Frequent Free Tree Discovery in Graph Data

Ulrich Rückert and Stefan Kramer
Technische Universität München
Institut für Informatik/I12
Boltzmannstr. 3
D-85748 Garching b. München, Germany

Abstract

In recent years, researchers in graph mining have been exploring linear paths as well as subgraphs as pattern languages. In this paper, we are investigating the middle ground between these two extremes: mining free (that is, unrooted) trees in graph data. The motivation for this is the need to upgrade linear path patterns, while avoiding complexity issues with subgraph patterns. Starting from such complexity considerations, we are defining free trees and their canonical form, before we present FreeTreeMiner, an algorithm making efficient use of this canonical form during search. Experiments with two datasets from the National Cancer Institute's Developmental Therapeutics Program (DTP), anti-HIV and anti-cancer screening data, are reported.

1 Introduction

Recent years have seen tremendous progress in the area of mining graph, sequence and tree data for patterns of interest [3, 7, 20]. While the advent of XML and the need for mining semi-structured data has sparked a lot of interest in finding frequent rooted trees in forests [16, 1, 21], other approaches have focused on mining for patterns in databases containing general graphs [12, 2, 18, 20, 7, 14]. The work described in this paper falls into this latter category: we are mining for patterns in general graph databases.

In this area, MolFea [14] was the first domain-dependent inductive database and integrated levelwise search and version spaces in a unified framework. MolFea uses patterns (linear molecular fragments) for mining graph data (molecular structures). The system has been shown to scale up nicely [11] and the patterns used as features in predictive data mining exhibit an excellent predictivity [10, 9]. While MolFea is restricted to linear fragments as patterns, other research groups have started working on the other side of the spectrum and investigated the use of subgraph patterns in graph mining. Today, there is a range of different algorithms capable of finding frequent subgraphs in graphs (such as Inokuchi, Washio and Motoda's AGM [6, 8, 7]). Motivated by complexity considerations, we are exploring the middle ground between MolFea and AGM in this paper. We address the task of finding frequent free (that is, unrooted) trees in graphs. Free trees can be seen as fragments with "recursive side chains", which are a generalization of linear path patterns, or as acyclic connected graphs, which are a specialization of general graphs.

When mining in graph databases, we have to face, among other things, three tasks:

- *canonization*: Is a pattern a syntactic variant of another? This task is important because we do not want to evaluate duplicates of the same pattern on the database.
- *subsumption*: Does a pattern subsume another pattern? This is important for pruning infrequent pattern candidates and computing borders (such as Bd^+, or G and S in version space terminology).
- *coverage*: Does a pattern cover a transaction in the database? This is the basic test for determining the frequency of a pattern.

When working on graph data and using subgraphs as patterns, all three tasks are computationally very expensive: subsumption and coverage tests boil down to subgraph isomorphism, which is NP-complete[1], and canonization is equivalent to solving the graph isomorphism problem, for which no polynomial algorithm is known, but which has not been proven to be NP-complete either. It is assumed to lie between P and NP. Since we are using free trees, we are computationally in a better position: While we do not know

[1]Subgraph isomorphism for induced subgraphs is in NP, but not NP-complete; the precise complexity class is not yet known. Current algorithms for induced subgraph isomorphism feature exponential time complexity.

about any polynomial algorithm for the coverage test (implying that the generation of unnecessary candidate patterns should be avoided as much as possible), subsumption is an instance of the subtree isomorphism problem, which can be solved in O($\frac{k^{1.5}}{\log k}n$) time, where k and n are the sizes of the subtree and the tree to be searched [15]. Canonization can be processed in polynomial time, as outlined in section 2. Since the coverage test is the computationally most demanding part, it makes sense to perform canonization and subsumption tests in order to avoid unnecessary coverage tests on the database. Based on such complexity considerations, we are introducing free tree patterns into graph mining as a compromise between the more expressive, but computationally hard general graph patterns and the fast, but less expressive linear path patterns.

Since the subsumption test can be done in polynomial time, it is easy to compute a boundary set representation of the solution space, which is an NP-complete problem for full-fledged subgraph patterns. One minor contribution of the paper is therefore the computation of border representations. In this sense, this paper represents a first step towards constraint-based graph mining.

