A Quickstart in Frequent Structure Mining can make a Difference

Siegfried Nijssen
LIACS, Leiden University
Niels Bohrweg 1, 2333 CA Leiden
The Netherlands
snijssen@liacs.nl
http://www.liacs.nl/home/snijssen/

Joost N. Kok
LIACS, Leiden University
Niels Bohrweg 1, 2333 CA Leiden
The Netherlands
joost@liacs.nl
http://www.liacs.nl/home/joost/

ABSTRACT

Given a database of graphs, a graph mining algorithm searches for substructures that that satisfy constraints such as minimum frequency, minimum confidence, minimum interest and maximum frequency. Examples of substructures include graphs, trees and paths. For these substructures many mining algorithms have been proposed. In order to make graph mining more efficient, we propose to use the "quickstart principle" exploiting the fact that the various substructures are contained in each other. In the search for sub-structures, first paths are considered, then paths are transformed to trees and finally trees are transformed to graphs. In this way one can use more efficient algorithms for the simple substructures and only use the advanced algorithms when they are really needed. We implemented a new GrAph/Sequence/Tree extractiON (GASTON) algorithm based on this principle and we present results on mining a large molecular database.

Keywords

Data mining, structures, semi-structures, graphs, frequent item sets

1. INTRODUCTION

In recent years data mining of complicated structures such as graphs, trees, molecules, XML documents and relational databases has attracted a lot of researchers. Especially the idea of discovering all frequent substructures of such databases has recently led to a large number of specialised algorithms for mining paths, trees and graphs in databases of trees or graphs. In this paper, we aim to take this research a step further by investigating the *interdependencies* between these patterns. Experiments on small molecular databases reveal that the largest numbers of frequent substructures in such databases are actually free trees. Free trees are much simpler structures than general, cyclic graphs, and many efficient algorithms exist for free trees. Therefore, we investigate the possibilities of *quickstarting* the search for frequent structures by integrating a frequent path, tree and graph miner into one algorithm called GASTON.

The main challenge in the development of this algorithm is how to split up the discovery process into several phases efficiently. Ideally, the algorithm should behave like a specialised free tree miner when faced with free tree databases, but should also be able to deal with graphs databases efficiently. In this paper, we show how this can be done, and we show that the application of the Quickstart principle can indeed make a difference in performance. We use GASTON to mine the NCI'99 database for frequent substructures and successfully compare the time efficiency of our algorithm with a number of state-of-the-art structure mining algorithms. We show that our algorithm can also easily be used to do *difference* mining experiments, in which we compare the NCI database to the NCI AIDS database by mining for emerging structures.

As background knowledge for our paper we could mention a large number of publications. In this paper we will only refer to recent publications. For a larger overview, the reader is invited to visit our homepage for frequent structure mining, which can be found at

`http://www.liacs.nl/home/snijssen/structmining`.

The overview of the rest of the paper is as follows: first we give mathematical preliminaries, then enumeration strategies, frequency evaluation and finally the experimental results. We made an effort to keep the paper self-contained, despite the complexity of the GASTON algorithm.

2. MATHEMATICAL PRELIMINARIES

To understand our algorithm some basic background knowledge is required with respect to graphs, trees and paths. We will briefly discuss them in this section. The definitions are similar to those used in other papers, for example [9, 21, 19, 7, 11, 18, 3]. A labeled graph G consists of a set of nodes V, a set of edges $E \subseteq V \times V$ and a labeling function $\ell : V \cup E \rightarrow \mathcal{L}$ that assigns labels $\mathcal{L}$ to all edges and nodes. We only consider undirected graphs, i.e. (u_1, u_2) is the same edge as (u_2, u_1). An edge is incident to a node if one of its endpoints is in that node. The number of edges that is incident to a certain node is called the degree of that node. Two nodes are adjacent if there is an edge between the two

.

nodes. A sequence of nodes $v_1, v_2, \ldots, v_m$ from V is a path if (v_i, v_{i+1}) is in E. The length of a path is defined by the number of edges in the path. If $v_1 = v_m$ the path is called a cycle. From now on, we only consider simple paths, which are paths in which $v_i \neq v_j$ if $i \neq j$. If there is a path between each pair of nodes in a graph, this graph is connected. In this paper, we will only consider connected graphs. Given two graphs $G_1 = (V_1, E_1, \ell_1)$ and $G_2 = (V_2, E_2, \ell_2)$, an embedding of G_1 in G_2 is an injective function $f : V_1 \to V_2$ such that (1) $\forall v \in V_1 : \ell_1(v) = \ell_2(f(v))$ and (2) $\forall (v_1, v_2) \in E_1 : (f(v_1), f(v_2)) \in E_2$ and $\ell_1(v_1, v_2) = \ell_2(f(v_1), f(v_2))$. The graph G_1 is a subgraph of G_2, denoted by $G_1 \subseteq G_2$, if there is an embedding of G_1 in G_2. If G_1 is a subgraph of G_2 and G_2 is a subgraph of G_1, then G_1 and G_2 are called isomorphic. Embedding is also called a subgraph isomorphism and defines a partial order on labeled graphs.

In this paper, we specifically study three special subclasses of graphs:

- *Paths.* They are special graphs in which two nodes have degree 1, while all other nodes have degree 2. Note that a path which *occurs in* a graph is not always a path *in* a graph.
- *Free Trees/Rooted Trees.* If a graph has no cycles, the graph is called a free tree. A rooted tree is a tree in which one node is singled out. This special node is called the root of the tree. We will always speak of either free trees or rooted trees to avoid confusion between these two types of trees. In a rooted tree, we will draw the root of the tree as the top node.

Examples are given in Figure 1. Note that paths are also free trees, and free trees are also graphs.

A special kind of tree is the spanning tree. A tree is a spanning tree of a graph if it has exactly the same number of nodes as the graph and the tree is furthermore a subtree of the graph.

We assume that a database $\mathcal{D}$ consists of a collection of graphs. The frequency of a graph G in $\mathcal{D}$ is defined by $freq(G, \mathcal{D}) = \#\{G' \in \mathcal{D} | G \subseteq G'\}$. The support of a graph is defined by

$$support(G, \mathcal{D}) = freq(G, \mathcal{D})/|\mathcal{D}|.$$

The primary task that our algorithm has to solve, is to find all graphs for which

$$support(G, D) \geq minsup,$$

for some predefined threshold *minsup* that is specified by the user. An important property is the following:

$$G_1 \subseteq G_2 \Rightarrow freq(G_1, \mathcal{D}) \geq freq(G_2, \mathcal{D}). \quad (1)$$

It follows that any (large) graph which contains a (smaller) graph which is not frequent, can not be frequent either. This property is the essential property that has been used for the construction of many data mining algorithms. The process of removing graphs from the search space using this property, is called (frequency based) pruning.

