The Gaston Tool for Frequent Subgraph Mining

Siegfried Nijssen and Joost N. Kok

LIACS, Universiteit Leiden, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
snijssen@liacs.nl

Abstract

Given a database of graphs, structure mining algorithms search for all substructures that satisfy constraints such as minimum frequency, minimum confidence, minimum interest and maximum frequency. In order to make frequent subgraph mining more efficient, we propose to search with steps of increasing complexity. We present the GrAph/Sequence/Tree extractiON (Gaston) tool that implements this idea by searching first for frequent paths, then frequent free trees and finally cyclic graphs. We give results on large molecular databases.

Key words: Frequent Subgraphs, Data Mining

1 Introduction

In recent years data mining of structures such as graphs, trees, molecules, XML documents and relational databases has attracted a lot of research. Especially the idea of discovering all frequent substructures has recently led to a large number of specialized algorithms for mining paths, trees and graphs. Frequent substructures give interesting information about the database. This information can be used in many different ways, for example for classification. As an example, Figure 1 shows molecular fragments that are frequent in an HIV inhibitor database, but not frequent in another database, thus providing features that distinguish between the two databases; the top leftmost fragment is a major constituent of AZT, a nucleoside reverse transcriptase inhibitor, and is used in many drugs for anti-HIV treatment. Our tool computes these fragments completely automatically.

Experiments with 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 efficient algorithms for them exist. Therefore, we integrate a frequent path, tree and graph miner into one tool called Gaston in order to gain efficiency. The main challenge in the development of the Gaston tool is how to split up the discovery process

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

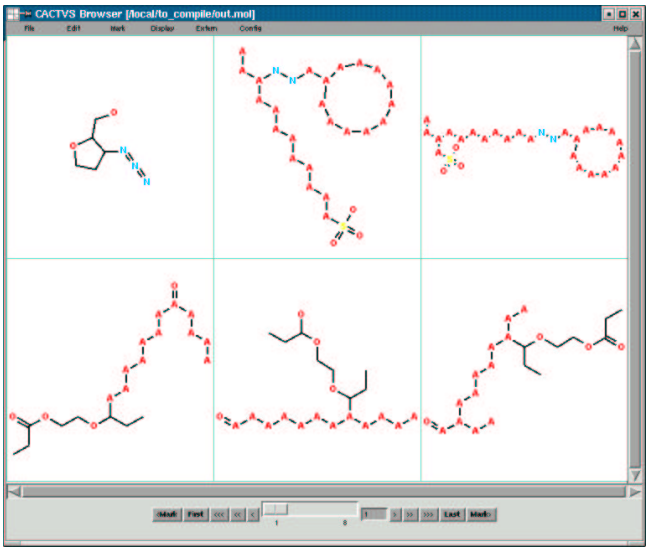

Fig. 1.

into several phases. Ideally, the tool should behave like a specialized free tree miner when faced with free tree databases, but should also be able to deal with graph databases efficiently.

Further references, overview papers and the source code of our tool can be obtained from our homepage for frequent structure mining `http://hms.liacs.nl/`.

2 Foundations

We will only briefly discuss the mathematical preliminaries: the definitions are similar to those used in other papers concerning frequent structure mining, for example [2,13,14,15]. A labeled graph G consists of a finite 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 from $\mathcal{L}$ to all edges and nodes. We only consider undirected graphs, i.e., (v_1, v_2) is the same edge as (v_2, v_1). When graph G is subgraph isomorph with graph G', we denote this with $G \subseteq G'$.

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 $\mathit{freq}(G, \mathcal{D}) = \#\{G' \in \mathcal{D} | G \subseteq G'\}$, the support of a graph is given by $\mathit{support}(G, \mathcal{D}) = \mathit{freq}(G, \mathcal{D})/|\mathcal{D}|$. The task is to find *all* graphs for which $\mathit{support}(G, D) \geq \mathit{minsup}$, for some predefined threshold minsup that is specified by the user. An important property that holds is that $G_1 \subseteq G_2$ implies that $\mathit{freq}(G_1, \mathcal{D}) \geq \mathit{freq}(G_2, \mathcal{D})$. A consequence of this property is that any (large) graph which contains a (smaller) graph which is not frequent, cannot be frequent too. This *Apriori* property is the basis on which many frequent structure mining algorithms have been built. The process of removing graphs from the search space using this property is called (frequency based) pruning.