This paper is organized as follows: next, we define free trees and a canonical form for them. In Section 3, an algorithm for mining free trees is presented. The algorithm has partly been inspired by the work on gSpan [20] in that growing and checking frequent free trees is combined into one procedure. In Section 3, we also sketch how a border representation can be computed for free trees. Section 4 is presenting experimental results with the new approach. Section 5 discusses how our approach is related to existing work and Section 6 concludes the paper.

2 Frequent Free Tree Mining

First of all, let us introduce the setting more formally. Given a set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a set of edges $E \subseteq V \times V$, $G =_{def} (V, E)$ constitutes a *graph*. Given a set of vertex labels L_V and a set of edge labels L_E, a *labeled graph* is a graph that has a vertex label associated with each node v, denoted by $label(v) \in L_V$ and an edge label associated with each edge, denoted by $label(e) \in L_E$. For our purposes, the class of *connected acyclic labeled graphs* is of special interest. Figure 1 (a) and (b) are examples of graphs in this class. Because of its acyclicity, each connected acyclic labeled graph has at least one node which is connected to the rest of the graph by only one edge, that is, a *leaf*. In such a graph one can label the leaves with zero and the other nodes recursively with the minimal label of its neighbors plus one (figure 1). This yields an unordered, unrooted tree-like structure, a so-called *free tree*. It is a well-known fact that every free tree has at most two nodes which minimize the maximal distance to all other nodes in the tree,

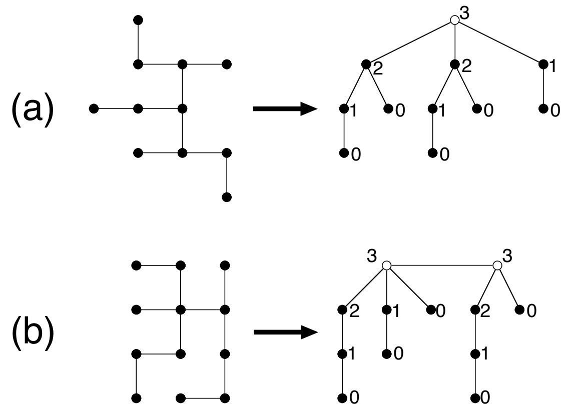

Figure 1. Illustration of center(s) of free trees. If the leaf nodes of a free tree (i.e. a connected acyclic graph) are labeled with zero and the other nodes are labeled with the minimal label of its neighbors plus one, the nodes with the largest labels are the *centers* of the graph. Every free tree has either one (a) or two (b) centers.

the so-called *centers*. In figure 1 the centers of the two free trees (a) and (b) are marked as empty nodes (◯).

There are various ways to store a free tree on a computer. For efficient processing, it is a good idea to use a representation that ensures two equivalent free trees are always encoded as the same pattern. Such a representation is called a *canonical form*. There is a broad range of possible canonical forms that can be used for free trees. In the following we will present a particular canonical form, whose properties allow for an efficient use in the mining algorithm in section 3. The canonical form of a free tree is derived in a two step algorithm. First, we identify the canonical center and use it as a root, therefore building a rooted unordered tree from a free tree. In the second step we order the nodes in the rooted tree to get an ordered rooted tree, that is, the canonical form.

An important building block is a suitable order $\preceq_t$ on ordered rooted trees. In a rooted tree, each edge connects a child node to its parent node. Thus, we can regard each node and the corresponding edge to its parent as a unit. This assigns exactly one label pair $label(e, n) \in L_E \times L_V$ to each node, except for the root node. For the sake of simplicity, we introduce an "empty" label l_ϵ in L_V and L_E. This label is assigned as an replacement for the non-existing parent edge of the root, so that each node n has associated exactly one label $label(e, n) \in L_E \times L_V$. Now, assume we have an arbitrary order $\preceq_{EV}$ on $L_E \times L_V$. In the following we use this order to design a lexicographic order on rooted trees.

