

A Quickstart in Frequent Structure Mining Can Make a Difference

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

Joost N. Kok
LIACS, Universiteit Leiden
Niels Bohrweg 1, 2333 CA Leiden
The Netherlands
joost@liacs.nl

ABSTRACT

Given a database, structure mining algorithms search for substructures 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 investigate the use of the “quickstart principle”, which is based on the fact that these classes of structures are contained in each other, thus allowing for the development of structure mining algorithms that split the search into steps of increasing complexity. We introduce the GrAph/SequeNce/Tree extractiON (GASTON) algorithm that implements this idea by searching first for frequent paths, then frequent free trees and finally cyclic graphs. We investigate two alternatives for computing the frequency of structures and present experimental results to relate these alternatives.

Categories and Subject Descriptors: H.2.8 [Database Management]: Database Applications—*Data Mining*

General Terms: Algorithms, Performance, Theory

Keywords: Frequent Item Sets, Graphs, Semi-Structures, Structures

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 research. Especially the idea of discovering frequent substructures of such databases has recently led to a large number of specialized 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 the 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 sim-

pler structures than general, cyclic graphs, and 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 specialized free tree miner when faced with free tree databases, but should also be able to deal with graph 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: on a common benchmark dataset, we obtain up to ten-fold speed ups in comparison with other algorithms. An important part of any frequent structure mining algorithm is its strategy for determining the support of a substructure. We consider two different strategies for computing the support of structures and introduce optimizations for these strategies. As molecular databases are currently the foremost application area for graph mining algorithms, we provide extensive experimental results for such databases.

As background knowledge for our paper we could mention a large number of publications. We will only refer to recent ones here. Due to space limitations we also omit many details of our own algorithm. More details, references, and the source code of our algorithm can be obtained from our homepage for frequent structure mining, <http://hms.liacs.nl/>.

The overview of the rest of the paper is as follows: first we give preliminaries, then we discuss enumeration strategies, frequency evaluation and finally we give experimental results.

2. PRELIMINARIES

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, 5, 7, 12, 13, 14]. 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) . 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 $v_1, v_2, \dots, v_m \in V$ is a path if for all $i \in \{1, 2, \dots, m-1\}$ edge (v_i, v_{i+1}) is in E .

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

KDD'04, August 22–25, 2004, Seattle, Washington, USA.
Copyright 2004 ACM 1-58113-888-1/04/0008 ...\$5.00.

Research Track Poster

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, the graph is connected. 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 $f : V_1 \rightarrow V_2$ 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, denoted by $G_1 \equiv G_2$. $G_1 \subset G_2$ is a shorthand for $G_1 \subseteq G_2 \wedge G_1 \not\equiv G_2$. An isomorphism of a graph to itself is called an automorphism.

Three special subclasses of graphs are *paths*, *free trees* and *rooted trees*. Paths are graphs in which two nodes have degree 1, while all other nodes have degree 2. If a graph has no cycles, the graph is called a free tree. A rooted tree is a free tree in which one node, the root, is singled out. The depth of a node is the length of the path from that node to the root. To avoid confusion we will always speak of either free trees or rooted trees. In a rooted tree we draw the root of the tree as the top node. Examples are given in Figure 1.

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 and the tree is furthermore a subgraph 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 given by $support(G, \mathcal{D}) = freq(G, \mathcal{D})/|\mathcal{D}|$. The primary task that our algorithm deals with is to find all graphs for which $support(G, \mathcal{D}) \geq minsup$, for some pre-defined threshold *minsup* that is specified by the user. An important property that holds is that $G_1 \subseteq G_2$ implies that $freq(G_1, \mathcal{D}) \geq freq(G_2, \mathcal{D})$. A consequence of this property is that any (large) graph which contains a (smaller) graph which is not frequent, cannot be frequent too. This *A priori* 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 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 1. A cycle closing refining leg consists of two different nodes in V and an edge label. An example is leg l_5 in Figure 1. The refinement operator $\rho(G, l)$ refines a graph G by adding leg l . Edges between graphs in Figure 1 correspond to refinement steps. Note that for legs l_1 and