If for two connected graphs $G_1 \subset G_2$ there is no G_3 with $G_1 \subset G_3 \subset G_2$, we call G_2 a refinement of G_1. One can show that G_2 is a refinement of G_1

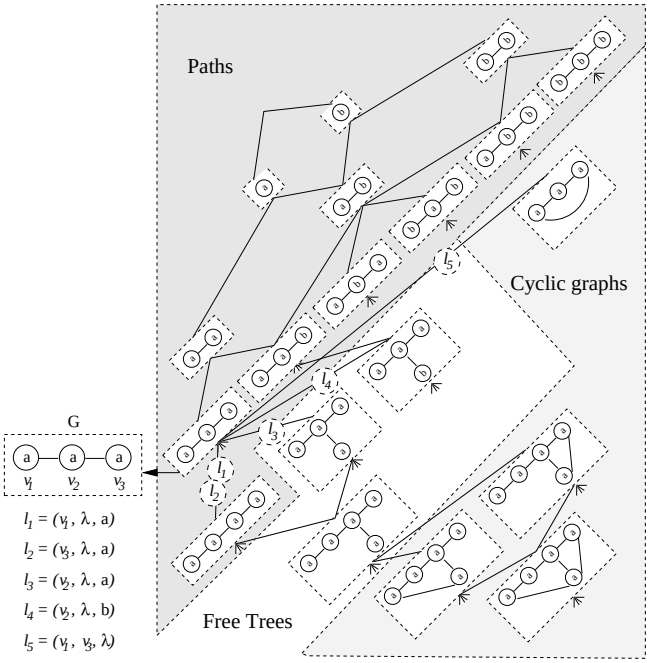

Fig. 2. Refinement of graphs

only if there is a $G_3 \equiv G_2$ with G_3 obtained from G_1 by one of the following two operations:

- node refinement: to G_1 a new node is added, this node is connected to a node of G_1 by a new edge (sometimes also called a *forward edge*);
- cycle closing refinement: to G_1 a new edge is added between nodes that were already connected by a path (sometimes also called a *backward edge*).

Extending terminology from [2] to cyclic graphs, we call an operation to refine a graph a *leg*. A node refining leg for a graph $G = (V, E, \ell)$ consists of a node in V, an edge label and a node label. Examples of node refining legs are l_1, l_2, l_3 and l_4 in Figure 2. A cycle closing refining leg consists of two different nodes in V and an edge label. An example is leg l_5 in Figure 2. The refinement operator $\rho(G, l)$ refines a graph G by adding leg l. Edges between graphs in Figure 2 correspond to refinement steps. Note that for legs l_1 and l_3 of graph G, l_3 is also a leg of $\rho(G, l_1)$ and l_1 is a leg of $\rho(G, l_3)$. Indeed, the only legs of $\rho(G, l_1)$ which are not legs of G are legs that connect to the node that is introduced in $\rho(G, l_1)$ by l_1. Furthermore note that two legs, l_1 and l_2, refine graph G to isomorph graphs due to automorphisms in G. One can show that for every graph an isomorphic graph can be constructed using a sequence of node refinements followed by a sequence of closing refinements.

An overview of a depth-first graph mining algorithm is given in Figure 3. To represent graphs *codes* are used. A graph code is a string that unambiguously defines a series of refinement steps that lead to a certain graph. The k-prefix of a graph code is the code which contains only the first k refinement steps defined by the code. Different codes have been proposed, among which

DFGraphMiner (A graph code C, a leg l, a set of legs L)

(1) C':= $\rho(C, l)$

(2) **if** C' is not canonical **then return**

(3) output graph C'

(4) $L' := \{l' | l'$ is a necessary leg of $C', support(\rho(C', l'), D) \geq minsup,$
$l' \in L$, or l' connects to the node introduced by l,
if l is a node refinement $\}$

