Journal of Machine Learning Research 10 (2009) 907-929 Submitted 2/08; Revised 2/09; Published 4/09

Polynomial-Delay Enumeration of Monotonic Graph Classes

Jan Ramon JAN.RAMON@CS.KULEUVEN.BE
Siegfried Nijssen SIEGFRIED.NIJSSEN@CS.KULEUVEN.BE
K.U.Leuven, Dept. of Computer Science
Celestijnenlaan 200A, B-3001 Leuven

Editor: Stefan Wrobel

Abstract

Algorithms that list graphs such that no two listed graphs are isomorphic, are important building blocks of systems for mining and learning in graphs. Algorithms are already known that solve this problem efficiently for many classes of graphs of restricted topology, such as trees. In this article we introduce the concept of a dense augmentation schema, and introduce an algorithm that can be used to enumerate any class of graphs with polynomial delay, as long as the class of graphs can be described using a monotonic predicate operating on a dense augmentation schema. In practice this means that this is the first enumeration algorithm that can be applied theoretically efficiently in any frequent subgraph mining algorithm, and that this algorithm generalizes to situations beyond the standard frequent subgraph mining setting.

Keywords: graph mining, enumeration, monotonic graph classes

1. Introduction

Among the most prominent graph mining problems is the problem of finding frequent subgraphs in databases of small graphs of any topology. This is witnessed by the large number of algorithms that have been proposed for this task (Yan and Han, 2002; Borgelt and Berthold, 2002; Kuramochi and Karypis, 2004; Inokuchi et al., 2003; Inokuchi, 2004; Huan et al., 2003; Nijssen and Kok, 2004; Leskovec et al., 2006). A fundamental problem that is addressed in all these works is how to enumerate a set of graphs such that no two graphs in the enumerated set are isomorphic with each other. The main motivation for this focus is that if duplicates would not be avoided, these algorithms would access the data more often than necessary and produce results that are larger than required.

To avoid isomorphic graphs in their output, all these existing graph mining algorithms use a methodology based on canonical codes. A canonical code is a code that uniquely identifies a set of isomorphic graphs. To determine if a graph should be part of the output, its canonical code is computed, and, in some algorithms, compared with the canonical codes of graphs found before.

A fundamental problem with the canonical code based approach, however, is that we essentially need to solve the graph isomorphism problem: if we could compute the canonical code of any graph efficiently, we could compute the codes of two graphs to determine if they are isomorphic. The state of the art is that no polynomial algorithm for the graph isomorphism problem is known. Consequently, it can be shown that when the existing graph mining algorithms are enumerating candidate subgraphs, the *delay* between two enumerated graphs is exponential in the worst case (in terms of the size of the largest graph enumerated).

The contribution of this article is that we introduce a novel algorithm for enumerating graphs that does not use canonical codes, and incrementally maintains data structures that ensure that no two isomorphic graphs are listed. We show that in contrast to other algorithms for enumerating graphs, this new algorithm outputs many classes of graphs, including arbitrary connected graphs, with polynomial delay, which makes this algorithm theoretically more efficient than any other graph enumeration algorithm used in the graph mining literature.

It is important to note that our algorithm works for many classes of graphs. If we would restrict the topology of the graphs, for instance, to only those graphs that are trees, the enumeration problem of frequent graph mining is already known to be more efficiently solvable, and algorithms are known (Wright et al., 1986; Nakano and Uno, 2004) and used in practice (Chi et al., 2005; Horváth et al., 2006).

Even though the frequency constraint is the most popular constraint in the graph mining literature, other constraints have been studied as well. An important property of the frequency constraint is that it is monotonic w.r.t. to subgraph isomorphism: if a graph is frequent, all its subgraphs are also frequent. In our algorithm we exploit this property to maintain data structures incrementally. An interesting question is to what extent enumeration with polynomial delay is feasible when the graphs to enumerate are not monotonic under the subgraph isomorphism relation. To this aim, we developed the concept of an *augmentation schema*. The augmentation schema defines relations between graphs in the space of graphs to enumerate (in the simplest case, the subgraph isomorphism relation). We will show that enumeration with polynomial delay is possible as long as an augmentation schema satisfies certain conditions, and the graphs to enumerate can be specified using a monotonic predicate w.r.t. the augmentation schema. We will specify our algorithm in terms of such augmentation schemas. This makes our method general enough to be applied in settings beyond the traditional frequent subgraph mining setting, and allows us also to enumerate both connected and unconnected graphs. For instance, we can also enumerate hereditary classes of graphs with bounded degree; a class of graphs is called *hereditary* if it is monotonic under the *induced* subgraph relation, instead of the traditional subgraph relation.

The problem of graph enumeration has not only been studied in the graph mining literature. In particular, Goldberg showed in the early nineties that there is a polynomial delay algorithm to list all graphs (Goldberg, 1992). We will provide more details about this algorithm in Section 3, where we will show that this algorithm cannot be used to list graphs that satisfy a monotonic predicate, as required in a graph mining setting. Many algorithms exist for enumerating classes of graphs without taking into account isomorphisms, such as classes of graphs described by first order logic formulas (Goldberg, 1993) and edge-maximal graphs with bounded branchwidth (Paul et al., 2006); it is not known how to list these classes while taking into account isomorphisms. Heuristic implementations exist for enumerating graphs in general (McKay, 1998), but these do not guarantee polynomial delay.

Our algorithm uses similar ideas as the algorithm of Goldberg (1992). In particular, as our algorithm maintains a data structure incrementally, our algorithm requires that all computed subgraphs are stored. In the pattern mining setting, where we are interested in finding these graphs, this is a common assumption.

This article is the full version of a workshop abstract (Ramon and Nijssen, 2007). Compared to the workshop abstract, in this article (1) we show how our method extends to other classes of graphs than connected graphs and (2) we provide full details and proofs.

The article is organized as follows. In Section 2 we introduce the problem of subgraph mining. In Section 3 we show why the algorithm of Goldberg is too limited for applications in graph mining. In Section 4 we introduce the concept of augmentation schemas and formally define the enumeration problems that we are addressing. In Section 5 we state our results. In Section 6 we provide a short introduction to concepts in group theory which we need in Section 7, where we outline our algorithm; Section 8 concludes. The proofs of our claims are given in an appendix.

2. Motivation

The main motivation for our work is the problem of efficiently mining subgraphs under constraints. The most common such problem is the problem of mining frequent subgraphs in a database of small graphs. We will first give a formal definition of this problem.

A graph g is a tuple (V,E) where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. We denote with $V(g)$ the set of vertices and with $E(g)$ the set of edges of a graph g. In this article we restrict ourselves to unlabeled, simple graphs (i.e., undirected, unweighted, no loops, no multiple edges between two nodes). It easy to lift these restrictions. In particular, in frequent subgraph mining it is usually assumed that graphs have labels. However, our discussion is simplified by assuming that we do not have labels; this is not a fundamental restriction of our methodology.

There are many ways in which one can restrict the topology of graphs. For instance, a *path* is a graph in which all nodes have degree 2, except two nodes, which have degree one. A *tree* is a connected graph with k nodes and $k-1$ edges. When we use the word *graph*, we refer to graphs that have no *apriori* restriction on their topology (except being unlabeled and simple).

Between two graphs we can define the graph isomorphism and the subgraph isomorphism relations. Our definitions are as usual in the literature: two graphs g_1 and g_2 are *isomorphic* iff there is a bijection $\varphi : V(g_1) \to V(g_2)$ such that $(v_1,v_2) \in E(g_1) \Leftrightarrow (\varphi(v_1),\varphi(v_2)) \in E(g_2)$. We denote this by $g_1 \simeq_\varphi g_2$, where φ is the bijection between the graphs. The bijection can be omitted if this is clear from the context.

A graph g_1 is *subgraph isomorphic* to g_2 iff there is a *subgraph* (V',E') with $V' \subseteq V(g_2)$ and $E' \subseteq E(g_2)$, such that g_1 is isomorphic with (V',E'). This is denoted with $g_1 \preceq_\varphi g_2$, where φ is the bijection between the nodes of g_1 and the subset of nodes of g_2. A subgraph (V',E') of g_2 is an *induced subgraph* if for all $v,v' \in V' : \{v,v'\} \in E(g_2) \leftrightarrow \{v,v'\} \in E'$; in other words, all edges in g_2 between nodes in V' are also present in E'. A graph g_1 is *induced subgraph isomorphic* to g_2 iff g_1 is isomorphic with an induced subgraph of g_2.