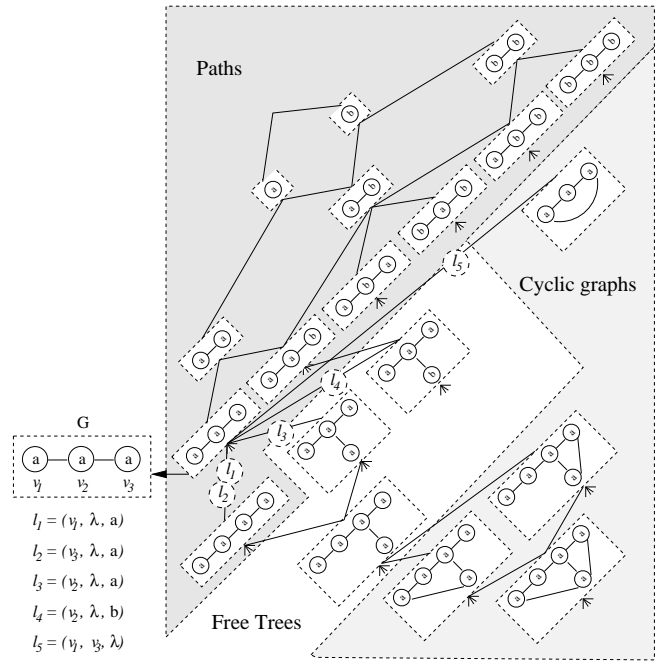


Figure 1: Refinement of graphs

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) $L' := \{l' | l' \text{ is a necessary leg of } C', support(\rho(C', l'), D) \geq minsup, l' \in L, \text{ or } l' \text{ connects to the node introduced by } l, \text{ if } l \text{ is a node refinement}\}$
- (5) **for all** $l' \in L'$ **do** DFGraphMiner (C', l', L')

Figure 2: Depth-first graph mining algorithm

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 2. 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 -predecessor 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 DFS codes (in gSpan [14]), adjacency matrices (in FFSM [5]) and tree codes with backtrack symbols (in FreeTreeMiner [2] and TreeMiner [15]). In order to make sure that no two isomorphic graphs are outputted 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 -predecessor 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.

However, all existing graph mining algorithms fail to meet these criteria. In gSpan, as part of the “pre-pruning” step in line (4), legs which are not on the rightmost path, as defined by the DFS code, are removed; furthermore legs are discarded by considering the labels of the neighbors to which the leg connects. Other legs are later rejected using “post-pruning” in line (2). In FFSM suboptimal canonical adjacency matrices are introduced, which also allow for legs that cannot be immediately added to a canonical graph. In both graph miners high computational costs are introduced as the check of line (2) may require an exhaustive search through all codes. In FreeTreeMiner [2] a linear characterization of necessary legs is given; however, in order to guarantee completeness of the search, the definition of a canonical code for free trees is relaxed to allow two canonical codes for *bi-centred free trees*, thus introducing additional costs as bi-centred free trees are considered twice.

By dividing the search into three phases, we show that one can avoid the check of line (2) in many cases. More precisely, we define a code for free trees with as property that one can determine necessary legs in constant time, and all these legs are canonical refinements immediately. Furthermore, we show that this approach can be combined with approaches for refining paths and cyclic graphs.

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 $bxaxa$. For each of these orientations, one can obtain a path code which consists of a sequence of node and edge labels. Each path has two potential predecessors which can be obtained by removing one of the endpoints; each of these predecessors has two orientations. We say that the lexico-

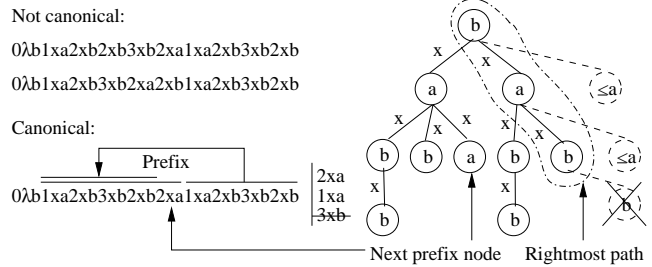


Figure 3: An unordered rooted tree, with some of its depth sequences and legs

graphically lowest among these 4 codes is *the* predecessor of the code. The code of a path is defined by the lowest of its two orientations. Given a path, we will see that each leg 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 which connects to the node v_i with the lowest index i is considered to be canonical.