(5) **for all** $l' \in L'$ **do** DFGraphMiner (C', l', L')

Fig. 3. Depth-first graph mining algorithm

DFS codes (in gSpan [15]), adjacency matrices (in FFSM [6]) and tree codes with backtrack symbols (in FreeTreeMiner [2] and TreeMiner [16]). In order to make sure that no two isomorphic graphs are emitted by the algorithm, in line (2) it is determined whether the current code C', which corresponds to a graph G, is the lowest (or highest) code among all possible codes for graph G; thus the graph mining algorithm only outputs graphs in a *canonical code*. If its code is not canonical, a graph is not further refined. To guarantee that the search can still potentially consider all possible graphs, the graph codes used in graph miners should therefore have the property that every k-prefix of a canonical code is also canonical.

To limit the number of legs that an algorithm has to consider, most depth first miners constantly maintain a set of feasible legs. Exploiting the principle of frequency based pruning, once a leg is found to be infrequent, this leg should not be considered as a leg in any refined graph. Therefore, in line (4) only legs are considered which were legs of the previous graph, or which connect to the node that was last added to the graph. In order to obtain all possible legs that can be added to a new node, it is necessary to pass the graph C' through the database and to compute all its occurrences. For each occurrence the legs of the new node should be determined.

A condition which is not needed for the correctness of the algorithm, but which is of vital importance for its performance, is the first condition of line (4), which states that only *necessary* legs should be evaluated and added to L'. A leg is necessary if among all descendants of the current graph code C', there is at least one canonical graph code which can only be obtained by applying the refinement defined by that leg. Ideally this condition would be computable in constant time and the code $\rho(C', l)$ obtained by applying each necessary leg l immediately is also canonical: in that case one could remove the test in line (2) and the support of every graph would be computed exactly once.

By dividing the search into three phases, we avoid the check of line (2) in

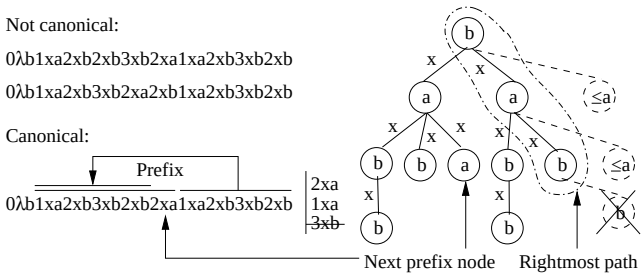

Fig. 4. An unordered rooted tree, with some of its depth sequences and legs

many cases. More precisely, we model free trees such that one can determine the necessary legs in constant time. Moreover, all these legs are canonical refinements.

3 Enumeration

In order of increasing complexity we will introduce the canonical strings that we use for each class of structures.

Paths The main problem of path enumeration is that a path can have two orientations, for example: *axaxb* and *bxaxa*. Each path has two potential predecessors which can be obtained by removing each of the endpoints. Of the two paths such obtained, we consider only the codes that start at the opposite side of the removed nodes. Then, we define that the lexicographically lowest string among these 2 codes is the unique predecessor of the code. Unfortunately, however, we will see that each leg of a path is a necessary leg for the free tree mining phase. No pre-pruning of legs in line (4) is therefore possible; some paths will thus be evaluated twice. In line (2) a path is considered to be canonical if it grows from its predecessor code. If this predecessor is symmetric, and a similar leg can therefore be added to both endpoints and there is no structural reason to prefer one endpoint above the other, only the leg is canonical which connects to the node that was added most recently.

Free trees The free tree code that we define here is based on a code for rooted trees that was independently proposed by [1] and [12], and has strong similarities with a method proposed in [10] for unlabeled free trees. Every rooted tree can be encoded with a *depth sequence*, of which examples are given in Figure 4. A depth sequence for a tree is obtained by performing a preorder depth first walk; each time that a node is visited for the first time, first its depth is emitted, then the label of the edge going into that node and finally the label of that node; we call this combination a depth sequence tuple. Clearly, the depth sequence depends on the order in which the children of a node in a tree are visited. For an unordered, rooted tree, the *canonical* depth sequence is defined as the depth sequence that is the lexicographically highest sequence among all possible depth sequences for that tree.