The graph isomorphism and *sub*graph isomorphism problems should not be confused with each other. The *sub*graph isomorphism problem is known to be NP complete, while the graph isomorphism problem is believed to be in a complexity class of its own. For both problems in general no polynomial algorithm is known (Köbler et al., 1993).

The problem of frequent subgraph mining can now be formalized as follows. Given is a database of graphs, $DB = \{g_1,g_2,\ldots,g_n\}$, and a threshold t. Then we are interested in finding all graphs g for which the support is higher than or equal to t. The *support* of a graph g is the number of graphs in DB with which g is subgraph isomorphic.

This frequent subgraph mining problem can be generalized by replacing the minimum frequency constraint with other predicates. For instance, a predicate could involve an additional maximum size constraint.

A predicate on graphs is called *monotonic* if all subgraphs of a graph that satisfies the predicate, will also all satisfy the predicate.[1] The support constraint and the maximum size constraint are examples of predicates that are monotonic under subgraph isomorphism.

The problem of constraint-based subgraph mining is closely related to the problem of frequent item set mining. Many algorithms have been developed to tackle the frequent item set mining problem, the most well-known being the APRIORI algorithm (Agrawal et al., 1996). Both frequent graph mining algorithms and frequent item set mining algorithms are considered to be constraint-based *pattern* mining algorithms. Constraint-based pattern mining algorithms look for patterns in a pattern language $\mathcal{L}$, and assume that these patterns are ordered using a partial order relation $\leq$ on $\mathcal{L}$. In the case of graph mining, $\leq$ is usually the subgraph isomorphism relation.

Many algorithms for pattern mining are level-wise (breadth-first) enumeration algorithms. These algorithms assume that the *size* of a pattern in the language is well-defined, and look for the patterns by listing them increasing in size. A high-level description of such an algorithm is given in Algorithm 1.

Algorithm 1 Level-Wise Pattern Miner

Require: A pattern language $\mathcal{L}$ and a monotonic predicate p
Ensure: output all $g \in \mathcal{L}$ with $p(g)$

1: $\mathcal{C}_1 \leftarrow$ patterns of size 1
2: $k \leftarrow 1$
3: **while** $\mathcal{C}_k \neq \emptyset$ **do**
4: $\quad \mathcal{F}_k \leftarrow \{g \in \mathcal{C}_k | p(g)\}$
5: $\quad$ Generate $\mathcal{C}_{k+1}$ from $\mathcal{F}_k$
6: $\quad k \leftarrow k+1$
7: **end while**
8: Output $\bigcup_k \mathcal{F}_k$

In this algorithm, $\mathcal{F}_k$ contains the patterns of size k that satisfy the predicate. In line 4 it is determined which candidates of size k satisfy the predicate. In frequent pattern mining, this line requires access to the data, and can be most time consuming. It is therefore essential that $\mathcal{C}_k$ be as small as possible.

The main focus of this article is on the computation that needs to be performed in line 5. In this line new candidates should be generated. This generation should ensure the following:

- by repeatedly generating new candidates we should be able to enumerate all patterns in the pattern space, in our case the space of all unlabeled, simple graphs;
- to ensure that the algorithm is as efficient as possible, we should not insert two patterns in $\mathcal{C}_{k+1}$ that are equivalent with each other; in our case, we should avoid inserting two graphs that are isomorphic;
- we should not insert patterns in $\mathcal{C}_{k+1}$ for which we can know beforehand that p will not be true; in our case, we should exploit the monotonicity of p to avoid inserting graphs of which a subgraph is not included in $\mathcal{F}_k$.

1. We adopt here the terminology most common in graph theory. Some authors in the data mining literature use the term 'anti-monotonic'.

In the graph mining setting, the second and third requirements are difficult, as the second requirement requires us to solve a graph isomorphism problem, and the third requirement involves a subgraph isomorphism problem.

Algorithm 1 has applications beyond traditional frequent subgraph mining. For instance, if we are interested in computing a decomposition graph kernel between two graphs which counts the number of *non-isomorphic* subgraphs that two graphs have in common, we could compute this kernel by providing algorithm 1 a database of two graphs as input and a threshold of $t = 2$. The size of the output is the desired kernel value.

Similarly, we could be interested in enumerating all *different* graphs that include one node in a network (Leskovec et al., 2006). In this case, the input of Algorithm 1 consists of one graph with all nodes up to a certain threshold distance from the node of interest, and the subgraph isomorphism should be restricted such that only bijections are considered in which at least one node in the pattern is mapped to the special node in the data.

In all cases, the essential problem of enumerating graphs without duplicates remains. Several algorithms have been proposed in the literature to address this graph enumeration problem. The main idea that has been employed, is that for every graph, we can compute a *canonical* code, that is, a code that is unique for all graphs that are isomorphic. The level-wise graph miners AGM (Inokuchi et al., 2003) and FSG (Kuramochi and Karypis, 2004) define a canonical code from adjacency matrices. Essentially, all subgraphs are stored in a data structure that is indexed according to this canonical code, and duplicates are avoided by computing for every candidate the canonical code. The approaches of AGM and FSG differ in their definition of *size*: AGM grows graphs by adding nodes, FSG by adding edges.

Other graph miners search depth-first, but their enumeration strategy can easily be modified for use in a level-wise algorithm (Yan and Han, 2002; Huan et al., 2003; Nijssen and Kok, 2004). Also these algorithms use a canonical code, but do not require the use of an indexed data structure. An algorithm for enumerating graphs surrounding a node in a network was proposed by Leskovec et al. (2006). Again, this algorithm used a canonical code.

Unfortunately, currently no polynomial algorithm is known to compute a canonical code; if one was known, we would be able to solve the graph isomorphism problem in polynomial time. Overall, this means that in all existing graph mining algorithms exponential time can be spent between two graphs that are inserted in the set of candidates.

Enumeration algorithms for which this is not the case, that is, algorithms that solve an enumeration problem such that between any two enumerated solutions polynomial time is spent (in terms of the largest enumerated solution), are known as algorithms with *polynomial delay*. To the best of our knowledge, all algorithms that have been proposed in the graph mining literature for enumerating graphs in general do not have polynomial delay. Only for restricted classes of graphs, such as trees and outerplanar graphs, algorithms with polynomial delay are known (Chi et al., 2005; Horváth et al., 2006).

However, the fact that graph isomorphism is not known to be polynomially computable, does not imply that graph enumeration cannot be solved with polynomial delay. Even though ignored in the data mining and machine learning literature, a polynomial algorithm for enumerating graphs does exist and was proposed by Goldberg (1992).

The problem with this algorithm is that it solves a rather simple enumeration problem: given a bound on the size of the graphs to enumerate, Goldberg's algorithm lists all graphs of this size with polynomial delay. In the case of data mining and machine learning, we are dealing with more

complicated monotonic constraints that are data-dependent. We will show in the next section that we can create databases such that the set of graphs to enumerate does not fulfill the basic assumptions that need to be satisfied in Goldberg's algorithm. Even worse, we will see that this type of data is very common.

It should be stressed that this paper only studies the candidate generation of graph mining algorithms; it does not study the frequency evaluation. For general graphs the frequency evaluation also takes exponential time; a general frequent graph miner which uses our enumeration algorithm will still have exponential delay due to the fact that frequency evaluation is still exponential. This article proposes an improvement only of the candidate generation phase. The key insight is that we devised a graph enumeration algorithm which does not use canonical codes to perform this task.

3. Goldberg's Algorithm

In this section we briefly discuss the key points in the algorithm of Goldberg (1992), which shows why this algorithm cannot be used in a pattern mining setting.

Goldberg's algorithm aims at listing all graphs with n nodes, and makes a distinction between easy and hard graphs. Easy is a graph g that satisfies at least one of these two properties:

- g has a vertex with degree $n-1$, that is, at least one vertex is connected to all other vertices;
- g has only one vertex, say v, of maximum degree and $g-v$ is rigid, that is, the graph $g-v$ has only one isomorphism with itself (called the identity automorphism in Section 6).

An example of a graph that is never rigid, is a path.

Let $E(n)$ be the set of easy graphs with n nodes, and $U(n)$ the set of all graphs with n nodes, then it was shown by Goldberg that

$$2|E(n)| \geq |U(n)|.$$

This property implies that a large fraction of the graphs to enumerate are in fact easy. It was then shown that $E(n)$ can be listed with polynomial delay, and that $H(n) = U(n) \setminus E(n)$ can be listed in $O(n^4|U(n)|)$ time steps, where $|U(n)|$ is exponential in n but linear in the number of solutions. The main idea is then to interleave these two methods. The method which lists easy graphs, makes sure that the delay is polynomial. The other method is allowed to spend an exponential number of steps between consecutive graphs, but these steps are spread over several iterations of the method that lists easy graphs. Effectively this gives an algorithm with polynomial delay.