3. ENUMERATION

An important part of any frequent structure mining algorithm is the strategy for traversing the set of all possibly frequent graphs, also called the enumeration strategy. For an efficient graph mining algorithm it is essential that:

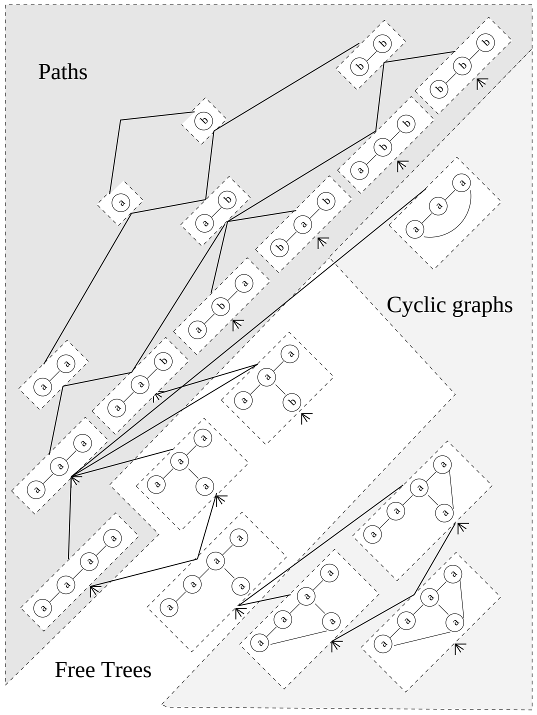


Figure 1: Part of the partial order of graphs

- large graphs are only considered after smaller subgraphs are considered (i.e. to allow for frequency based pruning);
- every graph is considered only once.

For simple structures, the second requirement can often be satisfied with not much computational overhead. For graphs the problem is more difficult. The source of the problem is that in order to make sure that one does not consider the same graph twice, one has to make sure that a graph is not isomorphic to another graph; at the moment no polynomial algorithm is known which accomplishes this task for general graphs. Hence in the worst case an exponential search is required.

A key observation is now that although for graphs no polynomial algorithm exists, for subclasses of graphs polynomial algorithms do exist. For example, for paths and trees efficient isomorphism algorithms do exist. This observation has led to development of specialised mining algorithms for paths [10] and free trees [3, 19].

Paths, trees and cyclic graphs can be put into a partial order, as follows. The top of the partial order consists of paths; paths to which an edge is added that connects to a different node than one of the end nodes, become free trees. After this transition from a path to a tree, further tree construction takes place by repeatedly adding one node and one incident edge. A free tree, in its turn, becomes a cyclic graph when an edge is added between two existing nodes. Starting from the root of the partial order, each cyclic graph can only be obtained by steps that first build a path, then (optionally) create a free tree, and finally create the cyclic graph. All of these “refinement” steps (except those that involve cyclic graphs), can be done with efficient, polynomial algorithms, leading to a “quickstart” of the search. The situation is illustrated in Figure 1.

In the sequel, each refinement of a structure by the addition of both a new node and an edge to connect this node, is called a *node refinement*. A refinement which only connects to existing nodes is referred to as a *a cycle closing refinement*.

Details of our enumeration strategies are discussed in the next subsections.

3.1 Path enumeration

The main problem in path enumeration is that a path can have two orientations, for example: $axaxb$ and $bxaxa$, where a and b denote node labels and x and y edge labels. One only wants to consider one of its orientations.

To guarantee enumeration without duplication, we follow an approach that has some similarities with the approach that was used in [10]. For each path, we define a unique predecessor in the partial order as follows. Given is a path in some orientation:

$$v_1 e_1 v_2 \cdots v_{n_1} e_{n-1} v_n.$$

First compare the labels at both ends of the path by comparing the tuple $(\ell(v_1), \ell(e_1))$ with the tuple $(\ell(v_n), \ell(e_{n-1}))$ lexicographically; if one end is higher than the other, the path without the highest tuple is considered to be the unique predecessor. If both tuples are equal, then we distinguish two cases. If the string is symmetric, it does not matter which one of the two end nodes is removed and the predecessor is uniquely defined too. If the string is not symmetric, we compare the following two oriented paths lexicographically with each other: $\ell(v_1)\ell(e_1)\ell(v_2)\cdots\ell(v_{n-1})$ and $\ell(v_n)\ell(e_{n-1})\ell(v_{n-1})\cdots\ell(v_2)$. The path corresponding to the lowest of these two strings is considered to be the predecessor of the path. Please note that the relation between these two strings is determined first by the relation between $(\ell(v_1), \ell(e_1))$ and $(\ell(v_n), \ell(e_{n-1}))$, and then by which of the two orientations of $\ell(v_2)\cdots\ell(v_{n_1})$ is the lowest.

A straightforward, recursive, approach to enumeration would now be as follows. If a path is symmetric, apply all possible node refinements at one end of the path; otherwise apply all possible node refinements at both ends of the path. Determine for each resulting path whether the original path is the unique predecessor of the resulting path and disregard paths which have not grown from their unique predecessor. Extend all other paths recursively.

For our algorithm, however, this approach is impractical. We would like to have a more precise characterization of the node refinements that are allowed. We obtain this as follows.

For one specific orientation of a path $v_1 e_1 v_2 \cdots v_{n_1} e_{n-1} v_n$, we maintain three *symmetry* variables, one for the oriented path $v_1 e_1 \cdots v_{n-1} e_{n-1} v_n$ (*total symmetry*), one for $v_1 e_1 \cdots v_{n-1}$ (*front symmetry*) and one for $v_2 \cdots v_{n-1} e_{n-1} v_n$ (*back symmetry*). Each of these variables has one of three values: 0, if the corresponding string is symmetric; -1, if the reverse string of the current orientation is the lowest; $+1$, if the string of the current orientation is the lowest.

If *total symmetry* is 0, a path may only be node refined at one end; otherwise the path may be extended in two directions as determined by the *symmetry* variables. Given a particular orientation of a path $v_1 e_1 v_2 \cdots v_{n_1} e_{n-1} v_n$ and its *symmetry* variables, one can determine as follows which labels $(\ell(v'), \ell(e'))$ may be appended at the back of that path:

- all labels for which $(\ell(v'), \ell(e')) > (\ell(v_1), \ell(e_1))$;
- the labels $(\ell(v'), \ell(e')) = (\ell(v_1), \ell(e_1))$ if *back symmetry* ≥ 0. Please note that if *back symmetry*$= 0$ and $(\ell(v'), \ell(e')) = (\ell(v_1), \ell(e_1))$, the next path is symmetric.

After appending a node after the path, the *total symmetry* is easily computed from the previous *back symmetry*, while the previous *total symmetry* is the *front symmetry* of the new path. Only the recomputation of *back symmetry* is therefore required and can be done in linear time. We will see later that even this computation is not always required.

The situation is analogous when nodes are prepended before the path. Please note that within our setup we have not defined which orientation is exactly generated, but we still have the guarantee that only one orientation is enumerated.