First we have to introduce some concepts. The *depth* of a node in a tree is the distance to the root node. The set of nodes with the same depth is a *level*. If the tree is ordered, each level can be represented by a sequence of nodes, the *level-sequence*. The *levelwise traversal* of a tree t enumerates the nodes in a tree from the top level to the bottom level of the trees, processing each level-sequence from left to right. If we have two nodes n_1 and n_2 in two ordered trees t_1 and t_2, the *structural completion* of the node n_1 with regard to n_2 is the addition of empty-labeled children to n_1 until n_1 has the same number of children than n_2. For example, if n_1 has three children and n_2 has one child, the structural completion adds two children, both labeled (l_ϵ, l_ϵ). If n_1 has less or equally many children than n_2, the structural completion does not change n_1. The structural completion of the tree t_1 with regard to t_2 is the recursive application of the structural completion for nodes on the nodes of t_1. Intuitively, the structural completion for trees adds empty nodes to t_1 so that each node in t_1 has its unique counterpart in t_2. Finally, the *levelwise traversal order* compares the structural completion of t_1 with regard to t_2 with the structural completion of t_2 with regard to t_1 using the levelwise traversal. Algorithm 1 sketches the idea. A nice property of this order is, that the order of trees with the same size does not change, if we add new nodes at the bottom level nodes. We will make use of this property in the mining algorithm in section 3. One can also use the levelwise traversal order to order the subtrees in a tree, thus transforming an unordered tree in an ordered tree.

Using this order we can now describe the two steps to build the canonical form of a free tree t. First of all, we identify the centers of t. If there is only one center, we have a unique root. If there are two centers, we remove the edge between the two centers, thus creating two subtrees of the same height. We order the two subtrees and compare them according to the levelwise traversal order. The root of the smaller of the two subtrees is used as the root of the whole tree. Now that we have a rooted tree, we can simply order the nodes in the tree to get the unique canonical representation. Calculating the canonical form can be done in polynomial time.

Algorithm 1 An algorithm calculating the levelwise traversal order on two ordered trees t_1 and t_2.

```
procedure IsSmallerOrEqual($t_1$, $t_2$)
  Append the (root($t_1$), root($t_2$)) to $queue$
  while $queue$ not empty do
    Remove ($n_1$, $n_2$) from the front of $queue$
    if label($n_1$) $\preceq_{EV}$ label($n_2$) then
      return true
    else if label($n_2$) $\preceq_{EV}$ label($n_1$) then
      return false
    end if
    for $i$ = 1 to max(| children($n_1$) | , | children($n_2$) | )
    do
      $c_i \leftarrow \begin{cases} i\text{th child of } n_1 \text{ if } i \leq |\text{children}(n_1)| \\ \text{empty node otherwise} \end{cases}$
      $d_i \leftarrow \begin{cases} i\text{th child of } n_2 \text{ if } i \leq |\text{children}(n_2)| \\ \text{empty node otherwise} \end{cases}$
      Insert ($c_i$, $d_i$) at the back of $queue$
    end for
  end while
  return true
end procedure
```

3 The FreeTreeMiner Algorithm

The performance of a frequent pattern mining algorithm usually depends on three critical issues. First of all, for sufficiently complex pattern languages one pattern can be represented in many different forms. Since the algorithms have to deal with internal representations rather than the patterns per se, they need some way to detect equivalent representations and to avoid processing the same pattern twice. Second, searching for all possible patterns is inefficient. If the pattern language can be ordered by a “more-general-than” relation, a pattern that has an infrequent successor according to this partial order must be infrequent as well. This allows for efficient search space pruning and most algorithms make use of such a partial order to boost the performance. Third, scanning the database for transactions that match the pattern is a very expensive operation. Most algorithms therefore try to minimize the number of database scans. In the following we will describe how the FreeTreeMiner algorithm deals with those three points. The main search loop can be implemented either as a Apriori-like levelwise search or as a depth-first search. In this paper we present only the depth-first version.

To overcome the first critical issue, FreeTreeMiner stores free trees only in the canonical form described in the previous section. Unlike all frequent-graph-mining algorithms, the check on whether or not a new candidate tree is in canonical form can be made in polynomial time. When some new nodes are appended to the leafs of two subtrees in a tree, the levelwise traversal order changes the order of those subtrees only, if they are equal. Thus, an extension of a free tree is in most cases still in canonical form and the test can be made in logarithmic time. Only if the original tree contains symmetric subtrees, the check on whether or not a new candidate tree is in canonical form is more expensive.