It is clear that this method fundamentally relies on the property that many graphs are 'easy'. This property does not hold for sets of graphs defined by a monotonic predicate. Let us illustrate this for the monotonic constraint that every node in a graph has a degree of at most three. If $n > 3$, it is easily seen that

- as the degree is at most 3, the number of graphs that contain a vertex that is connected to all other vertices is independent of n;
- every graph g that contains a single node v of maximum degree 3, consists, after removal of v, only of a set of (possibly unconnected) paths, hence $g-v$ is not rigid.

Consequently, $E(n)$ is a constant independent of n, while $U(n)$ grows with n. The condition of Goldberg's approach is therefore not satisfied for this class of graphs.

Moreover, to list all graphs in $H(n)$ in time $O(n^4|U(n)|)$, it is assumed that the average size of the automorphism groups of the elements of $U(n)$ is bounded. However, one can find subclasses for which this bound is not polynomial.

The most popular application of graph mining algorithms is in chemistry (Horváth et al., 2006). Most of the graphs in these databases have a degree bounded by four, and a majority of the subgraphs that need to be enumerated have a degree bounded by three. Thus, we do not believe that the conditions for Goldberg's method are satisfied in such data.

4. Problem Statement

Our problem setting has two parameters:

- an augmentation operator, which takes as input a graph, and outputs a set of augmentations of this graph, and whose closure, starting from a given set of graphs, describes a class of graphs;
- a predicate which restricts this class.

For instance, the augmentation operator can be used to describe the class of connected or unconnected graphs, while the boolean predicate can restrict this class further to those graphs that have bounded degree.

More formally, we will denote by $\mathcal{VE}$ the set that contains all pairs (r_V, r_E) where r_V is a set of vertices and r_E a set of edges (not necessarily between vertices in r_V). We will use set operators on elements of $\mathcal{VE}$ to denote the corresponding operations on their components, for example, $(r_V, r_E) \cup (r'_V, r'_E) = (r_V \cup r'_V, r_E \cup r'_E)$. Again, $V(r)$ and $E(r)$ describe the components of an element $r \in \mathcal{VE}$. An augmentation operator ρ^+ is a function that takes as input a graph, and outputs a set of descriptions of possible augmentations. This set is a subset of $\mathcal{VE}$. Every element $r \in \rho^+(g)$ describes a new graph $(V(g) \cup V(r), E(g) \cup E(r))$, abbreviated by $g + r$, that we call a *child* of g.

An example of an augmentation operator is

$$\rho_t^+(g) = \{(\{v_{new}\}, \{\{v, v_{new}\}\}) \,|\, v \in V(g)\};$$

where v_{new} is a new vertex (not belonging to $V(g)$). This operator adds a new vertex and connects it to an existing vertex. We can use this operator to describe the set of all (connected) trees. The minimal graph on which we apply the operator is in this case the graph with one node $\top_t = (\{v\}, \emptyset)$; in general, when we use one graph as the minimal element, we will denote this initial graph with $\top$.

The following operator enumerates all graphs:

$$\rho_a^+(g) = \{(\{v_{new}\}, \emptyset)\} \cup \{(\emptyset, \{\{v_1, v_2\}\}) | v_1, v_2 \in V(g) \wedge \{v_1, v_2\} \notin E(g)\}. \quad (1)$$

with $\top_a = (\emptyset, \emptyset)$, while the following allows for enumerating all connected graphs:

$$\rho_c^+(g) = \rho_t^+(g) \cup \{(\emptyset, \{\{v_1, v_2\}\}) | v_1, v_2 \in V(g) \wedge \{v_1, v_2\} \notin E(g)\}. \quad (2)$$

with $\top_c = (\{v\}, \emptyset)$.

As we can see in these examples, the vertices occurring in edges $E(r)$ do not have to occur in $V(r)$. Still it is useful to determine the entire set of vertices involved in an augmentation. For this we use the notation V^*, that is, $V^*(r) = V(r) \cup \{v \mid \exists e \in E(r) : v \in e\}$. Observe that the example

operators output a number of augmentations that is bounded by a polynomial in the size of g, and that for each r, the size of the set $V^*(r)$ is bounded by a constant.

The class of graphs defined by taking the closure of the augmentation operator on the minimal element is denoted by $\mathcal{L}_{\rho^+}$ (which we shorten further to $\mathcal{L}_a$ and $\mathcal{L}_c$ for the classes defined by ρ_a^+ and ρ_c^+). The operator ρ^+ defines an ancestry relation between the graphs. This relation is a partial order.

The second parameter of our problem setting is a predicate p on graphs. The set of graphs g in $\mathcal{L}_{\rho^+}$ such that $p(g)$ is true is denoted by $\mathcal{L}_{\rho^+,p}$. We only consider predicates that cannot distinguish between isomorphic graphs, that is, if $g \simeq g'$ then $p(g) = p(g')$. We call a predicate *monotonic* w.r.t. an augmentation operator ρ^+ if for every graph $g \in \mathcal{L}_{\rho^+,p}$ it also holds that $g' \in \mathcal{L}_{\rho^+,p}$ for every g' that is an ancestor of g. For instance, the predicate that tests if a graph has bounded degree, is monotonic under ρ_a^+ as defined in (1).

In this article, we consider the following problem.

Problem 1 *Given are an augmentation operator ρ^+ and a predicate p which is monotonic w.r.t. ρ^+. Then, enumerate all elements in $\mathcal{L}_{\rho^+,p}$ such that exactly one representative of every equivalence class under isomorphism of $\mathcal{L}_{\rho^+,p}$ is enumerated.*

In the next section we determine a set of sufficient conditions on the augmentation operator and the monotonic predicate that have to be fulfilled in order to obtain an algorithm with polynomial delay.

5. Main Result

The augmentation operator that we introduced in the previous section, generates the children of a graph. Our algorithm relies on the existence of an operator which can inverse this operator. We call this operator a *reduction operator*. The reduction operator generates the *parents* of a graph.

Formally, the definition of a reduction operator is similar to that of an augmentation operator; the input of a reduction operator ρ^- is a single graph, its output consists of a subset of $\mathcal{VE}$. We call each element $r \in \rho^-(g)$ a reduction of g. It defines a graph $(V(g) \setminus V(r), E(g) \setminus E(r))$, which is abbreviated by $g - r$.

For instance, in the case of connected graphs, we can define the following reduction operator:

$$\begin{aligned} \rho_c^-(g) \quad = \quad & \{r \,|\, r = (\emptyset, \{\{v_1, v_2\}\}) \wedge \{v_1, v_2\} \in E(g) \wedge \{v_1, v_2\} \text{ is in a cycle}\} \\ & \{r \,|\, r = (\{v_1\}, \{\{v_1, v_2\}\}) \wedge \{v_1, v_2\} \in E(g) \wedge v_1 \text{ has degree 1}\} \end{aligned} \tag{3}$$

Let $\mathcal{L}$ be a class of graphs. Then, an augmentation schema on $\mathcal{L}$ is a pair (ρ^+, ρ^-) of an augmentation operator ρ^+ and a reduction operator ρ^-, such that

- $\forall g \in \mathcal{L}, \forall r \in \rho^+(g) : g + r \in \mathcal{L} \wedge g \cap r = (\emptyset, \emptyset)$, that is, $\rho^+(g)$ contains augmentations that can be added to g to obtain a larger graph (child);
- $\forall g \in \mathcal{L}, \forall r \in \rho^-(g) : g - r \in \mathcal{L} \wedge r \subseteq g$, that is, $\rho^-(g)$ contains reductions that can be removed from g to obtain a parent;
- $\forall g \in \mathcal{L}, \forall r \in \rho^+(g) : r \in \rho^-(g + r)$, that is, the effects of the additions $r \in \rho^+(g)$ can be inverted by a deletion from $\rho^-(g + r)$;

- $\forall g \in \mathcal{L}, \forall r \in \rho^-(g) : \exists r' \in \rho^+(g-r), \exists \varphi : \big((g-r)+r' \simeq_\varphi g\big) \wedge \big(I_{g-r} \subseteq \varphi\big)$, that is, deletions $r \in \rho^-(g)$ can be inverted by additions from $\rho^+(g-r)$. Here $I_{g-r} = \{(v,v) \mid v \in V(g-r)\}$ is the identity permutation over the vertices of $g-r$;
- $\forall g_1, g_2 \in \mathcal{L} : g_1 \simeq_\varphi g_2 \Rightarrow \forall r \in \rho^+(g_1) : \varphi(r) \in \rho^+(g_2)$, that is, ρ^+ (and hence also $\rho^-(g)$) is invariant to isomorphisms.