3.2 Free tree enumeration

For the enumeration of free trees many approaches can be taken. All existing specialised data mining algorithms use a so-called breadth-first normal form [19, 3]. This method has however practical disadvantages which we will discuss later. Also theoretically the method is suboptimal: free trees can be enumerated in constant time [20], which means that for a given free tree one can determine in constant time which refinements of this tree are allowable if one wishes to avoid duplicates. When a breadth-first normal form is used 'only' linear enumeration is possible. The constant time method of [20] can however not be applied straightforwardly because free trees are not enumerated in increasing size. We will present a new enumeration strategy for labeled free trees that has constant time complexity and will not have the aforementioned practical disadvantages. It is based on an enumeration strategy for rooted trees that was independently proposed by [2] and [15], and has strong similarities with the method proposed in [13] for unlabeled free trees.

3.2.1 A backbone for free trees

We will first state a well-known property of free trees. Consider one of the paths P in a free tree T for which the length is maximal, $P = \{v_1, \ldots, v_m\} \subseteq V_T$, then all other paths of the same maximal length have one or two nodes in common with this path:

- if m is odd, also all other paths of maximal length go through node v_i where $i = \frac{m+1}{2}$; v_i is called the centre of the free tree;
- if m is even, also all other paths of maximal length go though the nodes $\{v_i, v_{i+1}\}$ where $i = \frac{m}{2}$; these nodes are called the bicentre of the free tree.

A free tree is therefore either centred or bicentred. A centred tree can be conceived as a single rooted tree, while a bicentred tree can be conceived as two separate rooted trees of which the roots are interconnected. Within each rooted tree, we define the length of the path from a node to the root node to be the depth of that node. The length m of the longest path is called the diameter of the free tree. The highest depth in a free tree is $\lceil m/2 \rceil - 1$.

Now consider all oriented paths of maximal length that start in (each) root of the tree. By applying the labeling function on each of these paths, corresponding strings

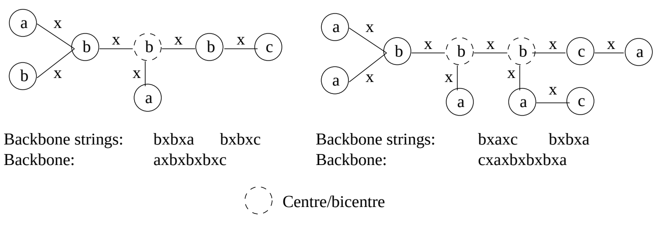


Figure 2: Examples of free trees, (bi)centres and backbones

are obtained from the paths; the strings we compare lexicographically. In centred trees, those two strings of maximal length which are lexicographically the lowest and which occur in paths that only have the root in common, we call the backbone strings of the free tree. In bicentred trees, the lexicographically lowest string in each of the two rooted trees is defined to be the backbone string of that rooted tree. By concatenating the reverse of the lowest backbone string and the highest backbone string, a single path is obtained which we call the *backbone* of the free tree. Examples of free trees, centres, bicentres and backbones are given in Figure 2.

Using this procedure for uniquely determining backbones for free trees, the partial order of free trees can be partitioned. Our enumeration strategy is built on the idea that free trees are only allowed to grow from a free tree which has exactly the same backbone and all free trees of a certain backbone grow from the path that corresponds to that backbone. If we can enumerate all free trees within a partition uniquely, in combination with our path enumeration strategy also all free trees are enumerated uniquely.

A tree is only refined using node refinements. Given a free tree with maximum path length m, refinements which would introduce another backbone in a tree are easily characterized:

- no node may be added at depth $\lceil m/2 \rceil$;
- if T has one centre and a node is added at depth $\lceil m/2 \rceil - 1$, the string of labels on the path from the root to the new node must be higher than or equal to the highest of the two backbone strings;
- if T is bicentred and a node is added at depth $\lceil m/2 \rceil - 1$, the new node occurs in one of the two rooted trees of the free tree; the string of labels on the path from the root to the new node must be higher than or equal to backbone string of the tree that the node occurs in.

We call these refinement constraints the *backbone constraints.*

3.2.2 *Depth sequences for rooted trees*

To enumerate free trees for a backbone, we use a method based on *depth sequences.* Details about depth sequences can be found in [15, 16, 2, 13]. Given a tree, a depth sequence is obtained by performing a prefix depth first walk; each time that a node is visited for the first time, first its depth is outputted, then the label of the edge going into that edge and finally the label of that node; we will call this combination a depth sequence tuple. Within the tuple for the root of the tree we represent the edge label by λ. Examples are given in Figure 4. Clearly, the depth sequence depends on the order with which the children of a node in the tree are visited. For an unordered, rooted tree, the *canonical*

```
Input: depth sequence $d_1\lambda\ell(v_1)d_2\ell(e_2)\ell(v_2)\dots d_n\ell(e_n)\ell(v_n)$
Output: an ordered tree $T$
$T_1$ := a tree with node label $\ell(v_1)$ as root.
for $i$:=2 to $n$ do
  $T_i$:= tree $T_{i-1}$, with a node with label $\ell(v_i)$
      connected by an edge with label $\ell(e_i)$ to the
      node at depth $d_i - 1$ of the rightmost path of $T_{i-1}$
return $T_n$
```

Figure 3: A tree construction algorithm

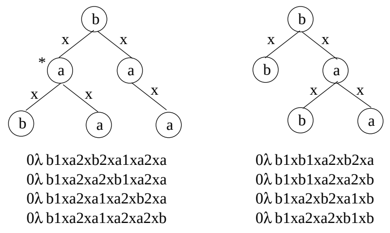


Figure 4: Unordered rooted trees, depicted in normal form, and all their possible depth sequences

depth sequence is defined as the depth sequence that is the lexicographically highest sequence among all possible depth sequences for that tree.

Every ordered tree corresponds to a depth sequence; the reverse is also true. The constructive algorithm of Figure 3 can be used to obtain a tree from a depth sequence. The mapping between ordered trees and depth sequences is therefore bijective. The canonical depth sequence therefore defines a canonical ordered tree.

Depth sequences have several properties that are of importance to efficient enumeration strategies [15, 16, 2, 13]. We wish to repeat one property here: every prefix of a canonical depth sequence is also the canonical depth sequence of the subtree that it represents. This property guarantees that the enumeration is still complete if one enumerates canonical depth sequences only and one refines these sequences only by adding a tuple at the back. Moreover, one can show that at most two tuples in a depth sequence —which correspond to two nodes in the tree— define which tuples may be added to the back of the sequence:

1. a new node must have a connecting edge label and a node label which sort lower or equal to those of its left sibling node;
2. the tuple which is appended at the back of the sequence must be equal to or lower than that of the *next prefix* tuple; the position of this tuple in the sequence is defined by the so-called *lowest prefix node*, which is the node with the lowest depth on the rightmost path for which the depth sequence of the subtree is a prefix of the depth sequence of its left sibling's subtree.

If one adds a depth tuple to a depth sequence, one can show that the position of the next prefix tuple in the resulting depth sequence can be computed in constant time from the position of this node in the previous sequence.

An important property of this enumeration strategy is the following. Consider a rooted tree and one of its allowable refinements which connects to a node v in that tree. Then this refinement is an allowable refinement of all ancestors trees in the partial order which also contain the node v. We will omit the proof (using depth sequences) here.