For the second issue, FreeTreeMiner searches from general patterns (i.e small trees) to specific patterns (large trees) according to the “is-subtree-of” relation. After each database scan, the algorithm discards infrequent candidate trees and generates new candidates only by extending an

Algorithm 2 The database scan part of FreeTreeMiner

```
procedure DatabaseScan($t$, $d$)
    $ext \leftarrow$ empty extension table
    for graph number $g$ in the database $d$ do
        for all occurrences $o$ of $t$ in graph number $g$ do
            for all extension point $p$ of $t$ do
                for all extensions $e$ to $p$ at $o$ do
                    if extension $(p : e)$ is not present in $ext$ then
                        Insert a row for $(p : e)$ into $ext$ with empty support set
                    end if
                    Add $g$ to the support set in row $(p : e)$
                end for
            end for
        end for
    end for
    return ext
end procedure
```

evidently frequent tree. The extension happens as follows: for every free tree, FreeTreeMiner keeps track of the leaves that have the largest distance to the root. Those leaves are marked as *extension points* in the internal representation. If a free tree is found to be frequent, the algorithm generates new candidates by appending new nodes to the extension points. A detailed explanation of the extension process is provided below.

For the third issue, FreeTreeMiner uses the database scan not only to determine the frequency of a given tree, but also to gather as much information as possible for possible extensions of the given tree. Once, the database scan routine has found the occurrence of a tree in a graph, it takes a second look at the extension points of the tree. All edges, that connect to those extension points in the graph and that are not yet contained in the free tree are saved in an *extension table*. At the end of the database scan, the algorithm has information not only about the frequency of the original tree, but also on the frequency of possible extensions. This information can be utilized to discard apparently infrequent extensions even during candidate generation process.

To describe the candidate generation process, we need a few basic concepts. First of all, consider a free tree t and a graph g. As explained above, the leaves with the largest depth in the tree are marked as extension points. Leaves that are not extension points are called *dead leaves*. An edge-node pair that can be appended to an extension point or dead leaf p is called a *p-extension*. In the following we will denote the extension of an edge-node-pair e to the extension point p by the ordered pair $(p : e)$. Now consider a particular frequent free tree t of height h_t and a dead leaf l of t at height h_l with $h_l < h_t$. If we extend t at l, the resulting tree is still of height h_t. So, if the FreeTreeMiner

Algorithm 3 The recursion step used in FreeTreeMiner. It utilizes the information gathered during the database scan to generate all frequent candidates for the next level.

```
procedure DepthSearch($t$, $d$, $m$)
    $ext \leftarrow$ DatabaseScan($t$, $d$)
    if the support of $t$ in $d$ is at least $m$ then
        Output "$t$ is frequent."
        $cand \leftarrow \{\{(p : e)\} \mid (p : e)$ is row in $ext\}$
        while $cand \neq \emptyset$ do
            $newCand \leftarrow \emptyset$
            for all candidates $c$ in $cand$ do
                if support for $c$ in $ext$ is at least $m$ then
                    Build tree $t'$ from $t$ and $c$
                    if $t'$ is in canonical form then
                        DepthSearch($t'$, $d$, $m$)
                    end if
                    if $t'$ is well-ordered then
                        $newCand \leftarrow newCand \cup \{c \cup \{(p : e)\}$
                        $\mid (p : e)$ is row in $ext$ and $(p : e) \notin c\}$
                    end if
                end if
            end for
            $cand$ = $newCand$
        end while
    end if
end procedure
end procedure
```

algorithm can guarantee that on each recursion level n all frequent trees of height n are found, it does not need to consider extensions at dead leaves to generate the candidates for step $n + 1$, because there exists a tree t' of height n, which is already extended at l, but still is of height n and has therefore been found in the previous recursion step. Extending t' at its extension points leads to the same trees of height $n + 1$ as if we had extended t at its dead leaf. Since FreeTreeMiner finds in fact all frequent trees of height n in recursion step n, it considers extensions at extension points only. A set of extensions to the extension points p of a tree t is called an *extension candidate* of t.