Given a graph g and two reductions $r_1, r_2 \in \rho^-(g)$, we are interested in applying both r_1 and r_2 to g. However, sometimes this is not possible directly. Consider for instance the class of connected graphs $\mathcal{L}_c$, the graph $g = (\{1,2,3\}, \{\{1,2\},\{2,3\},\{3,1\}\})$, and the reductions $r_1 = (\emptyset, \{\{1,2\}\})$ and $r_2 = (\emptyset, \{\{2,3\}\})$. We have $f_1 = g - r_1 = (\{1,2,3\}, \{\{2,3\},\{3,1\}\})$. We cannot apply r_2 to f_1 as this would result in a graph which is not in $\mathcal{L}_c$ due to the isolated node 2; r_2 is not an allowed reduction in $g - r_1$. We can however map the reduction r_2 to a reduction that is allowed; instead of r_2 we use $(\{2\}, \{\{2,3\}\})$, which is a valid deletion from f_1. This translation of r_2 to the context of $g - r_1$ for ρ_c^- is denoted by $r_2 \uparrow_c^g r_1$. More formally, for connected graphs $\mathcal{L}_c$, we define $r_2 \uparrow_c^g r_1$ to be equal to r_2, except in the case where $r_1 = (\emptyset, \{v, u_1\})$, $r_2 = (\emptyset, \{v, u_2\})$ and v has degree 2, in which case $r_2 \uparrow_c^g r_1 = r_2 \cup (\{v\}, \emptyset)$. Then, $(g - r_1) - \big(r_2 \uparrow_c^g r_1\big) = (\{1,3\}, \{\{1,3\}\})$.

We now more formally introduce the properties of these reduction translators. A reduction translator is an operator $\cdot \uparrow^\cdot \cdot$ mapping graphs g and reductions $r_1, r_2 \in \rho^-(g)$ to a new reduction $r_2 \uparrow^g r_1$ satisfying

$$\forall g, r_1, r_2 : r_2 \subseteq r_2 \uparrow^g r_1, \tag{4}$$

that is, after the translator is applied the reduction can only become larger,

$$\forall g, r_1, r_2 : r_1 \cap (r_2 \uparrow^g r_1) = (\emptyset, \emptyset), \tag{5}$$

that is, no element is removed twice, and

$$\forall g, r_1, r_2 : (r_2 \uparrow^g r_1) \cup r_1 = (r_1 \uparrow^g r_2) \cup r_2, \tag{6}$$

that is, reversing the order does not affect which vertices and edges are removed.

To allow for an efficient enumeration of graphs by our algorithm, we require that the augmentation schema of the class of graphs fulfills a *density* property. This property is based on the *graph of parents*. The graph of parents $GoP_{\rho^-,\uparrow}(g)$ of a graph g is defined as follows:

- $V(GoP_{\rho^-,\uparrow}(g)) = \rho^-(g)$, that is, every reduction for g corresponds to a vertex in its graph of parents.
- For any two $r_1, r_2 \in V(GoP_{\rho^-,\uparrow}(g))$, the graph $GoP_{\rho^-,\uparrow}(g)$ has edge $\{r_1, r_2\}$ iff $(r_1 \uparrow^g r_2) \in \rho^-(g - r_2)$ and $(r_2 \uparrow^g r_1) \in \rho^-(g - r_1)$.

The intuition is that there is an edge between two nodes in the graph of parents, iff it is possible to go from one parent to the other by first applying a reduction, and then applying an augmentation. An example is shown in Figure 1.

We say that $(\rho^+, \rho^-, \uparrow)$ is a dense augmentation schema for $\mathcal{L}$ iff

1. (ρ^+, ρ^-) is an augmentation schema for $\mathcal{L}$ and $\uparrow$ is a reduction translator.
2. for every non-minimal graph $g \in \mathcal{L}$ it holds that either the graph of parents $GoP_{\rho^-,\uparrow}(g)$ is connected, or $g - r$ is a minimal element of $\mathcal{L}$ for all $r \in \rho^-(g)$ (in most cases, $g - r = \top$).

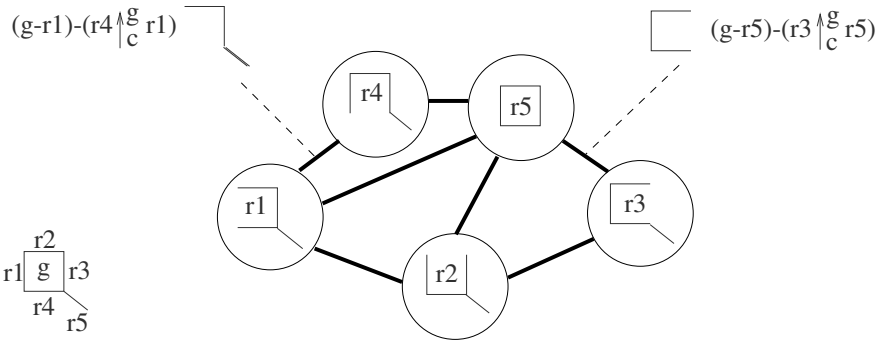

Figure 1: On the bottom-left, a connected graph g is drawn. In the middle, there is the graph of parents $GoP_{\rho_c^-,\uparrow_c}(g)$. For every of its vertices r_i, the corresponding $g - r_i$ of g is drawn. For the edges $\{r_1, r_4\}$ and $\{r_3, r_5\}$, the common (grand)parent is drawn. Note that there is no edge between for example r_3 and r_4 as $(g - r_3) - (r_4 \uparrow_c^g r_3)$ is not a connected graph.

Let us return to the examples of graph classes. For the class of all graphs $\mathcal{L}_a$, we define $r_1 \uparrow_a^g r_2 = r_1$. It is easy to show that $GoP_{\rho_a^-,\uparrow_a}(g)$ is a clique and that $(\rho_a^+, \rho_a^-, \uparrow_a)$ is a dense augmentation schema. We can also prove for connected graphs that $(\rho_c^+, \rho_c^-, \uparrow_c)$ is a dense augmentation schema (see the appendix for a proof).

The main result of this paper is the following:

Theorem 1 *Given an augmentation operator ρ^+ and a predicate p. We can solve problem 1 with polynomial delay if the following conditions are satisfied:*

- *there exists a reduction operator ρ^- and a reduction translator $\uparrow$ such that $(\rho^+, \rho^-, \uparrow)$ is a dense augmentation schema;*
- *the predicate p is monotonic w.r.t. ρ^+ over $\mathcal{L}_{\rho^+}$;*
- *ρ^+, ρ^- and p can be evaluated in polynomial time in their arguments;*
- *the number of vertices in $V^*(r)$ for every possible r resulting from ρ^+ and ρ^- is bounded by a constant.*

There are many classes of graphs for which these conditions are fulfilled. Below we give a non-exhaustive list of examples.

5.1 Monotonic Classes

Any predicate p that is monotonic w.r.t. edge and vertex deletion and that can be evaluated in polynomial time defines a class $\mathcal{L}_{\rho_a^+,p}$ that satisfies the conditions of Theorem 1, as $(\rho_a^+, \rho_a^-, \uparrow_a)$ is a dense augmentation schema, and all operations are polynomial. Hence, our algorithm can efficiently enumerate all these monotonic classes.

An interesting special case are the minor-closed classes of graphs. The minors of a graph can be obtained by deleting edges, vertices, or identifying adjacent vertices into a single new vertex that is adjacent to all vertices that were adjacent to any of the identified original vertices. A class is minor-closed if for any graph g all its minors are also included in the class. Some classes of graphs

can be characterized by forbidden minors, that is, minors that none of the graphs are allowed to have. One example is the class of all planar graphs (the forbidden minors being the 5-clique K_5 and the complete bipartite graph $K_{3,3}$). It can be determined in polynomial time if a graph contains a given forbidden minor, and minor-closed classes of graphs are monotonic for ρ_a^+.