As refinements for an unordered rooted tree it suffices to only consider

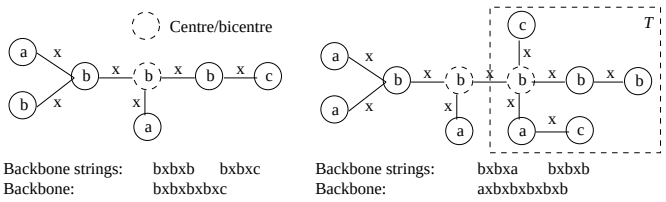

Fig. 5. Free trees, (bi)centres and backbones

legs that connect to the rightmost path of the tree, as illustrated in Figure 4. Whether a leg may be connected to the rightmost path can be determined in constant time using the *next prefix node* and the *left sibling* of the new node. For every leg of the rightmost path there is a corresponding depth tuple that would be appended after the depth sequence. This new tuple may not be higher than the tuple of the next prefix node, which is defined by considering the largest suffix of the depth sequence that is a prefix of a corresponding left sibling subtree. Furthermore the new node may not have a higher label than its left sibling. One can show that by only considering legs that immediately yield a canonical depth sequence, no legs are discarded that are needed as refinement later. For example, if a leg is added after several other refinement steps, its node label must still be lower than or equal to that of its left sibling. One can therefore characterize necessary legs efficiently.

To employ these principles in free tree enumeration, we use the following setup. First, for each free tree we define one path predecessor, as follows. A well-known property of free trees is that one can point out a (bi)-centre; if any longest path in the tree has odd length, the free tree has a bicentre consisting of the two middle nodes on this path; otherwise the tree has a centre (see Figure 5). A centred tree can be conceived as a single rooted tree, a bicentred tree can be seen as two separate rooted trees of which the roots are interconnected. Now consider all oriented paths of maximal length that start in the root of (each) tree. From each of these maximal paths a code can be obtained consisting of the labels on the nodes and edges. In centred trees, those two path codes which are lexicographically the highest and occur in paths that only have the root in common, are called the backbone strings of the free tree. In bicentred trees, the lexicographically highest path codes in each of the two rooted trees are defined to be the backbone strings of that rooted tree. By concatenating the reverse of one backbone string with the other backbone string, a single path is obtained which we call the *backbone* of the free tree. We arrange our procedure such that this path is the predecessor of the free tree.

To refine a given path we use the following idea. First, the path is split into two parts by removing the edge between the middle two nodes (in case of odd length paths) or by removing the single middle node (in even length paths). Each of the resulting paths is a rooted tree. Using the principle of depth sequences, rooted trees are grown for each of these initial rooted trees; by finally combining the rooted trees again, a free tree is obtained. To guarantee

that a free tree grows only from its backbone path, no refinement of each rooted tree is allowed which would result in a different backbone in the final free tree.

Cyclic graphs To strictly divide the frequent graph discovery process into phases, we only consider the cycle closing refinements in the very last phase. All cycle closing refinements connect two existing nodes in a tree; within our setup, during the first two phases constantly the set of all frequent closings is maintained. Once such a closing is applied, the tree becomes a cyclic graph. We define a code for cyclic graphs by concatenating two separate codes. The first code consists of the depth-sequence corresponding to a free tree. This free tree is a spanning tree of the graph. Each tuple in the sequence introduces a new node in the free tree; the nodes of the tree can be numbered by their occurrence order in this sequence. The second part of the code consists of a sequence of tuples of the form (v_i, v_j, ℓ); each such tuple defines two nodes $v_i < v_j$ that are connected with an edge.

The necessary legs for cyclic graphs are now obtained as follows. First, all node refining legs are discarded, to make sure that cyclic graphs only grow from spanning trees. Then, all tuples which sort lower than the closing leg last added, are discarded. The canonical code for a cyclic graph is given by the concatenated code that sorts the lowest among all possible codes for that graph.