3.2.3 *Depth sequences for free trees*

The concepts at the basis of our enumeration strategy for free trees are most easily understood for the class of bicentred free trees in which the two backbone strings are not equal. Each such free tree can be constructed in two phases by first constructing a rooted tree for the lowest backbone string, and by subsequently —independently— constructing a rooted tree for the second backbone string. The problem of enumerating free trees then reduces to the problem of enumerating all rooted trees that contain a certain path and that satisfy the backbone constraints (among which the maximal depth constraint).

We approach this as follows. Given an alphabet of labels, we define the order $\pi(\ell)$ of each label $\ell \in \mathcal{L}$ to be the number of labels in the alphabet that is lexicographically lower. For a given backbone string $\ell(v_1)\ell(e_1)\ell(v_2)\cdots\ell(v_{n-1})$, we now define the following function g which maps depth tuples to other depth tuples:

$$g(i,\ell_1,\ell_2) = \begin{cases} (i, |\mathcal{L}|-\pi(\ell_1), |\mathcal{L}|-\pi(\ell_2)) & \text{if } \ell_1 \neq \ell(e_i); \\ (i, |\mathcal{L}|-\pi(\ell_1), |\mathcal{L}|-\pi(\ell_2)) & \text{if } \ell_2 \neq \ell(v_{i+1}); \\ (i, |\mathcal{L}|+1, |\mathcal{L}|+1) & \text{otherwise.} \end{cases} \qquad (2)$$

Note that this function is in fact a bijective mapping. It maps original labels to new labels with a different order, depending on the depth at which the label occurs. Given a depth sequence for a rooted tree that contains the given backbone, after application of g a new valid depth sequence results. Let us consider the canonical depth sequences for the new alphabet, then each canonical sequence must start with the following sequence: $0\lambda L1LL2LL3LL\cdots mLL$, where m is the length of the backbone string and $L = |\mathcal{L}|+1$. This can easily be seen by observing that one can never obtain a string with a prefix of higher labels.

We can now use the enumeration strategy of the previous section to enumerate all depth sequences for the alternative alphabet, where the enumeration is started from the sequence $0\lambda L1LL2LL3LL\cdots mLL$. Each of the enumerated depth sequences uniquely corresponds to a depth sequence for the original alphabet. During the enumeration, one can efficiently check that the backbone constraints are not violated for the tree in the original alphabet; thus, we have obtained an efficient strategy for uniquely enumerating all possible trees with a certain backbone string.

Given a certain tree for the first backbone, if the second backbone string is different from the first one, the same procedure can be repeated to enumerate trees for the second backbone string. If the second backbone string equals the first backbone string, special care must be taken that the same free tree is not enumerated twice. To solve this problem the depth sequences can be used. As the backbone strings are equal, also the relabeling function g is the same for both trees; the relabeled depth sequences of both trees can be compared lexicographically. If one makes sure that the depth sequence for the second backbone string is never higher than that of the first backbone tree, unique enumeration is guaranteed. To check this in constant time, both relabeled depth sequences can be put into one depth sequence and the same constant-time lowest prefix node procedure as for single rooted trees can be used. We omit the details.

The situation is more complicated for trees with one centre. Here we have to enumerate single trees that contain two backbone strings uniquely. First consider the case that the two backbone strings are unequal. We observe that in such a free tree, only one adjacent node of the root is contained in a path for the lowest backbone string. We start the enumeration by first enumerating the subtree for that particular node. We do so by removing the first node from the lowest backbone string, and by then enumerating all trees that contain the remaining string using the previously discussed method. Within this first subtree growing procedure, the backbone constraints for the first backbone are constantly checked. After that this subtree is grown for the first backbone, we enumerate all subtrees for the entire second backbone while constantly checking the refinements against the backbone constraints for the second backbone. Note that during this enumeration also paths of maximal length may be grown which start in the root of the tree. As none of this paths can however be equal to the first backbone (every path of maximal length in the second three must be lower than or equal to the second backbone), we need not to worry that a new subtree may grow from the root which could also have been grown from the first backbone. If both backbones are equal, again we first enumerate the subtree for one of the adjacent nodes of the root, and then enumerate the remaining part of the tree: as the relabeling function for both tree enumerations is equal, we can lexicographically compare the depth sequence of the newly grown trees with that of the first backbone to make sure that no tree is grown which is higher than the first grown subtree. Also here, by putting the entire tree in a depth sequence and by updating the lowest prefix node in constant time for each refinement of the tree, one can perform this lexicographical comparison in constant time.

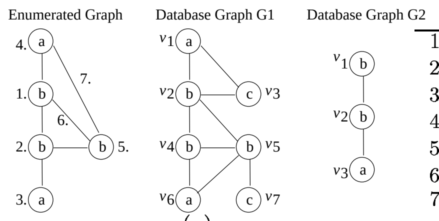
An important property for this free tree enumeration strategy is that any valid refinement of a free tree, is also a valid refinement of an ancestor free tree in the partial order, as long as that ancestor tree contains the node to which the refinement node connects.

3.3 General Graph enumeration

For cyclic graphs, a method for efficient duplication-free enumeration currently does not exist. All existing cyclic graph miners —including ours— have to resort to a generate-and-test method. In frequent graph mining algorithms, the problem is usually approached by defining new normal forms for graphs, for example using DFS codes (gSpan, [21]) or adjacency matrices (FSG [11], FFSM [7] and AcGM [9]). Once a new graph is generated in a certain encoding, these algorithms use a special-purpose exhaustive procedure to determine whether or not a lower encoding exists for the graph. Graphs for which such an encoding indeed exists, are discarded.

We approach the problem differently. Although the worst case performance of graph normalisation is exponential, for most practical graphs an excellent normalisation performance can be achieved by using so-called graph invariants. Instead of developing yet another normalisation procedure ourselves, we decide to rely on *Nauty* [12], which is a well-known, publicly available graph normalisation algorithm with excellent performance.

The idea is as follows. Once a cycle closing refinement is applied to a free tree or a path, this structure becomes a cyclic graph. From this moment on, we only allow further



(a)

	1	2	3	4	5	6	7	8
1.	(λ,G_1,v_2)	$\underline{(\lambda,G_1,v_4)}$	(λ,G_1,v_5)	(λ,G_2,v_1)	(λ,G_2,v_2)			
2.	$(1,G_1,v_4)$	$(1,G_1,v_5)$	$\underline{(2,G_1,v_2)}$	$(2,G_1,v_5)$	$(3,G_1,v_2)$	$(3,G_1,v_4)$	$(4,G_2,v_2)$	$(5,G_2,v_1)$
3.	$(1,G_1,v_6)$	$\underline{(3,G_1,v_1)}$	$(5,G_1,v_1)$	$(6,G_1,v_6)$	$(7,G_2,v_3)$			
4.	$(1,G_1,v_1)$	$\underline{(2,G_1,v_6)}$						
5.	$(1,G_1,v_5)$	$\underline{(2,G_1,v_5)}$						
6.	1	$\underline{2}$						
7.	$\underline{2}$							

(b)

Figure 5: A graph and the embeddings of all its ancestors in two graphs in a database

cycle closing refinements, similar to the approach that was suggested in [6]. In this way, graphs are only allowed to grow from a spanning tree of that graph. Each graph that is obtained by a cycle closing refinement, is normalised with Nauty and searched for in a hash structure. The graph is discarded if it turns out that the graph has already been enumerated, otherwise the graph is stored in the hash table and may be refined further.