During the database scan, FreeTreeMiner collects all extensions $(p : e)$ to all extension points p of a tree t for all occurrences of t in all graphs in the *extension table*. This data structure is organized as a table with two columns and each row representing one particular extension $(p : e)$. The algorithm starts with an empty table and adds a new row for each new extension that is encountered during the database scan. It stores for each row the name of the extension $(p : e)$ in the first column, and in the second column the set of graphs in which t extended by $(p : e)$ occurs, the so-called *support set*. Thus, one can determine the frequency of t extended by $(p : e)$ solely by examining the extension table. Pseudo

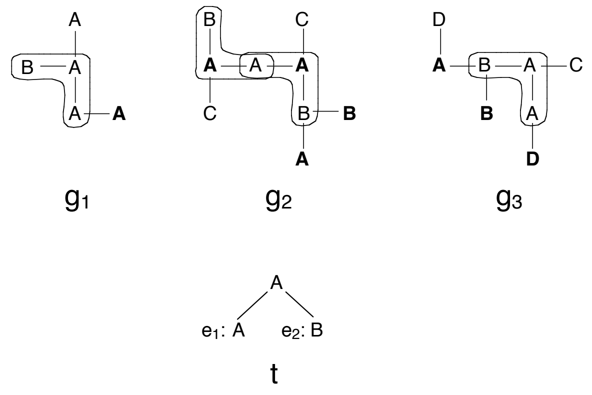

Figure 2. Example database and search tree.

Algorithm 4 The main loop of TreeFreeMiner. It mines for all free trees with a minimum support of m in the database d.

```
procedure MineFreeTrees($d$, $m$)
    $frequentLabels \leftarrow$ set of all node labels, which appear in at least $m$ graphs in $d$
    $oneTrees \leftarrow$ set of all trees with one node and a label from $frequentLabels$
    for all $t$ in $oneTrees$ do
        DepthSearch($t$, $d$, $minSupport$)
    end for
end procedure
```

code for the database scan is given in algorithm 2.

After the database scan, FreeTreeMiner uses the extension table to generate reasonable extension candidates. It starts with extension candidates that consist of exactly one extension $(p : e)$ and that are frequent according to the extension table. If those candidates are in canonical form, the DepthSearch routine calls itself recursively with each candidate to test the candidate's frequency and to generate further candidates. After the execution returns from the recursion, the algorithm generates candidates containing two extensions by adding one new extension from the extension table to the the candidates of size one. The frequency of the candidates of size two is then estimated by counting the number of elements of the intersection of the support sets of the two extensions. If this estimated support is larger than the minimum support and the candidate is in canonical form, it is used in a recursion step and as the basis for the generation of the candidates of size three. This loop continues with the generation of candidates of size three, four, etc. until all candidates are either infrequent or not in canonical form. Note that the size of the intersection of the support sets of the extensions in a candidate is not necessarily equal to the support of the extended tree in the database. Consider a tree with two extension points p_1 and p_2. Assume further that this tree occurs twice in a graph g, the first time with extension $(p_1 : e_1)$ and the second time with extension $(p_2 : e_2)$. Thus, the intersection of the support sets for the candidate $\{(p_1 : e_1), (p_2 : e_2)\}$ does contain g even though the tree extended by $(p_1 : e_1)$ and $(p_2 : e_2)$ is not contained in g. This can lead to false candidates that are filtered out during the database scan in the recursion step. Algorithm 3 sketches the candidate generation process.

As an example, consider the setting in figure 2: we are looking for patterns with a frequency of at least two in the database containing the three graphs g_1, g_2, and g_3. During the database scan we wish to determine the frequency of the free tree t, which has two extension points e_1 and e_2. Obviously, the frequency of t is three, because it occurs in all three graphs (one occurrrence in g_1 and g_3 and two occurences in g_2). During the scan all possible extensions of t in g_1, g_2, and g_3 (printed **bold** in the figure) are entered in the extension table. Thus, after the scan the extension table looks as follows:

Extension	Support Set
$(e_1 : A)$	$\{g_1, g_2\}$
$(e_1 : D)$	$\{g_3\}$
$(e_2 : A)$	$\{g_2, g_3\}$
$(e_2 : B)$	$\{g_2, g_3\}$

From this table the algorithm generates all candidates containing $(e_1 : A)$, $(e_2 : A)$, and $(e_2 : B)$.

The main loop of FreeTreeMiner, as described in algorithm 4, is straightforward: it simply searches for all frequent node labels in the database, generates all frequent trees with one node from this information and calls the recursion step with those trees.

