1 is not closed, because we can extend t_1 with the vertices of t_2 that are not part of t_1; the support of this new tree is equal to the support of t_1. Let $\phi(t|t')$ denote the occurrences of t that have a matching occurrence for t', i.e.

$$\phi(t|t') = \cup_{d \in D} \phi'_d(t) \text{ where } \phi'_d(t) =$$
$$\cup \{X | X \in \phi_d(t) \wedge \exists Y \in \phi_d(t') \exists t'' : X \cup Y \in \phi(t'') \wedge t \preceq t'' \wedge t' \preceq t'' \wedge t' \not\preceq t\}.$$

Let $\psi(t|t') = \sum_{d \in D} \psi_d(t|t')$ denote the support of t restricted to t' where $\psi_d(t|t') = 1$ if $|\phi_d(t|t')| > 0$ and 0 otherwise. A tree t is closed with respect to t' iff $\forall t'' \in B_t : \psi(t''|t') < \psi(t|t')$. Our definition of occurrence matching is an extension of the definition given in [6] where occurrence matching is only defined between a tree t and the trees in B_t.

To enumerate all frequent closed trees we use CMtreeminer, described in [6]. Here we give a brief summary of the CMtreeminer algorithm. Enumerating all frequent subtrees is done by using the rightmost extension techniques described in [2]: a $(k-1)$-tree is expanded to a k-tree by adding a new node *only* to a node on the rightmost branch of the $(k-1)$-tree. The rightmost branch of a tree is the unique path from the root to the rightmost leaf. Note that for each k-tree its parent is uniquely defined by removing the rightmost vertex. This procedure of extending pattern trees ensures that each pattern tree is counted exactly once. To compute the closed trees some extra pruning techniques are used: left-blanket pruning and right-blanket pruning. If a tree t is occurrence matched with a tree $t' \in B_t$, we distinguish two possible locations at which the additional vertex (v) should be added to extend t into t': **i)** v is added to the rightmost path of t, **ii)** v is added elsewhere. In the first case (right-blanket pruning) we should not extend t by adding vertices to any proper ancestor of v. In the second case (left-blanket pruning) t nor any extension of t can be closed, hence t can be pruned. When the trees are unordered rooted trees, multiple rooted unordered trees are equivalent to one rooted ordered tree. To solve this a canonical representation is needed, so in [5, 6, 9] fast canonical representations are developed. When the unordered trees are in canonical form, they can be further processed as ordered trees. All techniques used in CMtreeminer also apply to unordered trees that are in DFCF (depth first canonical form) as defined in [6]. In the rest of this work we assume that an unordered tree is in DFCF. Although CMtreeminer is also able to mine all maximal trees, we did not use this in our algorithm, because the computation of all maximal trees as done in CMtreeminer does not speed up the mining process. In fact, when mining only for maximal trees, all closed trees are a byproduct of the algorithm. Furthermore, maximal frequent trees have the disadvantage that with the extraction of all frequent trees from the set of maximal trees the exact support is lost; only a lower bound of the support can be determined.

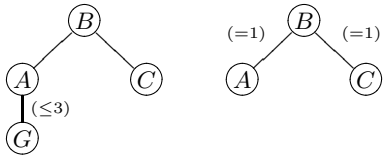


Fig. 1. Example of selection constraints.

4 Selection Constraints

Consider the web log of an online store. Suppose a data analyst is interested in customers who were in doubt between buying two different computers sold by the store. More specifically, the analyst is interested in users who visited both web pages that describe computers of type A and web pages that describe computers of type B. Suppose the analyst wants to know which points were discriminating for customers into making their choice (described by pages that can be reached from the pages about the specific computer types) and what other products these customers were interested in. With the selection constraints we define below, the data analyst can specify such constraints in the treemining algorithm. To specify the constraints in the constraint language, the node that represents the start page of the department that sells computer supplies is added as a root. To further specify the aforementioned constraints, the node that represents the top of the web pages describing computer type A, is added as the first child of the root. The second child of the root is the top of the web pages that describe computer type B. In general, selection constraints are defined as follows:

1. A labeled root ordered tree that consists of one or more nodes where the distance between an ancestor and a descendant can be constrained ($\leq, =, \geq$).
2. If S_1 and S_2 are two selection constraints then $S_1 \vee S_2$ is also a selection constraint.