5.2 Connected Graphs

It is proven in the appendix that the augmentation schema $(\rho_c^+, \rho_c^-, \uparrow_c)$ is dense and its operators are polynomial. It follows that connected graphs can be enumerated with polynomial delay. Any predicate which is closed under edge and vertex deletion is also monotonic w.r.t. ρ_c^+. Therefore, for any monotonic predicate p, our algorithm can list all connected graphs satisfying p with polynomial delay.

5.3 Hereditary Classes with Bounded Degree

A predicate p is hereditary iff for every graph g_1 such that $p(g_1)$ holds, $p(g_2)$ holds for any *induced* subgraph g_2 of g_1. As pointed out in Section 2, an *induced* subgraph must contain all edges between a selected set of nodes in the original graph. A hereditary predicate is monotonic w.r.t. the following augmentation operator:

$$\rho_h^+(g) = \left\{ \left(\{v_{new}\}, \{\{v_{new}, v'\} | v' \in V\} \right) \mid V \subseteq V(g) \right\},$$

where again v_{new} is a new vertex not in $V(g)$. The corresponding reduction operator should remove every single vertex as well as all edges emanating from it. It can be shown that the resulting augmentation schema is dense. However, the augmentation operator ρ_h^+ outputs a number of augmentations exponential in the size of the input. If we restrict ourselves to graphs with bounded degree, however, we can enumerate the resulting class with polynomial delay.

One example is the class of claw-free graphs. A graph is called claw-free if the graph

$$(\{v_1, v_2, v_3, v_4\}, \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}\})$$

is not one of its induced subgraphs. Claw-freeness is not a monotonic property in $\mathcal{L}_{\rho_a^+}$, as a subgraph of a graph without claws can contain a claw: for instance, consider the clique K_5; this graph does not have a claw as induced subgraph (all its induced subgraphs also being cliques), but many of its (ordinary) subgraphs are claws.

In general, claw-freeness is a hereditary property, as every induced subgraph of a graph without claws, is claw-free. To the best of our knowledge no polynomial algorithm is currently known for enumerating all claw-free graphs even if we do allow for duplicates (Goldberg, 1992). Our algorithm partially solves this problem by allowing for listing all claw-free graphs with bounded degree.

6. Automorphism Groups and Bases

In order to avoid enumerating duplicates, we will use some theory on automorphism groups. In this section, we will briefly review the necessary concepts.

An automorphism of a graph g is an isomorphism between g and itself. We will denote the identity automorphism with I_g. In Figure 2, a graph g^{ex} with 8 vertices is shown, together with

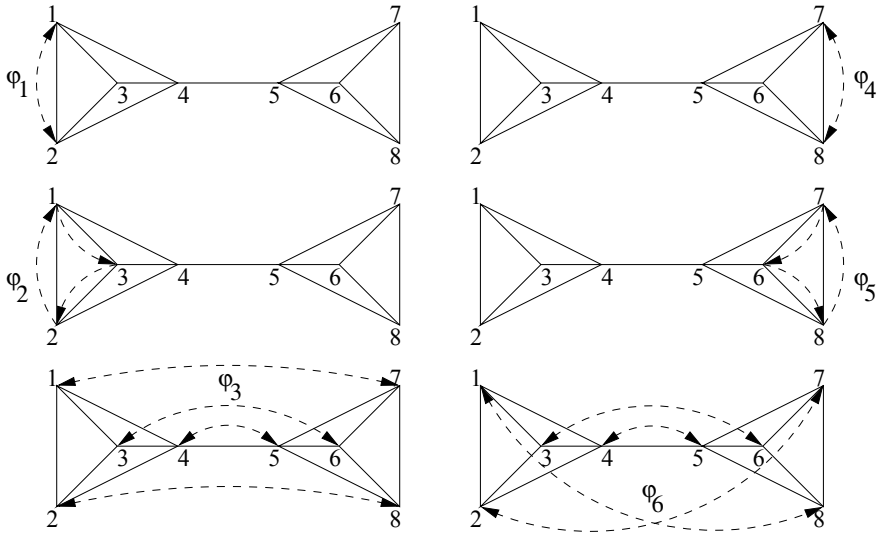

Figure 2: A graph g^{ex} and 6 of its automorphisms.

six of its automorphisms. An automorphism is a permutation of the vertices of g. The set of all automorphisms of g equipped with composition of permutations forms a permutation group acting on $V(g)$. This group is called the automorphism group of g, which we denote with $\mathcal{A}ut(g)$.

Let P be a permutation group and let $S \subseteq P$. We say S is a set of generators of P iff every element of P can be written as a composition of elements of S. We denote this fact with $P =< S >$. For example, in Figure 2, $\{\varphi_1, \varphi_2, \varphi_3, \varphi_5\}$ is a (non-minimal) set of generators of $\mathcal{A}ut(g^{ex})$. Automorphism φ_4 can be composed by $\varphi_4 = \varphi_3 \circ \varphi_1 \circ \varphi_3$. φ_6 can be composed as $\varphi_6 = \varphi_1 \circ \varphi_3 \circ \varphi_1$.

Let P be a permutation group acting on V and let $v \in V$. The stabilizer of v in P is the subgroup $P_v = \{\varphi \in P \mid \varphi(v) = v\}$. Consider for example the automorphism group $P = \mathcal{A}ut(C_5)$ on the 5-cycle graph $C_5 = (\{v_1, v_2, v_3, v_4, v_5\}, \{\{v_1, v_2\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_4, v_5\}\})$. This group has 10 elements, each of which can be decomposed by (possibly) a mirror permutation $\{(v_1, v_1), (v_2, v_5), (v_5, v_2), (v_3, v_4), (v_4, v_3)\}$ and rotations (0, 1 or more applications of $\{(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_1)\}$). For any vertex $v \in C_5$, the stabilizer P_v contains precisely 2 elements. E.g., P_{v_3} contains the identity permutation and the mirror $\{(v_3, v_3), (v_4, v_2), (v_2, v_4), (v_1, v_5), (v_5, v_1)\}$

One can apply the definition of stabilizers recursively; we will denote with $P_{v_1,\ldots,v_k}$ the stabilizer of v_k in $P_{v_1,\ldots,v_{k-1}}$. A sequence of points $B = [v_1 \ldots v_n]$ of V is called a *base* for permutation group P iff $P_{v_1,\ldots,v_{n-1}}$ only contains the identity permutation. For example, in Figure 2, $B_1 = [3, 1, 6, 8]$ is a base for g^{ex}. Indeed, only the identity automorphism $I_{g^{ex}}$ leaves all four vertices 1, 3, 6 and 8 fixed. Also $B_2 = [3, 1, 6, 8, 2, 4, 5, 7]$ is a base for g^{ex}.

Let P be a permutation group, B a base of P and $S \subseteq P$. S is called a strong generating set related to $B = \{v_1, \ldots, v_k\}$ if it contains generators for all permutations in $P_{v_1,\ldots,v_l}$ for $1 \leq l \leq k$. We will use the abbreviation BSGS for a pair (B, S) where B is a base and S is a strong generating set related to B.

Consider the base $B_1 = [3, 1, 6, 8]$ for the group $\mathcal{A}ut(g^{ex})$ in Figure 2. We can construct a strong generating set by first choosing generators for $\mathcal{A}ut(g^{ex})_{3,1,6,8}$, then choosing generators for $\mathcal{A}ut(g^{ex})_{3,1}$, $\mathcal{A}ut(g^{ex})_{3,1,6}$ and so on until we have generators for $\mathcal{A}ut(g^{ex})$. Each time, we use the generators of the subgroup and extend them to a set of generators for the larger group. In our exam-

ple, $\{\varphi_4\}$ is a set of generators for $\mathcal{A}ut(g^{ex})_{3,1,6}$, Next, $\{\varphi_4,\varphi_5\}$ is a set of generators for $\mathcal{A}ut(g^{ex})_{3,1}$ and $\{\varphi_4,\varphi_5,\varphi_1\}$ is a set of generators for $\mathcal{A}ut(g^{ex})_3$. Finally, $\{\varphi_4,\varphi_5,\varphi_1,\varphi_3,\varphi_2\}$ is a set of generators for $\mathcal{A}ut(g^{ex})$ and therefore a strong generating set for B.