In its standard version, FreeTreeMiner outputs all frequent subtrees for a given database. For our experiments we implemented a couple of extensions that enable more sophisticated queries and output formats. A first extension is to modify the recursion step to allow for a more sophisticated query language. The basic recursion step checks only for a minimum frequency criterion. However, one can modify the recursion step so that a candidate is a part of the solution only if a given conjunction of monotonic or anti-monotonic constraints is satisfied. For example, one could search for all free trees that have minimium frequency m_{min} and a maximum frequency m_{max}. One can even combine constraints on different databases. For the HIV experiment in section 4, we used a minimum frequency constraint on the first database, and a maximum frequency constraint on the second database.

Another extension deals with the way the solution space is represented. A solution space of patterns that satisfy a conjunction of monotonic and anti-monotonic constraints is a *version space*. It can be expressed by the set of all most maximal specific patterns that satisfy the constraints (the so-called *S-set* and a set of all most general patterns that satisfy the constraints (the *G-set*). One can show that G- and S-set form the borders of the solution space, in the sense that a free tree t is a part of the solution space iff there is a tree $s \in S$ and a tree $g \in G$ so that t is a supertree of g and a subtree of s. Storing the borders of a solution space is usually more space efficient than storing the whole space itself.

The S-set can be derived in the following way: for every frequent free tree t the algorithm keeps track of the frequency of the descendant trees c that were generated by extending t. If there is at least one frequent descendant c, t has a frequent supertree and is therefore not an element of the S-set. If none of the descendants is frequent, then t might be a part of the S-set and is stored for later retrieval. However, not every tree t without frequent descendants is a part of the S-set: it might be the case that there is a supertree of t whose canonical form can not be derived by extending t and was therefore not in the candidate list. Thus, after FreeTreeMiner finishes its mining process, it checks whether one tree is a subtree of another one for every pair of trees in the S-candidate set. All subtrees found in that way are removed from the candidate set. The remaining trees form the S-set.

The G-set for minimum frequency constraints can be generated in a similar way: all frequent trees that have only frequent ancestors are recorded in a candidate set. The post-processing removes all trees that are supertrees of at least one other tree in the candidate set. The remaining trees build the G-set. The post-processing step can be done in polynomial time, because the underlying subtree isomorphism test is polynomial.

4 Experimental Results

In this section, we present the result of the new approach on two datasets taken from the National Cancer Institute's (NCI) Developmental Therapeutics Program (DTP, `http://dtp.nci.nih.gov/`).

4.1 NCI DTP Anti-HIV Screening Data

The DTP AIDS Antiviral Screen program (`http://dtp.nci.nih.gov`) has checked tens of thousands of compounds for evidence of anti-HIV activity. Available are screening results and chemical structural data on compounds that are not covered by a confidentiality agreement. The available database (October 1999 Release) contains the screening results for 43,382 compounds. Since the first data mining application in this domain [11], it has been used as a benchmark for graph mining algorithms [8, 2].

The screen utilizes a soluble formazan assay to measure protection of human CEM cells from HIV-1 infection [19]. Compounds able to provide at least 50 % protection to the CEM cells were retested. Compounds that provided at least 50 % protection on retest were listed as moderately active (CM, confirmed moderately active). Compounds that reproducibly provided 100 % protection were listed as confirmed active (CA). Compounds neither active nor moderately active were listed as confirmed inactive (CI). In this experiment, we are only comparing CA with CM (further experiments, for instance, with CA vs. CI are under way).

The experiment is intended to enable a partial comparison with MolFea [11] and the latest version of AGM [8], because FreeTreeMiner is steering a middle course between these two. In order to obtain results comparable with both systems, we performed experiments as follows: the minimum support of fragments in CA is set to 16, the maximum support in CM is set to 12.

For the above setting, AGM took about 3 hours on a PentiumIII-667 MHz and returned about 730,000 induced subgraphs. An induced subgraph is a subset of the vertices of a graph together with all edges whose endpoints are contained in this subset. Our first experiment returned over 4,000,000 free trees, which is, at first sight, contradicting the results of AGM.

Upon closer investigation, AGM's performance is due to the restriction to induced subgraphs, which is boosting its performance. Applied to small molecules, this restriction means that AGM has to close rings (it cannot return rings with one atom missing). In order to enable a comparison, we also performed experiments with induced free trees in graphs. Here, we obtained 351,251 free trees in 6,549 seconds runtime, which is in line with the result obtained by AGM. So, although we are using a yet unoptimized version of FreeTreeMiner, the results are much as expected given the complexity considerations above: we are finding a subset of the graph patterns in a shorter runtime. With similar settings (cf. [8]), MolFea finds 684 linear path patterns in 2,054 seconds.