A tree T satisfies the constraints if T is an *embedded* supertree—with respect to the distances specified between ancestors and descendants—of at least one of the terms in the disjunction. More formally, a tree T satisfies a selection constraint if there exists a term C of the disjunction such that: there exists an injective function $\Phi : V_C \rightarrow V_T$ satisfying the following conditions for any $v_1, v_2 \in V_C$:

- Φ preserves the labels: $L_C(v_1) = L_T(\Phi(v_1))$.
- Φ preserves the order among the siblings: if $v_1 \leq_C v_2$ then $\Phi(v_1) \leq_T \Phi(v_2)$.
- Φ preserves the ancestor-descendants relation and respects the constrained distance between them : if $(v_1, v_2) \in E_C$ and $dist(v_1, v_2) \odot n$ then $\Phi(v_1)$ is an ancestor of $\Phi(v_2)$ in T and the path length between $\phi(v_1)$ and $\phi(v_2) \odot n$, with $\odot \in \{\leq, =, \geq\}$.

Two examples of selection constraints are given in figure 1. On the left, the distance between the root node and its descendants is unspecified, so they may occur at any distance. The node labeled G must however occur in maximum distance of 3 from node A. On the right, we insist the nodes labeled A and C are children of the root node. As a third example, in figure 2 the selection constraint C_t does not specify a distance between the root and its children, so any tree with label B that has descendants A and C that are not descendants of each other satisfies the constraint.

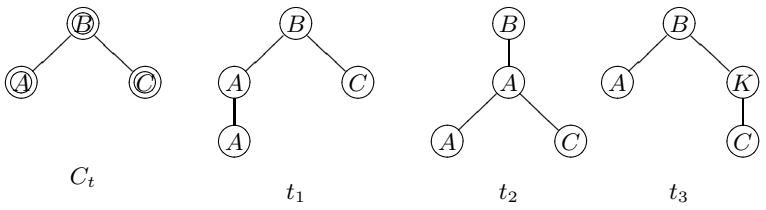


Fig. 2. A selection constraint (C_t) and three trees (t_1, t_2, t_3) that satisfy the constraint, since all the nodes of C_t occur in-order. t_1 is however not a base tree because it is not minimal.

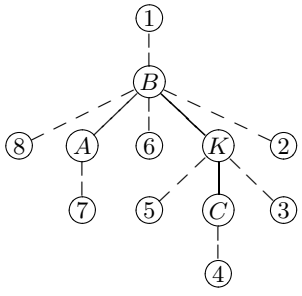
The selection constraints are inspired on succinct constraints used in frequent itemset mining [8] and regular expression constraints in sequence mining [7]. For example, suppose the selection constraints consist of two trees, the first tree with a root labeled A and a descendant node labeled B and the second tree with a root labeled B and a descendant node labeled A. These constraints, which are in fact all possible rooted trees with the nodes A and B, are similar to the succinct constraints for which the items A and B must be in the itemset. In sequence mining, with an alphabet $A - Z$ the regular expression constraint $A(C - Z)^*B|B(C - Z)^*A$ is similar as well. However, because trees consist of more structures than sequences or sets, it is natural to incorporate more structure in constraints for trees.

4.1 Optimization Using Selection Constraints

In order to compute all frequent trees that satisfy the constraints, we first find all *minimal* trees that satisfy the constraints (*minimal* in the sense that no subtree satisfies the constraints). This selection is done in the first scan through the database; due to space limitations we will not elaborate on this further. These minimal trees are henceforth called base trees. Any frequent tree satisfying the constraints must be a supertree of *at least* one of the frequent base trees.

The idea is to extend each frequent base tree—similar to the frequent one patterns in other treemining algorithms—in such a way that all closed frequent trees that satisfy the constraints are enumerated. With the standard rightmost extension technique, we can however not generate all supertrees if the enumeration is started from the base trees. To illustrate this point, in figure 3 all possible locations to add new nodes to the base tree t_3 (the rightmost tree in figure 2) are shown. For example, if a node labeled H occurs frequently on location 6 of t_3, the frequent tree obtained by extending t_3 with H at location 6 should clearly be part of the output. However, this tree can not be enumerated by using the rightmost extension technique, because the position of the new node is not on the rightmost path. Besides the rightmost extension (extension positions 2, 3 and 4 in figure 3), base trees should therefore be extended in the parent direction (marked with label 1 in figure 3) and depending on the shape of the base tree also on the leftmost path (extension positions 7 and 8) and in the area between the leftmost child and the rightmost child of the root (extension positions 5 and 6).