A BSGS can represent a permutation group acting on n elements using only $O(n\log(n))$ generators. For example, even though in Figure 2, the automorphism group $\mathcal{A}ut(g^{ex})$ contains $2*3*2*3*2=72$ elements, only 5 generators are needed to represent it. Moreover, an $O(n^5)$ algorithm exists to transform a BSGS into a BSGS with a different base (Butler, 1991). In Figure 2, consider the BSGS $([3,1,6,8],\{\varphi_4,\varphi_5,\varphi_1,\varphi_3,\varphi_2\})$ and suppose we want a strongly generating set for the base $[3,1,8,6]$. Then, we have to combine the generators of $\mathcal{A}ut(g^{ex})_{3,1,6}$ and $\mathcal{A}ut(g^{ex})_{3,1}$. One possible strongly generating set is $\{\varphi_4\circ\varphi_5,\varphi_5,\varphi_1,\varphi_3,\varphi_2\}$.

In general, it is easy to see that we can reduce any strong generating set for a permutation group on n elements to at most $n(n-1)/2$ elements. Let $B=\{v_1,\ldots,v_k\}$ be a base ($k\leq n$), and let S_i ($1\leq i\leq k$) be the subset of the strong generating set containing all generators fixing v_j for all $1\leq j<i$ but mapping v_i on a different element. Now for all i from 1 to k we can do the following. As long as any S_i has more than $n-i$ elements, there are two permutations $p_1,p_2\in S_i$ such that $p_1(v_i)=p_2(v_i)$, and we can remove p_2 from S_i and add $p_2\circ p_1^{-1}$ to S_{i+1}. The permutation group generated by the new strong generating set $\cup_i S_i$ remains the same, as any permutation requiring p_2 in its construction can still be constructed with $p_2\circ p_1^{-1}\circ p_1$. After performing such replacements until all S_i contain at most $n-i$ elements, we have a strong generating set of size at most $n(n-1)/2$.

An important property is that given a strong generating set for some base, one can efficiently compute a strong generating set for another base. A basic step in such base change is to interchange two vertices, that is, given a strong generating set for $B=\{v_1\ldots v_n\}$, find a strong generating set for $B'=\{v_1\ldots v_{l-1},v_{l+1},v_l,v_{l+2}\ldots v_n\}$ for some l with $1\leq l<n$. Let S be a strong generating set with $S=\cup_{i=1}^{n}S_i$ where S_i contains the generators fixing v_j ($1\leq j<i$) and not fixing v_i. The only part of S that should be changed when swapping v_l and v_{l+1} in the base are S_l and S_{l+1}. Let $S'=S'_l\cup S'_{l+1}\cup\bigcup_{i\notin\{l,l+1\}}S_i$ be the strong generating set for the new base B', where S'_l fixes v_i ($1\leq i<l$) and S'_{l+1} fixes v_i ($1\leq i<l$) and v_{l+1}. One can construct S'_i by ensuring it contains permutations that map v_{i+1} on all its possible images. These images can be found with a so-called reachability graph. This means one starts with a set of possible images $I=\{v_{i+1}\}$, and as long as there is a permutation in $\cup_{i=l}^{n}S_i$ which maps an element $v\in I$ on an element $v'\notin I$, one adds v' to I; in the mean time, for each element v' one maintains a corresponding permutation. After constructing S'_l, one can construct S'_{l+1} by starting with $S'_{l+1}=S_l\cup S_{l+1}$ and then removing any permutations from it that are redundant. In this way, it is possible to find in polynomial time a strong generating set for a base in which two vertices have been swapped. By iterating this procedure, it is possible to efficiently find a strong generating set for any new base. Note that we here only described one naive strategy. In the literature, much more advanced algorithms have been proposed, which allow to perform these operations significantly more efficiently.

7. Algorithm Outline

Our algorithm enumerates the graph ordered by size, that is, no graph will be output before all its ancestors under ρ^+ are listed. Indeed, we can show that if $(\rho^+,\rho^-,\uparrow)$ is a dense augmentation schema, every graph has a unique size, that is, there is a function *size* that maps every graph to an integer such that for every g in $\mathcal{L}_{\rho^+}$ it holds that $r\in\rho^+(g)$ implies that $size(g+r)=size(g)+1$. This results allows us to order the graphs that we need to enumerate level-wise.

Superficially, the idea is then as follows. We maintain graphs that we are enumerating in a queue. Initially, this queue contains the graph $\top$. Repeatedly, we pop a graph from the queue, apply the augmentation operator, and those children which are not equivalent to any graph in the queue, are pushed in the queue. Due to the level-wise enumeration, equivalent graphs must be in the queue. We need to address two issues:

- how do we avoid that we insert a child which is equivalent to a child of another graph?
- how do we avoid that we insert two children of the same graph that are equivalent with each other?

To make these computations possible in polynomial time, we also keep all parents of the graphs in the queue in memory. For each graph, both those in the queue and their parents, we store the following information:

- a representation for the graph g
- for each augmentation (and reduction) r of g, we store r together with an isomorphism mapping $\varphi = aug(g,r)$ (resp. $\varphi = red(g,r)$) such that $\varphi(g+r)$ (resp. $\varphi(g-r)$) is the stored representative of the isomorphism class of $g+r$ (resp. $g-r$). Until we assign the $aug(g,r)$ and $red(g,r)$ variables with a value, we will assume them initialized $aug(g,r) =' ?'$ and $red(g,r) =' ?'$. If $p(g+r)$ is false, with p the monotonic predicate of interest, we will assign $aug(g,r)$ the value *nil*.
- a base and strong generating set (BSGS) $bsgs(g)$ of the automorphism group $\mathcal{A}ut(g)$.

We output a graph when we pop it from the queue. Furthermore, we determine the BSGS at that point. The BSGS allows us to compute for each pair of augmentations of a graph if they result in equivalent graphs, and thus allows us to avoid two equivalent children of the same graph from being pushed in the queue.

To avoid that two different graphs insert children in the queue that are equivalent, we make sure that the first parent of a child marks at least one augmentation in each of the other parents of the child. When this alternative parent is popped from the queue later, it can use this information to avoid pushing this child and all its equivalent children.

To prove that this procedure is polynomial, we need to show that we can compute the BSGS in polynomial time, and that we can find all parents of a child in polynomial time. Let us start with this second point.

One of the conditions of Theorem 1 is that the augmentation schema is dense, that is, that the graph of parents is connected; hence we can compute a spanning tree for the graph of parents. If we create a child, we know the parent that generated this child; by traversing the spanning tree starting from the reduction achieving this parent, we can traverse all possible reductions. At the same time, for every step we take in this spanning tree, we can determine a corresponding stored parent: every step in the graph of parents corresponds to a reduction followed by an augmentation, for which we have stored associated permutations that point to stored representatives.

To compute the BSGS the key observation is that we can incrementally compute the BSGS of a graph from the BSGS of one of its parents, in a similar way to the algorithm of Goldberg (1992). Given a child and its parent, we perform a base change for the parent, such that we obtain permutations in which the vertices contained in the augmentation are stabilized. This gives generators

for all vertices in the child, except those contained in the augmentation. By traversing all parents (as computed when the child was created), we can determine permutations and images for these vertices as well. The resulting set of generators can be reduced to a BSGS in polynomial time.

Details of this algorithm, including optimizations and proofs of complexity, can be found in the appendix.

8. Conclusions

We introduced an algorithm for listing graphs that makes sure that no two equivalent graphs are being output. We showed that for a well-defined set of conditions on a class of graphs to enumerate, this algorithm is correct and achieves polynomial delay. Classes of graphs that can be enumerated with low run-time complexity are connected graphs, planar graphs, minor closed graphs, monotonic classes of graphs in general, and hereditary classes with bounded degree. To the best of our knowledge, this is the first algorithm to be general enough to be able to list this range of classes efficiently, and for several of these classes no polynomial delay algorithm was presented before. In the appendix, we show that our algorithm runs with delay $O(n^5)$ for the class of all graphs, which is an improvement over the known method of Goldberg (1992), which achieves a delay of $O(n^6)$, where n is the number of vertices in the largest graph that is listed.

Most pattern mining algorithms consist of a candidate generation part and an interestingness evaluation part. This work contributes to the theory of pattern mining by providing a polynomial-delay algorithm for the first of these two common tasks. Also, our algorithm can be used as a generic candidate pattern generator for a wide range of algorithms for mining structured patterns, avoiding the need to research specialized canonical forms and enumeration strategies.

In contrast to other graph pattern mining systems, our algorithm provides at the same time a data structure in which one can look up a pattern in polynomial time. Indeed: given a pattern g, one can construct g from the empty graph by a sequence of augmentations, and then follow the augmentation pointers through the data structure. Efficient lookup could be very useful when the set of patterns resulting from a pattern mining step is queried by the user or by algorithms taking this set of patterns as input.