At first sight, this strategy may not seem very promising as the number of possible cycle closing edges can be very large. The following observations may reduce the problem. First, by considering the order in which the nodes are introduced in a tree or path, all nodes in a tree or path can be numbered uniquely. This is illustrated in Figure 5(a). Every possible cycle closing edge can be identified uniquely by a tuple of two node numbers and an edge label ($(1, 4, -)$ for edge 6. and $(4, 5, -)$ for edge 7. in the example); the cycle closing tuples can be ordered first on lowest node number, then on highest node number and finally on the edge label. For a given spanning tree, we therefore obtain a set of tuples of which —in principle, at least— all possible subsets have to be enumerated. For this enumeration, we use similar mechanisms as in item set algorithms to enumerate subsets uniquely. Please keep in mind that cycle closing refinements may lead to graphs that are pruned because of isomorphism; the completeness of the search is however not affected by the set enumeration approach as any edge which can be added to a pruned graph can also be added to all its isomorphic graphs.

Second, although in theory the number of edges can be very large, in practice this number can be limited using an efficient scheme for frequency based pruning that we will discuss in the next section.

Third, although in general enumeration strategies which require that all previous structures are stored, are undesirable, our enumeration strategy is part of a data mining algorithm which has as explicit task to store discovered structures.

To hash a graph in the hash table, we use the following hash values: first, the number of nodes in the graph, then, the number of edges and finally the product of the degrees of all nodes. Then, we perform a binary search on (1) the node labels that are used, (2) the edge labels that are used, (3) the number of times that labels are used and (4) the normal form of the graph as determined by Nauty.

4. FREQUENCY EVALUATION

Just like frequent item set miners, frequent structure miners can be subdivided into two classes: breadth first [8, 11] and depth first miners [19, 21, 7, 6, 3, 1, 22, 15, 16, 2, 17]. At the expense of additional main memory, the latter class of mining algorithms is widely known to yield better performance. Also in our algorithm we will use a depth-first strategy. The advantage of a depth first strategy is that it makes it feasible to store additional information with each structure regarding its embeddings in the database. One can use that information to speed up the determination of embeddings of the refined structures.

Building on this principle, there is still a large variety in the amount of additional embedding information that can be stored with each structure. For trees approaches have been studied that store a subset of all possible embeddings [15, 2]. The idea is that one structure can often be embedded in multiple ways in the same set of nodes of another structure and space can be saved by only storing embeddings for different sets of nodes. Prior to the development of our graph miner, we performed an experimental comparison between a rooted tree miner in which a subset of all embeddings is stored [15], and an approach in which all embeddings are stored. We found out that the latter approach did not only have a better time performance, but also that it required less main memory. In practice, the number of multiple embeddings for the same subset of nodes was so low that the overhead of keeping track of them was too high. Also our approach is built on the idea of maintaining all embeddings of a graph in main memory, but we developed memory saving data structures to avoid the allocation of too much memory.

The following efficient data structure is used to store all embeddings of a structure and its ancestors in the partial order. For the root of the partial order —which corresponds to a single label— we store an embedding list of all occurrences of that label in the database. For a structure lower down the partial order which is obtained by a node refinement, we store an embedding list of embedding tuples which consist of (1) a pointer to an embedding tuple of the parent structure in the partial order and (2) the identifier of a graph in the database and a node in that graph. We denote the respective components of an embedding tuple t by t.parent, t.graph and t.node. If a structure is obtained by a cycle closing refinement, the embedding list consists solely of pointers to embedding tuples of its parent structure. The data structure is maintained such that of each structure all embeddings can be obtained by scanning its embedding list, and by following the parent pointers of each tuple in that list. The frequency of a structure is determined from the number of different graphs in its embedding list.

The embedding lists are illustrated in Figure 5(b), with the following notation. Each row in the table denotes the embedding list of an ancestor of the graph of Figure 5(a); the ancestors of a structure in the partial order are numbered such that 1. is the oldest ancestor and the root of the partial order. Each tuple in an embedding list has a unique position (or index) in that list, as given by the number of the column of the table. The indexes are used in the embedding tuples of the refined structure to encode pointers. The only embedding of graph Figure 5(a) in the database is underlined. Note that the parent tuple of an embedding tuple not necessarily belongs to an adjacent node.

Apart from the embedding list of the current structure, we also store a, possibly large, set of additional embedding lists for possible refinements of that structure. The embedding list for a refinement consists of exactly the same list that would result if that refinement would be applied to the current structure. The idea is to use these embedding lists later on to determine the embedding lists of further refined structures. In detail, we store embedding lists for the following refinements, if the refinement leads to a frequent structure:
For paths: all possible node refinements, including refinements which would lead to a non-canonical path; all possible cycle closing refinements, including the refinements which would lead to a graph that has already been enumerated. The rationale of also storing node refinements that lead to non-canonical paths is that such refinements may still later be used during the construction of free trees; the reason for also determining isomorph cycle closing refinements is that such refinements may not lead to isomorphic graphs later on.
For free trees: all canonical node refinements and all possible cycle closing refinements; during our discussion of free tree enumeration, we have pointed out that a non-canonical refinement of a free tree can never be applied to a later free tree to obtain a canonical tree;
For cyclic graphs: all cycle closing refinements, including those that lead to a graph isomorph to an earlier cyclic graph.

Each frequent refinement for which the embedding list is also stored, we call a *leg* of that structure, following the terminology introduced by Chi et al. in FreeTreeMiner [3]. The idea however also closely corresponds to that of maintaining suboptimal CAMs in FFSM [7], and the idea of maintaining right siblings in FARMER [17].

In Figure 6 we present the outline of the GASTON algorithm. It shows how the legs are maintained during the depth-first run of our algorithm. The main idea is that all embedding lists of all frequent legs of a certain structure can be computed from the embedding lists of legs of its predecessor structure in the partial order, except for those legs that connect to a node that was added by a node refinement. Next we will provide some comments on the code of our algorithm as given in Figure 6.

In lines (2) a new graph is constructed from the previous one; in line (4) the embedding list of each leg of the new structure is computed, either by an extension algorithm or by a list join algorithm that takes one of the legs of the ancestor structure as input. All legs of the ancestor structure are considered, including the ones that cannot actually be added to the current path to avoid duplicates. The situation is different for free trees. In line (12) only legs are added for the new node that could immediately be used as refinement for the current tree. Likewise in line (13) only those legs are joined which are immediate valid refinements for the current tree. In line (19) only those cycle closing legs are copied in which higher numbered nodes are involved; still, some of these legs may lead to a graph which is isomorphic to an earlier graph, and may not be considered for immediate further refinement in line (17).