4 Graph Counting

In the Gaston tool we can use two alternatives for graph counting:

Embedding lists (EL) For graphs with a single node we store an embedding list of all occurrences of its label in the database. For other graphs a list is stored of embedding tuples that consist of (1) an index of an embedding tuple in the embedding list of the predecessor graph and (2) the identifier of a graph in the database and a node in that graph. 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 complete embedding information for a structure can be obtained by scanning its embedding list, and by following the predecessor pointers. The frequency of a structure is determined from the number of different graphs in its embedding list. An example is provided in Figure 4. For each leg of a graph an embedding list is constructed. Graph G in the example has two legs with corresponding graphs and embedding lists. When a graph is canonically refined, embedding lists for the legs of the new graph are computed as much as possible using a list join operation that joins embedding lists of two earlier legs. New embedding lists are constructed for legs that were not present in the predecessor graph. Clearly, this approach requires a lot of main memory: not only for the current graph embedding lists for all legs have to be stored, but due to the backtracking procedure, embed-

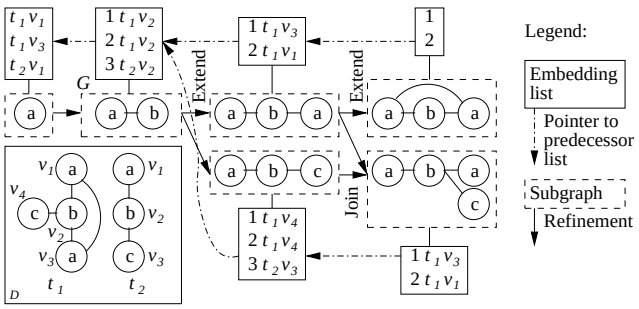

Fig. 6. Maintenance of embedding lists

Name	Contents
L10	100,000 artificial trees [16]
CAN2DA99	32,557 molecules [11]
AID2DA99	42,689 molecules [11]
NCI	250,251 molecules [11]

Fig. 7. Summary of data sets

ding lists for legs of predecessor structures must also be stored. Embedding lists are quick, but they do not scale very well to large databases.

Recomputed embeddings (RE) Our other approach is based on maintaining a set of "active" graphs in which occurrences are repeatedly recomputed. As subgraph isomorphism is NP complete [4], essentially a backtracking procedure is required. We found that several techniques can increase the performance. For each graph we first compute a strategy comparable to a query evaluation plan. In general a breadth-first walk of a graph turns out to perform better than a depth-first walk. Furthermore, with each node in the database, we reserve space in which the graph miner can store *hints*. One such hint is whether a node is part of some occurrence of a predecessor structure.

5 Experiments

An overview of results of experiments with our tool can be found in Figure 8 and a description of the data sets in Figure 7. Unless noted otherwise, all experiments were performed on an Athlon XP1600+ with 512MB main memory, running Mandrake Linux 10; the algorithm was implemented in C++ using the STL and compiled with the –O3 compilation flag. We compare our tool with a broad range of other mining tools: the graph miners gSpan, FSG and the FTM free tree miner of Chi [2]. All miners were obtained from their original authors. For the free tree mining experiments we have used a modified version of a data set generator that generates data sets mimicking webserver access logs as described in [16]. Set L10 is obtained by sampling 100k trees