These extension problems can be solved by using the properties of closed trees. Consider an arbitrary base tree bt_0. Rather than starting the enumeration from bt_0 itself, we start the enumeration from its root t_1. Starting from t_1 we are able to enumerate



t_3

Fig. 3. Locations for an additional vertex to be added to the base tree t_3.

all closed frequent supertrees of bt_0 with root t_1 that satisfy the constraints using *only the rightmost extension* technique as follows: let t_k be a tree constructed from t_1 by applying the rightmost extension technique one or more times. If tree t_k can be extended to t_{k+1} such that $\phi(t_{k+1}|bt_0) = \phi(t_k|bt_0)$, then t_k is not closed with respect to bt_0, since then $\psi(t_{k+1}|bt_0) = \psi(t_k|bt_0)$. If there is a node in bt_0 that is not contained in t_k, then t_k cannot be closed with respect to bt_0, because we can extend t_k to t_{k+1}, with the node from bt_0 not contained in t_k, such that $\phi(t_{k+1}|bt_0) = \phi(t_k|bt_0)$. With this approach we condense a base tree to one node (its root), because we have enough information at this vertex together with the occurrences of the base tree.

For example consider the base tree t_3 in figure 3. We start by taking the root node of t_3 (B); we extend B with the node A—call this tree t_k. Because there is a supertree t_{k+1} of t_k—the extension of t_k with node K – for which $\phi(t_k|t_3) = \phi(t_{k+1}|t_3)$, t_k is not closed with respect to t_3; hence we keep extending t_k. Furthermore, suppose there is a frequent extension with a node labeled H, that is a sibling of A. This tree t_{k+1} is not closed because there is still a supertree that is occurrence matched with respect to t_3; hence we keep extending t_{k+1}. With this approach the positions $2-8$ of t_3 as shown in figure 3 are considered for extension and all valid results are supertrees of t_3.

With the previous observations we have solved the enumeration problem from the base pattern to extensions on the leftmost path and the area between the first child of the root and the last child. For extension of the base tree (bt_0 with root node t_1) in the parent direction, we compute the parents of t_1; call this set P_{t_1}. If there is a node in P_{t_1}, t_c, that is occurrence matched with t_1, we can condense the base tree to the new node t_c. As a result we extend bt_0 with t_c as the new root of the base tree; the computation is then repeated for the extended base tree and its new root. If t_1 is not occurrence matched with any node in P_{t_1} we add t_1 to the frequent one-patterns and for each frequent node in P_{t_1} we construct a new base tree bt_{0_i}, that is, bt_0 extended with the frequent node as new root node. For each extended base tree of bt_0, to which we further refer as $bt_{0_1}, \ldots, bt_{0_n}$, this procedure is repeated. In figure 4 we give our selection constraint algorithm in pseudo-code.

With the algorithms previously specified, we introduce duplicates. For example, when root nodes with the same label of different base patterns have the same occur-

Algorithm SCTreeminer

$S \leftarrow$ all frequent base trees $1 \ldots n$
$i \leftarrow 1$
$out \leftarrow \varnothing$
while $(S \neq \varnothing)$
  $bt_i \leftarrow$ a tree from S
  $S \leftarrow S \setminus \{bt_i\}$
  $v_i \leftarrow v_0$ /* the root node of bt_i */
  $\phi(v_i) \leftarrow \phi(v_0|bt_i)$
  $F_1 \leftarrow$ MineParent(v_i, bt_i)
  $out \leftarrow out \cup$ ComputeClosedtrees(F_1, bt_i)
  $i \leftarrow i + 1$
return out

Function MineParent (set S, current base tree bt_i)

$out \leftarrow \varnothing$
while $(S \neq \varnothing)$
  $t \leftarrow$ a tree from S
  $S \leftarrow S \setminus t$
  $P_t \leftarrow$ the frequent parents of t
  if $\exists t' \in P_t : t$ is occurrence matched with t'
    then delete t
    else mark $\phi(t)$ with a tag from bt_i
      $out \leftarrow out \cup \{t\}$
  $S \leftarrow S \cup P_t$
  return out