Even though its run-time complexity is favorable, the space complexity of our algorithm is an issue. The space required is polynomial in the size of the output; given that the number of listed graphs can be exponential, the storage requirements for large classes of graphs can be exponential. However, in those applications where the listed graphs need to be stored anyway, such as applications in data analysis, this drawback is of minor concern. Moreover, we know of no other general approaches that obtain a better space complexity.

As future work, we conjecture that the run-time complexity of our algorithm can be reduced further, at least to a delay of $O(n^4)$ for the class of all graphs and the class of connected graphs, through the definition of a canonical representation over the graphs. Furthermore, we hope to reduce the space requirements of our algorithm in problem settings where not all listed graphs are required to be stored, and plan to implement it in concrete data mining systems. Finally, relieving the constraint of bounded degree for hereditary classes remains an interesting problem.

Acknowledgments

Jan Ramon and Siegfried Nijssen are post-doctoral fellows of the Fund for Scientific Research of Flanders (FWO-Vlaanderen). Siegfried Nijssen was also supported by the European Commission under the 6th Framework Programme, project "Inductive Querying", contract number FP6-516169, and under the 7th Framework Programme FP7-ICT-2007-C FET-Open, project "Bison", contract number BISON-211898. We are grateful to Maurice Bruynooghe for proof-reading our final version.

Appendix A. Algorithmic Details and Proofs

In this appendix we provide details of our algorithm and proofs for our results.

A.1 Dense Augmentation Schemas

An important property of a dense augmentation schema is the following:

Lemma 2 *Let $\mathcal{L}$ be a class of graphs, and let $(\rho^+, \rho^-, \uparrow)$ be a dense augmentation schema for $\mathcal{L}$. Then, there exists a function size : $\mathcal{L} \rightarrow N$ such that $\forall g \in \mathcal{L}, \forall r \in \rho^+(g) : size(g+r) = size(g)+1$.*

Proof We will say that a graph g can be constructed from $\top$ in n steps if there exists a sequence $\top = g_0, g_1, \ldots, g_n = g$ such that $g_{i+1} = g_i + r_i$ for some $r_i \in \rho^+(g_i)$ for $i = 0 \ldots n-1$.

We first show that there is no graph g which both can be constructed from $\top$ in n_1 steps and can be constructed from $\top$ in n_2 steps for two distinct numbers n_1 and n_2.

Assume that such a graph g exists, and consider a minimal such graph g (according to the order induced by the augmentation schema). Then, there exists a parent $g - r'_1$ of g which can be constructed from $\top$ in $n_1 - 1$ steps, and a parent $g - r'_2$ of g which can be constructed from $\top$ in $n_2 - 1$ steps. As the graph of parents $GoP_{\rho^-,\uparrow}(g)$ of g is connected, there must exist two $r_1, r_2 \in \rho^-(g)$ such that (r_1, r_2) is an edge of $GoP_{\rho^-,\uparrow}(g)$, $g - r_1$ can be constructed from $\top$ in n'_1 steps, $g - r_2$ can be constructed from $\top$ in n'_2 steps and $n'_1 \neq n'_2$. As (r_1, r_2) is an edge of $GoP_{\rho^-,\uparrow}(g)$, $(r_1 \uparrow^g r_2) \in \rho^-(g - r_2)$ and $(r_2 \uparrow^g r_1) \in \rho^-(g - r_1)$, there exists some graph p such that $p \simeq (g - r_2) - (r_1 \uparrow^g r_2)$ and $p \simeq (g - r_2) - (r_2 \uparrow^g r_1)$. Let n_p be the number of steps needed to construct p from $\top$. Then, remembering that augmentation schemas are isomorphism-invariant, we can conclude that both $g - r_1$ and $g - r_2$ can be constructed from $\top$ in $n_p + 1$ steps. Now as $n_1 \neq n_2$ either $n_1 - 1 \neq n_p + 1$ or $n_2 - 1 \neq n_p + 1$. Without loss of generality we can assume $n_1 \neq n_p + 1$. This means that $g - r_1$ can be constructed from $\top$ in both $n_p + 1$ and $n_1 - 1$ steps, which is a contradiction with the assumption that g is a minimal graph for which it holds that it can be constructed from $\top$ in two different numbers of steps.

Therefore, we conclude that no such graph exists. Hence, in order to obtain a function *size* satisfying the requirement, one can define $size(\top)$ to be 0 and for every g, $size(g)$ to be the number of steps in which g can be constructed from $\top$. ■

In Section 5 we stated the following.

Lemma 3 *$(\rho_c^+, \rho_c^-, \uparrow_c)$ is a dense augmentation schema.*

Proof We consider the different elements of the definition of a dense augmentation schema.

First, remember that (ρ_c^+, ρ_c^-) was defined by Equation (2) and (3) (see page 913 and 914). This is clearly an augmentation schema. Also, we defined $r_1 \uparrow_c^g r_2$ to be equal to r_1, except in the case where $r_1 = (\emptyset, \{v, u_1\})$, $r_2 = (\emptyset, \{v, u_2\})$ and v has degree 2. In that case $r_1 \uparrow_c^g r_2 = r_1 \cup (\{v\}, \{\})$. It is easy to see that $\uparrow_c$ satisfies Equations (4), (5) and (6), and hence is a reduction translator.

It remains to be shown that for every non-minimal graph $g \in \mathcal{L}_c$ it holds that either the graph of parents $GoP_{\rho_c^-, \uparrow_c}(g)$ is connected, or $g - r$ is the minimal element of $\mathcal{L}_c$ for all $r \in \rho^-(g)$.

Consider a connected graph g with at least 2 edges. We prove that $GoP_{\rho_c^-, \uparrow_c}(g)$ is connected. If g is a tree, then every reduction in $\rho^-(g)$ removes a leaf and the edge adjacent to it. By the definition of $\uparrow_c$, $r_1 \uparrow_c^g r_2 = r_1$ and $r_1 \in \rho_c^-(g - r_2)$ will hold for any distinct $r_1, r_2 \in \rho_c^-$. So if g is a tree, $GoP_{\rho_c^-, \uparrow_c}(g)$ is a clique.

If g is not a tree it contains at least one simple cycle C. We can partition $\rho_c^-(g)$ into two sets R_1 and R_2 such that R_1 contains all reductions removing an edge from the cycle C, and R_2 contains all other reductions. For any $r_1 \in R_1$ and $r_2 \in R_2$, it holds that removing r_2 from $g - r_1$ does not disconnect g and vice-versa. So such two r_1 and r_2 are adjacent in $GoP_{\rho_c^-, \uparrow_c}(g)$. Therefore, $GoP_{\rho_c^-, \uparrow_c}(g)$ is certainly connected if $\#R_2 \geq 1$. On the other hand, if $\#R_2 = 0$, g is a simple cycle. In that case, two reductions of R_1 are adjacent iff they remove two adjacent edges of the cycle. So in that case, $GoP_{\rho_c^-, \uparrow_c}(g)$ is a cycle and hence connected. ■

A.2 Algorithm

In this section we explain in more detail our algorithm. As stated in Section 7, for each graph, we store the following information:

- a representation for the graph g
- for each augmentation and reduction r of g, we store r together with an isomorphism mapping $aug(g,r)$ (resp. $red(g,r)$) that maps $g + r$ (resp. $g - r$), to the stored representative of their isomorphism class. In the algorithm, we use $aug(g,r)$ and $red(g,r)$ as functions that return the isomorphism. When the isomorphism is not yet computed, $aug(g,r)$ and $red(g,r)$ will return $'?'$. If $p(g+r)$ is false, $aug(g,r)$ returns the value *nil*.
- a base and strong generating set (BSGS) $bsgs(g)$ of the automorphism group $\mathcal{A}ut(g)$.

Algorithm 2 shows the high level algorithm. It repeatedly takes an unprocessed graph $g + r$ and processes it; $g + r$ must be minimal according to the *size* function from Lemma 2. A graph which has been processed remains in memory till it is no longer needed in the computation for one of its ancestors.

As shown by Algorithm 3, the processing of a graph g includes computing a BSGS and the automorphism group of g, computing reachability graphs of $\mathcal{A}ut(g)$ (line 3), finding isomorphic variants of children (lines 8-12), and examining all other children of g including constructing all $red(\cdot,\cdot)$ and some $aug(\cdot,\cdot)$ links (lines 13-24). We will describe each of the steps of the algorithm below.