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 [11], and has strong similarities with a method proposed in [9] for unlabeled free trees. Due to space limitations, we will summarize the results of these papers using an example. Every rooted tree can be encoded with a *depth sequence*, of which examples are given in Figure 3. 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 outputted, then the label of the edge going into that edge and finally the label of that node; we call this combination a depth sequence tuple. Clearly, the depth sequence depends on the order with 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 legs that connect to the rightmost path of the tree, as illustrated in the figure. 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

Research Track Poster

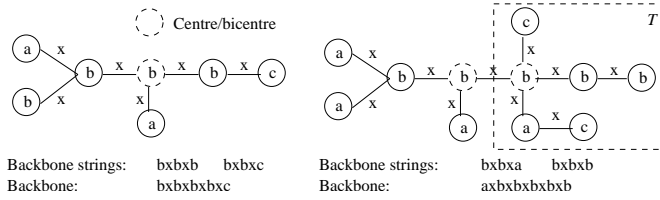


Figure 4: Free trees, (bi)centres and backbones

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 4). 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.

In order for this procedure to work, several details have to be specified. Due to space limitations we are obliged to omit most of them here. Cases which require special attention are free trees that grow from symmetric paths, and procedures that allow for free trees in which the centre has more than two neighbors. We will only give details for growing bicentred free trees with asymmetric backbones here. Consider one of the two initial rooted trees, then additional constraints apply to depth tuples that may be appended after the depth sequence: (1) no node may be added at a larger depth than the deepest initial node (as otherwise the resulting free tree would have a longer backbone than given by the initial path); (2) if a node is added at the highest allowed depth, the string of labels on the path from the root to the new node must be lower than that of the initial backbone path. Furthermore, a different lexicographical order on the depth tuples is necessary. To see this consider the rooted tree defined by $0\lambda b1xc1xb2xb1xa2xc$; without modified lexicographical order, this would be the canonical depth

sequence for the rooted tree T in Figure 4. Depth sequences of this part of the free tree should however grow from the backbone string which corresponds to the depth sequence $0\lambda b1xb2xb$. To allow for appending depth tuple $1xc$, we modify the order of the labels in each depth as follows. Let $D = \{0\lambda b, 1xb, 2xb\}$ be the set of depth tuples obtained from the backbone string, then the order between tuples with different depths and the order between tuples which are both not in D remain unaffected. However, if a tuple in D is compared with a tuple not in D , we define that the tuple in D always sorts higher. In this new order $0\lambda b1xb2xb1xc$ sorts higher than $0\lambda b1xc1xb2xb$ and is therefore canonical. One can show that in this new order, canonical strings always begin with the backbone tuples; all trees with a certain backbone can grow from the initial string of backbone tuples.

Using this procedure, in constant time all necessary node refining legs can be characterized precisely. The situation is worse for cycle closing legs. In contrast with the DFS code, the opportunities for pre-pruning such legs using labels are more limited. In most cases all cycle closing legs are necessary for the next phase.

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.

An alternative, of which we skip the details here, would be to store all cyclic graphs already found and to use a graph isomorphism algorithm like Nauty [8] to check that a graph is not generated twice.