Fig. 4. Pseudo-code of the SCTreeminer algorithm.

rences in the database, the enumeration of the subtree starting at these nodes will be initiated twice, leading to duplicate trees in the output. Also the MineParent algorithm causes duplicates: in figure 5 the data tree d_1 contains two base patterns that match the constraint C_t in figure 2; call these base patterns bt_0 (occurring in the left subtree of the root node) and bt_1 (occurring in the right subtree). When the parent sets of bt_0 and bt_1 are computed the base pattern with root node E is present twice. These base patterns $E - D - bt_0$ and $E - bt_1$ need not be the same, because the occurrence list of both patterns might be different; but when we mine all closed trees starting from the two 1-trees with root E and occurrences $\phi(E - D|bt_0)$ and $\phi(E|bt_1)$ respectively we can get d_1 twice in the output.

To overcome this problem, the algorithm requires extra rules when we start the enumeration from a base pattern as well as when extending the current pattern at the root node of another base pattern. Note that the root nodes of a base pattern are all "frequent one patterns" previously discovered by the MineParent algorithm.

The basic idea is to define an ordering on the initially discovered base patterns. We use the support of the base patterns as ordering criterion, but the order is in fact arbitrary. Suppose we have n base patterns with ordering: $bt_1 < bt_2 < \ldots < bt_{n-1} < bt_n$. Notice that for each base pattern bt_i, a set $bt_{i_1}, \ldots, bt_{i_n}$ is associated that are the extensions (generated by the MineParent algorithm) of bt_i; the order of these extended base trees equals the order of the associated base tree. In the rest of this paper, we use bt_i as an abbreviation of the base tree bt_i or an extended base tree from the associated set of bt_i.

When *extending* bt_i the mining algorithm can enumerate all base patterns that have a higher order, i.e. $bt_{i+1}, bt_{i+2}, \ldots, bt_n$. In the other case, when we are extending the current base pattern bt_j with a root node from another base pattern bt_i, with $bt_i < bt_j$ we have to determine if the extension t_k of the current base pattern is occurrence matched with bt_i. As long as it is not occurrence matched we should extend t_k, because t_k may have children that are part of bt_i or children that can not be reached starting from bt_i. Note that in this case one can not enumerate the whole base tree bt_i from t_k. When *starting* from bt_i, suppose the root node of bt_i is also the root node of other base

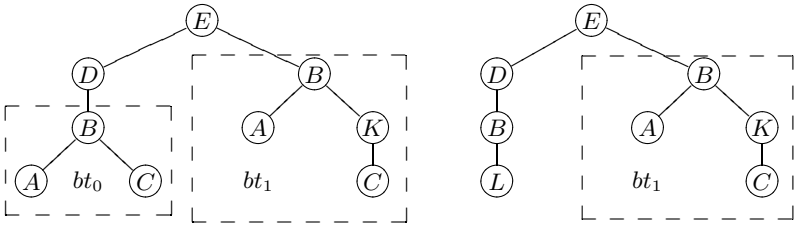


Fig. 5. Two example data trees d_1 (left) and d_2 (right).

patterns. If all these base patterns have a higher order, we proceed with bt_i. Otherwise, we proceed with bt_i until it is occurrence matched with one of the lower ordered base patterns.

To compute $\phi(t_k|bt_i)$ in an efficient way, we mark in the database all root labels of the base trees and the extended base trees with its corresponding order; this is done in the same sweep trough the database as the computation of the frequent one patterns. With this approach we can determine $\phi(t_k|bt_i)$ without any overhead. For the example with data tree d_1 in figure 5, if we have computed $E - D - bt_0$, we extend it with the nodes of the right subtree of d_1, because $bt_0 < bt_1$. Hence, if d_1 is frequent, it is part of the result. For another example, again consider the two data trees in figure 5. Suppose the extension of bt_1 with nodes E, D and B is frequent, call this tree t_k. Because some occurrences of B are part of the base pattern bt_0 and $bt_1 > bt_0$, we determine if t_k and bt_0 are occurrence matched. Since this is not the case we extend t_k to t_{k+1} by adding a new label A. Since t_{k+1} and bt_0 are occurrence matched, we prune t_{k+1}. Another extension possibility is adding node L to t_k. In this case t_{k+1} is a valid $k+1$ candidate tree and if this tree is frequent it should be be further extended. In figure 6 the pseudo-code is shown to compute closed trees; including our duplicate elimination technique.