Note that, given such a result, it should be possible to compute a border representation of the solution set, whereas this would be much harder given that the subgraph isomorphism test is NP-complete and the solution space contains 730,000 patterns.

4.2 NCI DTP Anti-Cancer Screening Data

Whereas we were using the anti-HIV screening data as a benchmark, we are now showing the scalability on another dataset from the DTP without defined classes of compounds. To this purpose we are applying FreeTreeMiner to the 37,330 compounds studied in the DTP Human Tumor Cell Line Screen program.

Table 1. Minimum support, number of solution free trees, size of the positive border (Bd^+ aka S) and runtime of FreeTreeMiner on the compounds used in the NCI DTP human tumor cell line screen.

Min.supp.	$\|SolTrees\|$	$\|Bd^+\|$	Runtime (s)
20 %	128	32	947
15 %	232	48	1,283
10 %	476	125	1,900
5 %	1,893	464	5,017
2 %	13,762	3,160	45,013

Note that for the HIV data the *minimum* support was computed on a few hundred compounds only. In contrast, our goal on the DTP Anti-Cancer Screening data is to show the scalability of the approach on a much larger dataset of chemical compounds, which has not been done before in the data mining literature. Another motivation was to use this as a starting point for further investigations with these data in the domain of toxicogenomics, which represents a natural extension of our previous work in predictive toxicology [4, 17, 9].

Table 1 summarizes the results for varying minimum support thresholds. As can be seen, it is possible to work with low support levels on such datasets. Another observation is that the the combinatorial explosion becomes apparent only at a minimum support level of 2 %. All in all, these results suggest that a large-scale analysis of such data is within reach.

5 Related Work

AGM (Apriori-based graph mining) [6, 8, 7] is an Apriori-style algorithm for mining frequent subgraphs in graph data. Two frequent subgraphs from the previous level are merged to obtain candidate subgraphs for the current level. Graphs and subgraphs are represented as adjacency matrices. A canonical form is computed on the basis of an order on the vertex labels and a lexicographic order on the adjacency matrices. Since the generated candidates are not necessarily in canonical form, large parts of AGM deal with the problem of transforming the adjacency matrices into a normal form and from that to the canonical form. AGM can be extended to derive only connected induced subgraphs, which reduces the search space and solution space dramatically on some data sets. Since AGM uses Apriori's levelwise search approach, it has to process large amounts of candidates for each level. It also suffers from the comparatively high costs for generating new candidates and pruning infrequent candidates.

gSpan [20] is an algorithm for frequent subgraph discovery based on a canonical form of graphs (derived from its minimum spanning tree), and depth-first search. gSpan was motivated by the observation that candidate pruning can be very costly in graph mining. The main difference between gSpan and traditional approaches is, that gSpan traverses a search tree of so-called DFS-codes, where each code represents the canonical form of a graph. Another difference is that gSpan's use of depth first search instead of levelwise search allows for the removal of edges and whole graphs from the graph database, after all relevant patterns for this edge or graph are processed. Just as FreeTreeMiner, gSpan generates patterns during the database scan. Like AGM, gSpan needs elaborate computations to deal with non-canonical forms. gSpan has been applied to the Predictive Toxicology Evaluation Challenge dataset, which is a bit outdated [4] and might be affected by data quality issues [5]. Thus, we do not have a direct comparison between gSpan and FreeTreeMine.

FSG [12] is also an Apriori-style algorithm working with adjacency matrices. It is conceptually similar to AGM, but is employing a different edge growing strategy.

Recently, tree mining has attracted a lot of attention, mostly motivated by the need for analyzing XML documents [21, 16, 1]. Often the task is to find variants of rooted subtrees in rooted trees. In most approaches unconnected matches are allowed, such that not only the direct ancestors are considered, but also indirect ancestors. Virtually all approaches are defining a canonical form of trees in order to avoid redundancy in search.