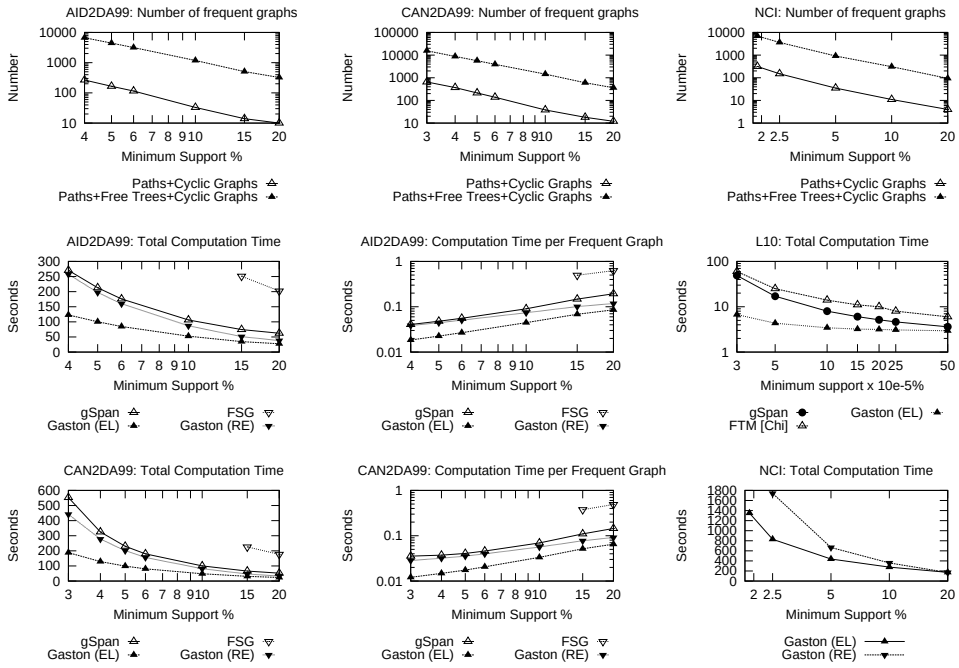

Fig. 8. Results of our experiments

of maximal depth 10 from a master tree of 10k nodes with 3 node labels and fan-out 20. Our other experiments regard large molecular databases. We transform these data sets into graphs using the procedure given in [15]. These data sets were obtained from the National Cancer Institute [11]. To test the scale-up properties of our tool, we have run it on the database of all 250,251 compounds in the NCI'99 release. Here, we subdivided some atom types into classes according to their position in the molecule to obtain labels. To run the algorithm based on embedding lists, we used a Sun Enterprise Server with 4 processors of 400Mhz and 4GB main memory; Gaston (EL) required 1.7GB memory, Gaston (RE) 150MB.

To exploit the activity information that is available for compounds in the CAN2DA99 and AID2DA99 data sets, in [5,13,14] it was proposed to use *version spaces*. The idea is to only output molecular fragments that are frequent in the active part of a dataset, and to discard fragments which are also frequent in the inactive part. Using the assumption that the entire NCI database is representative for a broad range of molecules, we are interested in discovering frequent fragments of active compounds that have a significantly different support in the total NCI database. We performed this experiment for known active compounds of AID2DA99. Results are summarized in Figure 9.

6 Conclusions

The Gaston tool finds all frequent substructures in tree and graph databases. It is an interesting tool with very competitive running times, especially for

Minimum Support	Run time	Frequent graphs with $> 10\%$ difference in support
15%	246.67s	3,637
10%	295.77s	12,283
5%	596.21s	12,751

Fig. 9. Differences between AID2DA99-active and compounds in NCI'99

large molecular databases. Behind the tool is an innovative algorithm, which finds the frequent substructures in a number of phases of increasing complexity. Future work on the tool includes the intelligent pruning of the results.

References

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

[2] Y. Chi, Y. Yang, R. R. Muntz. HybridTreeMiner: An Efficient Algorithm for Mining Frequent Rooted Trees and Free Trees Using Canonical Forms. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2004.

[3] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In *Proceedings of the SIGKDD*, pages 30–36, 1998.

[4] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*, 1979.

[5] 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.

[6] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the ICDM*, 2003.

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

[8] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the ICDM*, pages 313–320, 2001.

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

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

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

[12] S. Nijssen and J. N. Kok. Efficient discovery of frequent unordered trees. In *First International Workshop on Mining Graphs, Trees and Sequences*, pages 55–64, 2003.

[13] L. D. Raedt and S. Kramer. The level-wise version space algorithm and its application to molecular fragment finding. In *Proceedings of the Seventeenth IJCAI*, pages 853–859, 2001.

[14] U. Rückert and S. Kramer. Frequent free tree discovery in graph data. In *Special Track on Data Mining, ACM Symposium on Applied Computing*, pages 564–570, 2004.

[15] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proceedings of the SIGKDD*, pages 286–295, 2003.

[16] M. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the SIGKDD*, pages 71–80, 2002.