In line (24) of the join procedure all embedding tuples are combined which extend a common parent embedding of the predecessor structure in the partial order, such to obtain embedding tuples for the leg of the new structure. For joins of cycle closing legs and node closing legs, as well as joins of cycle closing legs and cycle closing legs, the procedure is similar; for joins of two cycle closing legs the procedure comes down to the computation of an intersection of two lists of integer index lists.

In line (31) and line (33) by a 'corresponding' leg we mean the following. Consider an embedding of the current structure in a graph in the database; then each node in the embedding corresponds to a node in the current structure. Each node which is adjacent to a node v in the current embedding, but which is not part of the current embedding itself, would be part of the embedding of the refined structure which is obtained by connecting an edge to the node corresponding to v.

In line (29) for each embedding tuple the parent pointers are followed to determine whether an adjacent node does not already occur in the embedding to which this tuple corresponds. The procedure restricted-extend which is called in line (12) differs from the normal extend procedure in the sense that only for those candidate refinements which may immediately be added to the current structure embedding lists are constructed.

5. DISCUSSION

In section 3.2, we stated that breadth first enumeration strategies have practical disadvantages. We come back to that statement now. We stressed that within our free tree enumeration strategy only embedding lists are built for node legs that can immediately be used as refinement. As such, no redundant node refinement legs are ever evaluated during the procedure for discovering free trees. This is not the case for breadth first normal forms. The algorithm which was developped by Chi et al. for searching frequent free trees [3] requires that every bicentred free tree is evaluated two times, once for each tree that is obtained by choosing one of the bicentres as the root.

Although we eliminated the redundancy for the free tree enumeration phase, for the other two phases still some legs are evaluated that are not used as immediate refinement. For paths, this is a small problem, as the case in which a leg is never actually used as a refinement is rare. Only legs which occur at the end of a path which cannot be extended further in that direction, may be redundant if their label is so low that they cannot be used without violating the backbone constraints.

For graphs, the problem of evaluating redundant legs may be larger, especially if the number of frequent patterns with many cycles is large. Within our algorithm, the evaluation of redundant legs is required to guarantee the completeness of the search. There are two reasons for this problem: (1) we require redundant legs as we do not take into account the *automorphisms* of the graph: the addition of two different edges to one graph may lead to two isomorphic graphs; the joint addition of two edges could however yield a graph that is not obtainable otherwise; (2) to take into account the automorphisms, we need a different normalisation than the one provided by Nauty. We are not convinced that this change would lead to a better performance: if the number of

```
     Find Paths (A path $P$, a set of legs $L$)
(1)  for each allowable refinement leg $l$ in $L$ do
(2)      $G'$:= $l$.refinement applied to $P$
(3)      if $l$.refinement is a node refinement do
(4)          $L'$:= extend $(l) \cup \{\text{join}(l,l') \mid l' \neq l \in L\}$
(5)          if $G'$ is a path then Find Paths ( $G'$, $L'$ )
(6)          else Find Trees ($G'$, $L'$)
(7)      else $L' := L' \cup \{\text{join}(l,l') \mid l' \neq l \in L\}$
(8)             Find Cyclic Graphs ($G'$, $L'$)

     Find Trees (A tree $T$, a set of legs $L$)
(9)  for each allowable refinement leg $l$ in $L$ do
(10)     $G'$:= $l$.refinement applied to $T$
(11)     if $l$.refinement is a node refinement do
(12)         $L'$:= restricted-extend ($l$)
(13)         $L' := L' \cup \{\text{join}(l,l') \mid l' \in L, l'$ allowable in $G'\}$
(14)         Find Trees ($G'$, $L'$)
(15)     else $L' := L' \cup \{\text{join}(l,l') \mid l' \neq l \in L\}$
(16)            Find Cyclic Graphs ($G'$, $L'$)

     Find Cyclic Graphs (A graph $G$, a set of legs $L$)
(17) for each allowable refinement leg $l$ in $L$ do
(18)     $G'$:= $l$.refinement applied to $G$
(19)     $L' := L' \cup \{\text{join}(l,l') \mid l' > l \in L\}$
(20)     Find Cyclic Graphs ($G'$, $L'$)

     Join (legs $l_1$ and $l_2$)
(21) $l'$.refinement := $l_2$.refinement
(22) $l'$.embeddinglist := $\{(k, t_j.\text{graph}, t_j.\text{node}) \mid$
(23)        $t_k \in l_1$.embeddinglist, $t_j \in l_2$.embeddinglist,
(24)        $t_k.\text{parent} = t_j.\text{parent}\ \}$
(25) if $l'$ is frequent then return $l'$ else return $\emptyset$

     Extend (leg $l$)
(26) $\mathcal{C}$: set of candidate legs
(27) for each $t_k \in l$.embeddinglist do
(28)     for each adjacent node $v'$ of $t$.node in $t$.graph do
(29)         if $v'$ occurs in the embedding of $t_k$ do
(30)             append index $k$ to $c$.embeddinglist of the
(31)                 corresponding cycle closing leg $c \in \mathcal{L}$
(32)         else append $(k, v', t.\text{graph})$ to $c$.embeddinglist of the
(33)                 corresponding node refinement leg $c \in \mathcal{L}$
(34) return $\{c \mid c \in \mathcal{C}, c$ is frequent$\}$
```

Figure 6: Outline of the Gaston algorithm

PTE: Runtimes in seconds

MinSup %	2%	3%	4%	5%	6%	7%	8%	9%	10%	20%	30%
MinSup Abs.	7	10	14	17	20	24	27	31	34	68	102
#freq. graphs	136949	22758	5935	3608	2326	1770	1323	977	844	190	68
#freq. trees	119378	20481	5514	3376	2172	1644	1230	909	779	177	62
gSpan	98.0	20.3	6.3	3.4	2.0	1.4	1.0	0.8	0.6	0.3	0.2
FFSM	-	>6.4	>1.6	>0.8	-	-	-	-	-	-	-
	-	<11.3	<3.1	<1.9	-	-	-	-	-	-	-
Gaston (FreeTrees)	8.1	1.9	0.8	0.5	0.4	0.3	0.3	0.3	0.2	0.2	0.1
Gaston (NoIsomorph.)	10.0	2.4	0.9	0.6	0.4	0.3	0.3	0.3	0.3	0.2	0.1
Gaston (Nauty)	14.2	2.7	0.9	0.6	0.4	0.4	0.3	0.3	0.3	0.2	0.2
FSG	307.4	43.9	11.0	6.3	4.0	2.9	2.4	1.8	1.6	0.6	0.3
FARMER	-	572	172	93	54	37	28	20	17	6	4
WARMR	-	-	-	-	-	-	-	-	11351	733	172

Figure 7: Results for the PTE dataset

Name	Contents
S5	10.000 artificial trees [22]
L10	100.000 artificial trees [22]
Multicast	1.000 multicast trees [3]
DTP	421 molecules [21]
PTE	340 molecules [21]
CAN2DA99	32.557 molecules [14]
AID2DA99	42.689 molecules [14]
NCI	250.251 molecules [14]

Figure 8: Summary of datasets

graphs with cycles is large, within our approach a larger set of redundant legs has to be evaluated; in the other approach, we would have to rely heavily on a special-purpose normalisation procedure which is most likely not as as efficient as that of Nauty.