In summary, our approach differs in that we consider free trees for more efficient graph mining, that we add more than one edge or node per recursion step, and that we utilize the information gained during the database scan to prune the lattice of possible extensions. In this way we can minimize the number of database scans.

6 Conclusion

In this paper we have introduced free tree patterns into graph mining. Free trees have several computational advantages over general subgraphs, which make them suitable candidates for advanced graph mining systems, such as inductive databases performing constraint-based mining on graphs. We presented FreeTreeMiner, a new algorithm for mining free trees in general graphs, which is making very efficient use of a canonical form defined for this class of patterns. In experiments we have found that the algorithm can mine at a minimum support level of 2 % in a database of more than 37,000 chemical compounds. Considering all theoretical and practical arguments, we believe that free trees and their intelligent use in specialized mining algorithms should have a place in the arsenal of graph mining algorithms. In the future, we are planning to investigate the use of smart data structures such as version space trees [13] for free trees, to explore the possibility of constraint-based mining in more detail and to extend the general framework for branched, three-dimensional molecular fragments.

Acknowledgements

We would like to thank Takashi Washio, Akihiro Inokuchi, and Christoph Helma for their help and fruitful discussions.

References

[1] K. Abe, S. Kawasoe, T. Asai, H. Arimura, and S. Arikawa. Optimized substructure discovery for semi-structured data. In *Proceedings of the 6th European Conference on Principles and Practice of Data Mining and Knowledge Discovery (PKDD-2002)*, pages 1–14, 2002.

[2] C. Borgelt and M. Berthold. Finding relevant substructures of molecules: Mining molecular fragments. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 51–58, 2002.

[3] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, 1:231–255, 1994.

[4] C. Helma and S. Kramer. A survey of the predictive toxicology challenge 2000–2001. *Bioinformatics*, in press, 2003.

[5] C. Helma, S. Kramer, B. Pfahringer, and E. Gottmann. Data quality in predictive toxicology part 1: Identification of chemical structures and calculation of chemical descriptors. *Environmental Health Perspectives*, 108:1029–1033, 2000.

[6] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the 4th European Conference on Principles and Practice of Data Mining and Knowledge Discovery (PKDD-2000)*, pages 13–23, 2000.

[7] A. Inokuchi, T. Washio, and H. Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354, 2003.

[8] A. Inokuchi, T. Washio, Y. Nishimura, and H. Motoda. General framework for mining frequent patterns in structures. In *Proceedings of the ICDM-2002 workshop on Active Mining (AM-2002)*, pages 23–30, 2002.

[9] S. Kramer, E. Frank, and C. Helma. Fragment generation and support vector machines for inducing SARs. *SAR and QSAR in Environmental Research*, 13(5):509–523, 2002.

[10] S. Kramer and L. D. Raedt. Feature construction with version spaces for biochemical applications. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML-2001)*, pages 258–265, 2001.

[11] S. Kramer, L. D. Raedt, and C. Helma. Molecular feature mining in HIV data. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-01)*, pages 136–143, 2001.

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

[13] L. D. Raedt, M. Jaeger, S. D. Lee, and H. Mannila. A theory of inductive query answering. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 123–130, 2002.

[14] L. D. Raedt and S. Kramer. The level-wise version space algorithm and its application to molecular fragment finding. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 853–862, 2001.

[15] R. Shamir and D. Tsur. Faster subtree isomorphism. In *Israel Symposium on Theory of Computing Systems*, pages 126–131, 1997.

[16] A. Termier, M.-C. Rousset, and M. Sebag. Treefinder: a first step towards XML data mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 450–457, 2002.

[17] H. Toivonen, A. Srinivasan, R. D. King, S. Kramer, and C. Helma. Statistical evaluation of the predictive toxicology challenge 2000–2001. *Bioinformatics*, in press, 2003.

[18] N. Vanetik, E. Gudes, and S. E. Shimony. Computing frequent graph patterns from semistructured data. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 458–465, 2002.

[19] O. Weislow, R. Kiser, D. Fine, J. Bader, R. Shoemaker, and M. Boyd. New soluble formazan assay for HIV-1 cytopathic effects: application to high flux screening of synthetic and natural products for AIDS antiviral activity. *Journal of the National Cancer Institute*, 81:577–586, 1989.

[20] J. H. Xifeng Yan. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 721–724, 2002.

[21] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the 8th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM press, 2002.