Function ComputeClosedtrees (set of k-trees F_k, current base pattern bt_m)
  $out \leftarrow \varnothing$
  while $F_k \neq \varnothing$
    $t \leftarrow$ a tree from F_k
    $F_k \leftarrow F_k \setminus t$
    if $\exists bt_n : bt_n < bt_m$ **and** t is occurrence matched with bt_n
      then delete t
      else $F_{k+1} \leftarrow F_{k+1} \cup$ all valid and frequent extensions of t according to CMtreeminer
    if t is closed
      then $out \leftarrow out \cup \{t\}$
  $k \leftarrow k + 1$
  return out

Fig. 6. Pseudo-code of CMtreeminer with the duplicate elimination algorithm.

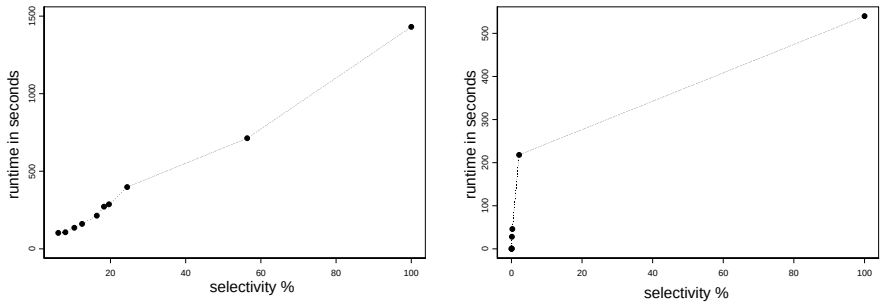


Fig. 7. The running time for different levels of selectivity. The rightmost point in both plots (where the selectivity equals 100) is the case where there are no constraints specified, i.e. the closed treemining algorithm. *Left*: the results for the data set T1D with minimum support value of 1%. *Right*: the results for RPI data set with minimum support of 0.042%.

5 Experimental Results

We implemented all algorithms in C++ and studied the performance of the applications extensively. All experiments were run on a 2.8GHz PC with 500 MB of RAM, running Red Hat Linux. We performed experiments on two data sets: a synthetic data set T1D and the RPI CS web log data. Both the synthetic data set generator and the web log data were kindly provided by Mohamed Zaki and are described in [16]. We briefly describe the synthetic data set generator. First a master tree is generated with the following parameters: the number of distinct node labels $N = 50$, the total number of nodes in the master tree $M = 100,000$, the maximum fanout of a node $F = 10$ and the maximal depth of the tree $D = 15$. From this master tree the data set is created by selecting a number $T = 10,000$ of subtrees from the master tree. The average number of nodes in the trees equals 230. The real data set RPI CS consists of logs of the RPI computer science website. After processing, RPI CS consists of $13,361$ unique web-pages and $59,691$ user browsing subtrees.

The goal of the experiments is to examine the use of selection constraints in frequent treemining. Because our SCTreeminer algorithm is an extension of the closed treeminer algorithm, we chose data sets that require some running time of the closed treeminer algorithm, and compared it to the run time of our SCTreeminer algorithm. The run time of the closed treemining algorithm is a lower bound for the time needed to compute all frequent subtrees that satisfy selection constraints with a post processing approach. We compared the closed treemining algorithm with SCTreeminer for the two data sets with several sets of constraints, with different selectivity. A selectivity of $x\%$ means that $x\%$ of the closed frequent trees satisfies the constraints. Another option was to compare a frequent treemining algorithm with post processing selection constraints and our SCTreeminer algorithm, with a post processing step to extract all frequent trees from the closed trees. But since on the data set we used, the closed treemining algorithm needed about one quarter of an hour to complete, while a standard frequent treemining algorithm needed more than 3 days, the comparison would not have been very illustrative.