The setup that we presented in this paper is not only the description of one algorithm, it is also the proof of a concept. The "quickstart" idea can also be applied further: although in this paper, trees are refined into general graphs, one can imagine several phases in between trees and general graphs, for example phases in which *planar* and *outerplanar* graphs are grown first. Also for planar and outerplanar graphs polynomial normalisation procedures exist, just like for free trees. Using the quickstart principle, the part of the partial order which is enumerated using exponential algorithm can be narrowed down to a very tiny part of all possible frequent structures.

6. EXPERIMENTAL RESULTS

An overview of the results of our experiments can be found in Figures 7, 9 and 10, a description of the datasets in Table 8. Unless noted otherwise, all experiments were performed on an Athlon XP1600+ with 512MB main memory, running Mandrake Linux 9.1; the algorithm was implemented in C++ using the STL and compiled with the -O3 compilation flag. We tried to compare our algorithm with a wide range of functionally comparable frequent structure miners. The graph miners gSpan and FSG, and the free tree miner FTM [Rückert] [19]) were provided to us as binaries. FTM [Chi] [3]) was provided as source code and compiled under exactly the same circumstances as our algorithm. [1] Some of the binaries provided to us had restrictions regarding the number of labels or were restricted to molecular databases. For these algorithms we only publish limited results.

As our algorithm contains a specialized free tree mining procedure, our first experiment is intended to determine its performance on tree shaped datasets. The S5 and L10 datasets were obtained using a slightly modified version of an artificial dataset generator kindly provided by Mohammed Zaki. To a certain extent, the artificial datasets mimic datasets that could be constructed from webserver access logs [22]. S5 is a small dataset obtained by sampling 10k trees of maximal depth 5 from a master tree of 10k nodes with 3 node labels and fan-out 20. L10 is larger and obtained by sampling 100k trees of maximal depth 10 from a master tree of 10k nodes with 3 node labels and fan-out 20. On S5 our algorithm is 6 times faster for support 0.5% than the second best algorithm; for L10 our algorithm is 7 times faster for support 3%. The Multicast dataset was provided to us by Yun Chi [3]. Although our algorithm keeps a good performance downto a support of 70%, for lower supports its performance degrades. This is due to the dense structure of this dataset. Closed structure mining are reported to obtain

[1]Note to the reviewers: much to our regret, our attempts to obtain the FFSM graph miner before the SIGKDD deadline have failed. This is also a disappointment to us, as the comparison with FFSM would be the most interesting one. For one dataset (PTE) we tried to extrapolate the performance of FFSM as reported in [7] to our algorithm. We however acknowledge that this is not good practice. More details can be found on `http://www.liacs.nl/home/snijssen/farmer/results.html`.

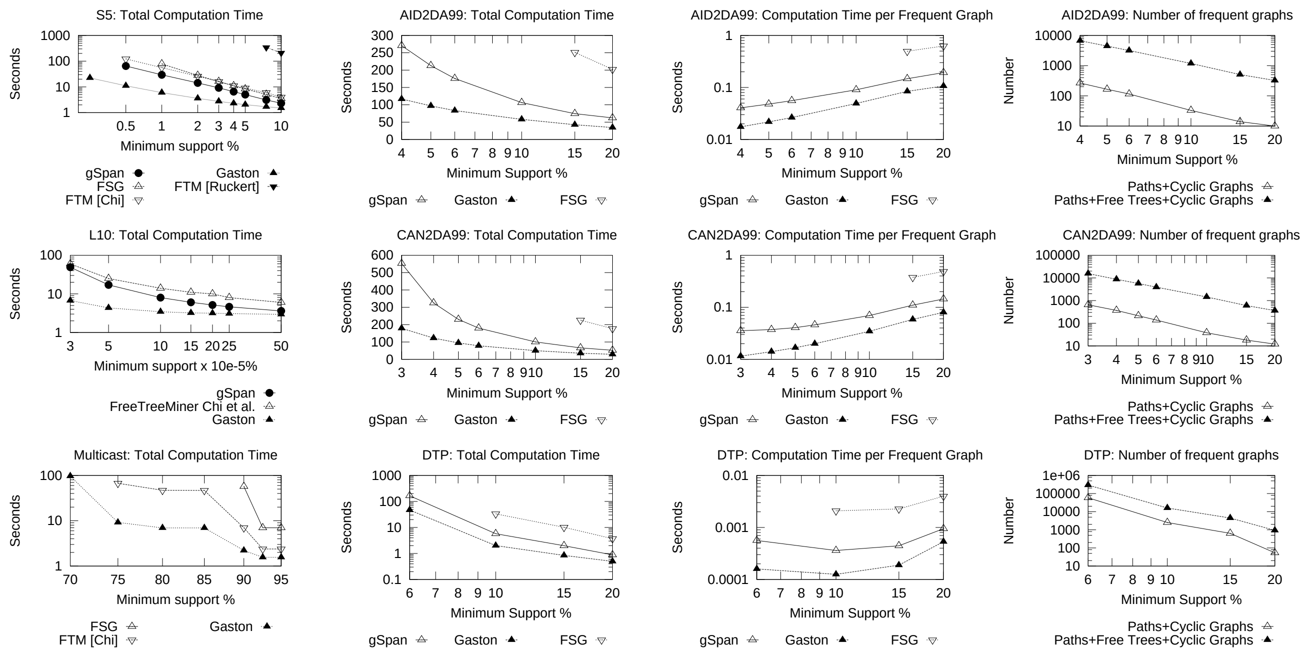


Figure 9: Results of our experiments

better performance [3].

Our remaining experiments regard molecular databases. The first dataset which we consider is the Predictive Toxicology dataset (PTE) which can be downloaded from `http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/PTE/`. The dataset was first used in an experiment with WARMR [4] in 1998. We transform the dataset into graphs using the procedure given in [21]. The table in Figure 7 gives a detailed insight into the performance of several algorithms on this dataset. Both FARMER and WARMR are multi-relational data mining algorithms which are not really suited for the mining tasks discussed here. We run GASTON in three different setups. In the first setup, GASTON Free Trees, we disable the cycle closing phase of GASTON such that only frequent free trees are discovered. In the second setup, GASTON No Isomorphism, we disable the Nauty isomorphism chech such that the same cycled graph may be generated multiple times. In the third setup Nauty is enabled. Our experiments reveal that in this small dataset, the cost of evaluating a redundant set of graphs using the embedding lists is actually cheaper than removing those redundant graphs from the search.