Theoretical Evaluation. In comparison with the DFS code approach of gSpan, our method has advantages and disadvantages. The possibilities for pre-pruning backward edges using labels are more limited in our approach than in gSpan. On the other hand, for one class of graphs—the free trees—our approach allows for much more pre-pruning, and we are able to entirely remove the costly check of line (2). An interesting question is whether it is harder to compute if a cyclic graph is canonical in our code than in gSpan. To compute our code several steps are needed: first, we have to enumerate all spanning trees. This number is bounded by $O(|E_c|^c)$, were E_c is the number of edges of the graph that occurs in a cycle and c is the number of edges that should be removed to obtain a tree. Each spanning tree can then

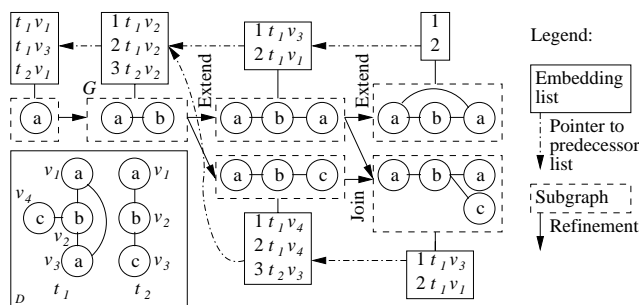


Figure 5: Maintenance of embedding lists

be normalized in $O(|E| \log |E|)$ time; the remaining closing legs can be normalized in $O(mc \log c)$ time, where m is the number of automorphisms of the spanning tree. In total this is $O(|E_c|^c (|E| \log |E| + mc \log c))$. If c and m are small, this computation is polynomial in the size of the graph to be normalized. Just like with a DFS code, this worst case is rarely reached as one can stop as soon as a better code is found than the current one. Overall, we may conclude that our approach is promising in cases where the number of labels is low and the number of cycles is not too large.

4. GRAPH COUNTING

We have considered several alternatives.

Embedding lists (EL). This approach is similar to that of MolFA [4], FFSM [5] and FreeTreeMiner [2], and is based on the idea of temporarily storing all occurrences of a graph. 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, embedding lists for legs of predecessor structures must also be stored.

Recomputed embeddings (RE). Embedding lists are quick, but scale up badly to large databases. Our other approach is similar to that of gSpan, and is based on maintaining a set of “active” graphs in which occurrences are repeatedly recomputed. As subgraph isomorphism is NP

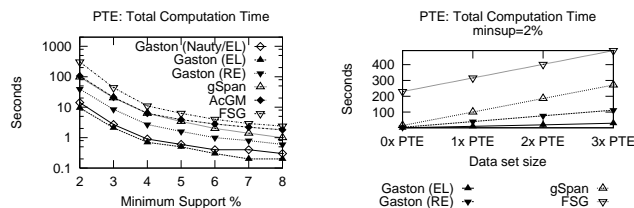


Figure 6: Results for the PTE dataset

Name	Contents
S5	10,000 artificial trees [15]
L10	100,000 artificial trees [15]
DTP	421 molecules [14]
PTE	340 molecules [14]
CAN2DA99	32,557 molecules [10]
AID2DA99	42,689 molecules [10]
NCI	250,251 molecules [10]

Figure 7: Summary of datasets

complete, 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 such as used in gSpan. 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. As a further optimization, we can apply the *quickstart* principle to find occurrences of graphs in free trees polynomially.

5. EXPERIMENTS

An overview of the results of our experiments can be found in Figures 6 and 8, 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 algorithm with a broad range of other algorithms: the graph miners gSpan, FSG and AcGM [6], and the FTM free tree miners of Rückert [13] and Chi [2]. All algorithms were obtained from their original authors. For our 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 [15]. Data set S5 is obtained by sampling 10k trees of maximal depth 5 from a master tree of 10k nodes with 3 node labels and fan-out 20. Set 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. Our main experiments regard molecular databases. We transform these datasets into graphs using the procedure given in [14]. Figure 6 gives a detailed insight into the performance of several algorithms on a commonly used benchmark [3]. By using linear regression in the scale-up experiment we can estimate how much time each algorithm spends in data size independent procedures, such as subgraph normalization. Detailed overhead runtimes are: 1s for GASTON (EL), 3s for GASTON (RE, thus strategies are computed), 14s for gSpan and 230s for FSG.