In figure 7 the results of the experiments are shown. For the synthetic data set $T1D$ the run time appears to increase linearly with selectivity of the output. The maximum speedup achieved is about 14. In the real data set RPI, five out of nine runs with selection constraints used less than 0.5 seconds of CPU time. The number of closed frequent trees that satisfy the constraints for these runs was between 28 and 65, a maximum selectivity of 0.06%. Note that we did not get a selectivity higher than 3% for any run with selection constraints. An explanation for this is the high number of labels compared with the number of closed frequent trees, i.e. many node labels occur in only very few trees. Note that the computation time for the run with a selectivity of 3% is fairly high (218 seconds). This is because when mining this data set with such a low minimum support (0.042%) value, the size distribution of the frequent closed trees has a long right tail. This tail is likely to be caused by webcrawlers which contrary to regular users visit a large number of webpages. The run with a selectivity of 3% contained many of these trees with large size: 58% of the trees were of size greater than 18, while in the run with no constraints only 3% of the trees were of size greater than 18.

6 Conclusion

In this paper we proposed the use of selection constraints for frequent treemining. Selection constraints allow the user to give a partial specification of patterns of interest. We have shown in our experiments that the use of selection constraints leads to a reduction of the search space (and hence computation time), and may yield a considerable reduction of patterns that are of no interest to the user. The reduction of the search space is achieved by pre-selecting records of the database that satisfy the constraints.

Further research can be divided into two directions. The first direction is to change the subtree inclusion relation, i.e. the extension of the selection constraints treemining algorithm in such a way that also embedded and incorporated trees are covered. Another interesting possibility in this direction is to define similar constraints and develop similar algorithms for graphs as well. The second research direction is to extend the constraint specification language such that also constraints on attributes of nodes can be incorporated. We plan to further investigate these topics in the near future.

Acknowledgment

We would like to to thank professor Mohamed Zaki for providing the data set and the synthetic data generator. This work is supported by the Netherlands Organisation for Scientific Research (NWO) under grant no. 612.066.304.

References

1. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 1994.

2. T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In R. L. Grossman, J. Han, V. Kumar, H. Mannila, and R. Motwani, editors, *Proceedings of the Second SIAM International Conference on Data Mining*, 2002.
3. R. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proceedings of the 15th International Conference on Data Engineering*, page 188, 1999.
4. B. Bringmann. Matching in frequent tree discovery. In *ICDM '04: Proceedings of the Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 335–338, 2004.
5. Y. Chi, Y. Yang, and R. R. Muntz. Indexing and mining free trees. In *ICDM '03: Proceedings of the Third IEEE International Conference on Data Mining*, page 509, 2003.
6. Y. Chi, Y. Yang, Y. Xia, and R. R. Muntz. Cmtreeminer: Mining both closed and maximal frequent subtrees. In *The Eighth Pacific Asia Conference on Knowledge Discovery and Data Mining (PAKDD'04)*, May 2004.
7. M. N. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 223–234, 1999.
8. R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In A. T. Laura and M. Haas, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 13–24, 1998.
9. S. Nijssen and J.N. Kok. Efficient discovery of frequent unordered trees. In *In Proceedings of the first International Workshop on Mining Graphs, Trees and Sequences (MGTS2003)*, pages 55–64, 2003.
10. N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT '99: Proceeding of the 7th International Conference on Database Theory*, pages 398–416, 1999.
11. J. Pei and J. Han. Can we push more constraints into frequent pattern mining? In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2000.
12. U. Rückert and S. Kramer. Frequent free tree discovery in graph data. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 564–570, 2004.
13. K. Wang and H. Liu. Discovering structural association of semistructured data. *Knowledge and Data Engineering*, 12(2):353–371, 2000.
14. X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, 2003.
15. L. H. Yang, M. Lee, W. Hsu, and S. Acharya. Mining frequent quer patterns from xml queries. In *Eighth International Conference on Database Systems for Advanced Applications (DASFAA '03)*, pages 355–362, 2003.
16. M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proc. of the Int'l Conf. on Knowledge Discovery and Data Mining*, pages 71–80, 2002.
17. M. J. Zaki and C. C. Aggarwal. Xrules: an effective structural classifier for xml data. In L. Getoor, T. E. Senator, P. Domingos, and C. Faloutsos, editors, *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 316–325, 2003.