The other datasets were obtained from the National Cancer Institute [14]. In the DTP AIDS program 42.689 compounds (dataset AID2DA99) were classified into three classes: CI (confirmed inactive), CM (confirmed moderately active) and CA (confirmed active). The latter class consists of 422 molecules (dataset DTP). For these datasets the speedup of GASTON in comparison with gSpan is lower, 2 to 3 times. Similarly, in the DTP Human Tumor Cell Line Screen 32.557 compounds were classified (dataset CAN2DA99). Also here GASTON improves on gSpan. Our statement that free trees constitute a significant part of the results of graph mining algorithms, is confirmed by the experiments. In most experiments, 90% of the frequent structures are free trees, which justifies our choice to concentrate on efficient free tree discovery. Experiments on similar datasets were reported in [7, 21].

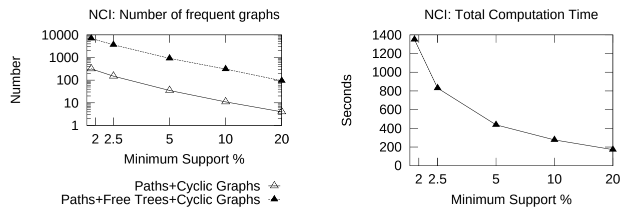


Figure 10: Results for frequency mining on the NCI

To test the scale-up properties of our algorithm, we have also run our algorithm on the database of all 250.251 compounds in the NCI'99 release. In this transformation, some atom types were subdivided into several classes according to their position in the molecule to obtain labels. The main disadvantage of our algorithm revealed itself here, as we had to resort to a computer with more main memory than 512MB. Results in Figure 10 for the NCI database were obtained on a Sun Enterprise Server with 4 processors of 400Mhz and 4GB main memory. As far as we are aware of, no other comparable results have yet been published for the whole NCI'99 database.

To exploit the classification of compounds into three classes in CAN2DA99 and AID2DA99, in [10, 18, 19, 6] it was proposed to use an approach using *version spaces*. Algorithms are proposed which output only those frequent molecules in the active part of the dataset which have a low support in the inactive part, as these structures may be a good classifier for the activity of a compound. We modified our algorithm to allow for similar experimental results which involve two datasets and have investigated the idea of *emerging patterns* [5], which is closely related. If one assumes that

Minimum Support	Run time	Frequent graphs with $> 10\%$ support difference
15%	246.67s	3637
10%	295.77s	12283
5%	596.21s	12751

Table 1: The difference between AID2DA99-active and all compounds in NCI'99

the entire NCI database is representative for a broad range of molecules, it can be interesting to discover which submolecules of a database of active compounds have a support in the active dataset which is significantly *different* from the support in the entire NCI database. We performed this experiment for known active compounds of AID2DA99. For a minimum difference threshold of 10%, results are summarized in table 1. Most time is spent to evaluate the support of a graph in the NCI database. [2]

7. CONCLUSIONS

In this paper, we introduced GASTON, a new efficient frequent graph mining algorithm. We observed that most frequent substructures in practical graph databases are actually free trees, and used this observation to implement an algorithm that "quickstarts" its search by using a highly efficient enumeration strategy for enumerating the frequent free trees first. Experiments confirm that our algorithm is competitive with existing frequent graph miners. To show that our algorithm also scales up to large databases, we performed experiments on the entire NCI'99 database. An extension of our algorithm which makes it possible to specify maximum frequency constraints or frequency difference constraints was successfully shown.

Acknowledgements

We are grateful to Yun Chi for discussions, for providing the Multicast dataset and for providing the source code of his free tree mining algorithm, which has been a source of inspiration to us. Furthermore, we like to thank Xifeng Yan for providing a binary of gSpan, Ulrich Rückert for providing a binary of his FreeTreeMiner, Hendrik Blockeel for providing the binary of WARMR, Mohammed Zaki for providing the tree data set generator and Michihiro Kuramochi for pointing us to his binary of FSG.

8. REFERENCES

[1] K. Abe, S. Kawasoe, T. Asai, H. Arimura, and S. Arikawa. Optimized substructure discovery for semi-structured data. In *Proceedings of 6th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD2002)*, pages 1–14, 2002.

[2] T. Asai, H. Arimura, T. Uno, and S. Nakano. Discovering frequent substructures in large unordered trees. *Technical Report University of Kyushuu*, (216), 2003.

[3] Y. Chi, Y. Yang, and R. R. Muntz. Mining frequent rooted trees and free trees using canonical forms. *UCLA Computer Science Department Technical Report*, 2004.

[4] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining (KDD1998)*, pages 30–36, 1998.

[5] G. Dong, J. Li, and X. Zhang. Discovering jumping emerging patterns and experiments on real datasets. In *Proceedings of the fifth International Conference on Knowledge Discovery and Data Mining*, pages 43–52, 1999.

[6] H. Hofer, C. Borgelt, and M. R. Berthold. Large scale mining of molecular fragments with wildcards. In *Advances in Intelligent Data Analysis V*, pages 380–389, 2003.

[7] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the 2003 International Conference on Data Mining (ICDM2003)*, 2003.

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

[9] A. Inokuchi, T. Washio, K. Nishimura, and H. Motoda. A fast algorithm for mining frequent connected subgraphs. In *IBM Research, Tokyo Research Laboratory*, page 10, 2002.

[10] S. Kramer, L. D. Raedt, and C. Helma. Molecular feature mining in hiv data. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD2001)*, 2001.

[11] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the 2001 International Conference on Data Mining (ICDM2001)*, 2001.

[12] B. D. McKay. Practical graph isomorphism. 30:45–87, 1981.

[13] S. Nakano and T. Uno. A simple constant time enumeration algorithm for free trees. In *IPSJ SIGNotes ALgorithms*, number 091 - 002, 2003.

[14] National Cancer Institute (NCI). Dtp/2d and 3d structural information, `http://cactus.nci.nih.gov/ncidb2/download.html`. 1999.

[15] S. Nijssen and J. N. Kok. Efficient discovery of frequent unordered trees. In *First International Workshop on Mining Graphs, Trees and Sequences*, 2003.

[16] S. Nijssen and J. N. Kok. Efficient discovery of frequent unordered trees: Proofs. *Leiden Institute of Advanced Computer Science Technical Report*, (1), 2003.

[17] S. Nijssen and J. N. Kok. Efficient frequent query discovery in farmer. In *Principles of Knowledge Discovery and Data Mining 2003 (PKDD2003)*, 2003.

[18] 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 (IJCAI2001)*, 2001.

[19] U. Rückert and S. Kramer. Frequent free tree

[2]As a side-note: for a support of 5% on the same database of active compounds, FSG requires 1300s to find all frequent structures in the set of active compounds *only*.

discovery in graph data. In *Special Track on Data Mining, ACM Symposium on Applied Computing (SAC2004)*, 2004.

[20] R. A. Wright, B. Richmond, A. Odzlyzko, and B. D. McKay. Constant time generation of free trees. 15(2), 1986.

[21] X. Yan and J. Han. Closegraph: Mining closed frequent graph patterns. In *Proceedings of the 2003 Conference on Knowledge Discovery and Data Mining (SIGKDD2003)*, 2003.

[22] M. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the SIGKDD 2002*, 2002.