The other datasets were obtained from the National Can-

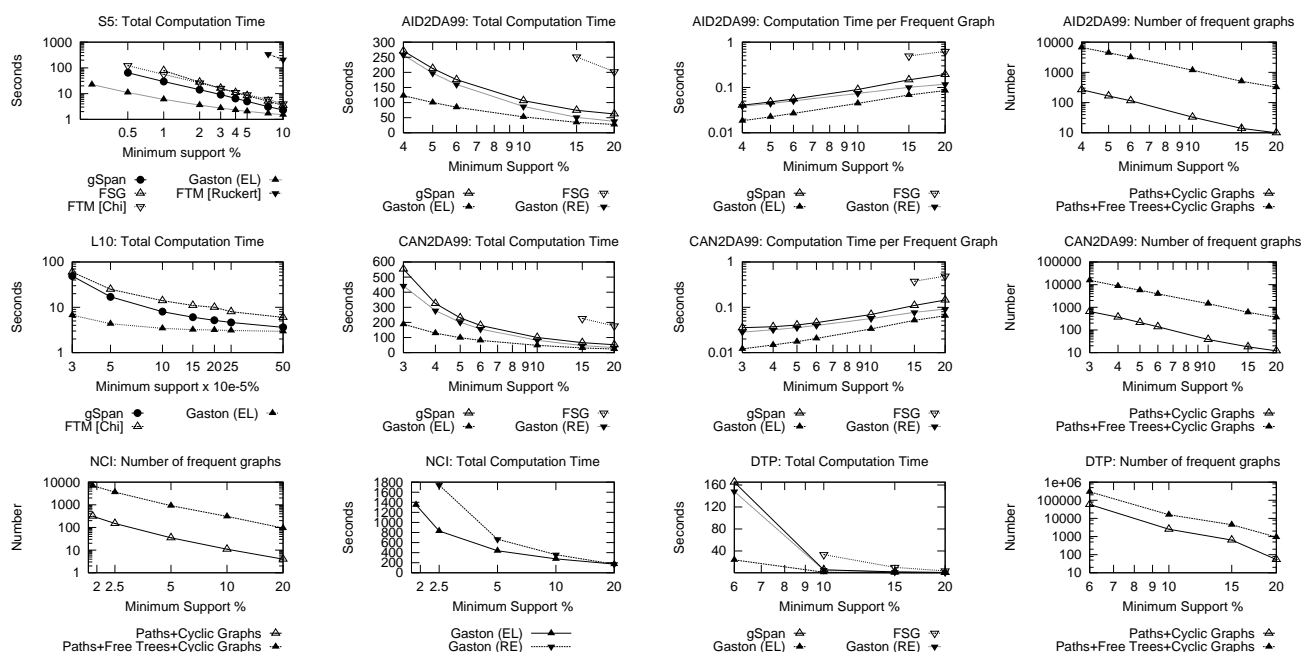


Figure 8: Results of our experiments

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

Figure 9: Differences between AID2DA99-active and compounds in NCI'99

cer Institute [10]. To test the scale-up properties of our algorithm, we have run our algorithm 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 (LE) required 1.7GB memory, GASTON (RE) 150MB. To exploit the activity information that is available for compounds in the CAN2DA99 and AID2DA99 datasets, in [4, 12, 13] 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. Thus better features for classifiers can be obtained. We modified our algorithm to allow for similar experiments. 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.

Acknowledgements We are especially grateful to Yun Chi for providing his source code and for discussing the topic. We wish to thank Xifeng Yan, Ulrich Rückert, Michihiro Kuramochi and Mohammed Zaki for providing algorithms and data sets.

6. 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] 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.
- [5] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Proceedings of the ICDM*, 2003.
- [6] 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.
- [7] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the ICDM*, pages 313–320, 2001.
- [8] B. D. McKay. Practical graph isomorphism. 30:45–87, 1981.
- [9] S. Nakano and T. Uno. A simple constant time enumeration algorithm for free trees. In *IPSP SIGNotes Algorithms*, number 091–002, 2003.
- [10] National Cancer Institute (NCI). DTP/2D and 3D structural information, <http://cactus.nci.nih.gov/ncidb2/download.html>. 1999.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] X. Yan and J. Han. CloseGraph: Mining closed frequent graph patterns. In *Proceedings of the SIGKDD*, pages 286–295, 2003.
- [15] M. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the SIGKDD*, pages 71–80, 2002.