Let us first consider what is known at the point PROCESS_GRAPH(g,r_0) is called. First, all graphs h for which $p(h)$ and $size(h) < size(g)$ have been processed and outputted, and the values for $aug(h,\cdot)$, $red(h,\cdot)$ and $bsgs(h)$ have been computed. Second, all graphs f for which $p(f)$ and

Algorithm 2 Highlevel algorithm

Require: a graph class $\mathcal{L}$, a monotonic predicate p and a dense augmentation schema $(\rho^+, \rho^-, \uparrow)$
Ensure: output all $g \in \mathcal{L}$ with $p(g)$

1: $RefQueue \leftarrow \{(\top, (\emptyset, \emptyset))\}$
2: **while** $RefQueue \neq \emptyset$ **do**
3: Let $(g,r) \in RefQueue$ such that $size(g+r)$ is minimal
4: $RefQueue \leftarrow RefQueue \setminus \{(g,r)\}$
5: PROCESS_GRAPH(g,r)
6: Output g
7: **end while**

$size(f) = size(g)$ have either entered $RefQueue$ or are fully processed. In any case, all values $red(f,\cdot)$ have been computed.

Let S_V denote the bound on the number of vertices and edges that can occur at most in any augmentation $r \in \rho^+(g)$ with $g \in \mathcal{L}$. Then we have the following lemma.

Lemma 4 *At line 2 of Algorithm 3 one can compute a BSGS for $\mathcal{A}ut(g)$ in time $O(|V(g)|^5 + |V(g)|^3 \cdot |\rho^-(g)| \cdot S_V!)$.*

Proof The case $g = \top$ is trivial, so we assume $g \neq \top$ and $r_0 \in \rho^-(g)$. A parent $g - r_0$ is known, and so is $bsgs(g - r_0)$ (as $size(g - r_0) = size(g) - 1$ and hence $g - r_0$ has been fully processed). By performing a base change, we can obtain a BSGS for the stabilizer $\mathcal{A}ut(g)_{V^*(r_0)}$ which fixes all elements in $V^*(r_0)$; please note that $V^*(r)$ contains all nodes involved in the reduction. Hence, we compute a base in which all nodes involved in the reduction are fixed. This is possible in time $O(|V(g)|^5)$. We can then extend the set of generators corresponding to this BSGS to a set of generators for $\mathcal{A}ut(g)$ by adding for every coset of $\mathcal{A}ut(g)_{V^*(r_0)}$ in $\mathcal{A}ut(g)$ one representative. Each such representative φ maps r_0 on some different $\varphi(r_0)$. Graphs $g - r_0$ and $g - \varphi(r_0)$ are isomorphic, which is reflected by $red(g, r_0)(g - r_0) = red(g, \varphi(r_0))(g - \varphi(r_0))$; here, $red(g, r_0)$ returns the stored permutation for reduction r_0 on graph g, which after application on the nodes and edges in graph $g - r_0$ yields the same graph as for the equivalent reduction $\varphi(r_0)$. One can therefore find all such representatives by iterating over all $r \in \rho^-(g)$, checking whether $red(g, r_0)(g - r_0) = red(g, r)(g - r)$ (all $red(g,\cdot)$ were computed earlier) and for all of these listing all possibilities to extend $red(g,r)^{-1} \circ aug(g, r_0)$ to an automorphism $(red(g,r)^{-1} \circ aug(g, r_0)) \cup \varphi_0$ (where $\varphi_0 : V(r) \to V(r_0)$). In total, the number of representatives of cosets of $\mathcal{A}ut(g)_{V^*(r_0)}$ is bounded by $|\rho^-(g)| \cdot S_V!$. One can eliminate the redundant automorphisms in this list in time $O(|V(g)|^3 \cdot |\rho^-(g)| \cdot S_V!)$. ■

Line 3 of Algorithm 3 computes reachability graphs Q_i, which are used in line 9 of the algorithm to determine if two augmentations yield isomorphic graphs. These graphs help to exploit the knowledge of the just computed BSGS $bsgs(g)$ of $\mathcal{A}ut(g)$. A reachability graph is computed for the number of nodes of graph g involved in the augmentation. In the case of (un)connected graphs, an augmentation involves either one or two nodes of graph g, and hence a reachability graph is used for $i = 1$ or $i = 2$. Q_i can be computed in time $O(i \cdot |V(g)|^i \cdot |bsgs(g)|)$. As for a reduced BSGS we have $|bsgs(g)| \leq |V(g)|^2$, line 3 can be performed in time $O(S_V \cdot |V(g)|^{S_V+2})$.

Algorithm 3 Processing one graph

```
1:  procedure PROCESS_GRAPH(g,r_0)
2:      bsgs(g) ← Compute_BSGS(g, r_0)
3:      Ensure reachability graph Q_i exists for g where i = |V*(r_0) ∩ V(g)|,
        V(Q_i) = (V(g))^i, E(Q_i) = {(W, φ(W)) | W ∈ V(Q_i) ∧ φ ∈ bsgs(g)}
4:      R_iso+ ← {r ∈ ρ+(g) | aug(g,r) ≠'?'}
5:      done ← FALSE
6:      while not(done) do
7:          if R_iso+ ≠ ∅ then
8:              Let r ∈ R_iso+
9:              for all r' ∈ ρ+(g) s.t. ∃φ : (∀v ∈ V(g) : φ(v) ∈ V(g)) ∧ g + r ≃_φ g + r' do
10:                 aug(g,r') ← aug(g,r) ∘ φ^-1
11:                 R_iso+ ← R_iso+ \ {r'}
12:             end for
13:         else if ∃r ∈ ρ+(g) : aug(g,r) ='?' then
14:             s = g + r
15:             (ok, s_par_list) ← SEARCH_PARENTS(g,r,s)
16:             if ok ∧ p(g) then
17:                 for all (r', φ') ∈ s_par_list do
18:                     f ← φ'(s − r') ; aug(f, φ'(r')) = φ'^-1 ; red(s,r') = φ'
19:                 RefQueue = RefQueue ∪ {(s,r)}
20:             else
21:                 for all (r', φ') ∈ s_par_list do
22:                     aug(φ'(s − r'), φ'(r')) ← nil
23:             end if
24:             R_iso+ ← R_iso+ ∪ {r}
25:         else
26:             done ← TRUE
27:         end if
28:     end while
29: end procedure
```

It is possible that previous calls of PROCESS_GRAPH have assigned a value already to some of the $aug(g,\cdot)$, and line 4 collects these augmentations so that their isomorphic variants can be computed in lines 8-12.

Next, the while loop at line 6 runs until all $aug(g,\cdot)$ values have been computed. As soon as a new child is identified (either by previous calls to PROCESS_GRAPH (line 4) or by newly examined children (line 24) it is added to R_{iso+}, and in the next iteration all its isomorphic variants are computed. If R_{iso+} is empty, the $r \in \rho^+(g)$ for which $aug(g,r) =' ?'$ are isomorphic to none of the r' for which $aug(g,r')$ has already been assigned a value and a new child is considered (lines 13-24).

We will first discuss the computation of isomorphic variants of an $r \in R_{iso+}$ in line 9. Recalling the definition, all r' are searched for which there is a mapping φ such that $(\forall v \in V(g) : \varphi(v) \in V(g)) \wedge g + r \simeq_\varphi g + r'$. One can do this as follows. First, use the reachability graph Q_i (with $i = |V^*(r) \cap V(g)|$) to find all possible images of $V^*(r) \cap V(g)$ under the automorphism group $\mathcal{A}ut(g)$.

Algorithm 4 search_parents

```
1: procedure SEARCH_PARENTS(g, r_0, s)
2:     Construct a spanning tree T for GoP_{ρ⁻,↑}(s), the graph of parents of s
3:     s_par_list ← {(r_0, I_g)}}
4:     Perform a depth-first search of T, starting at r_0
5:     for all edges (r_1, r_2) visited during the depth first search do
6:         φ ← GET_PARENT(s, r_1, r_2)
7:         if φ = nil then return (FALSE,s_par_list)
8:         s_par_list ← s_par_list ∪ {(r_2, φ)}
9:     end for
10:    return (TRUE,s_par_list)
11: end procedure
```

Algorithm 5 Get one parent

```
1: procedure GET_PARENT(s,r_1,r_2)
2:     φ_f1 ← ext(s_par_list(s, r_1), I_s)
3:     f_1 ← φ_f1(s − r_1)
4:     φ_p ← ext(red(f_1, φ_f1(r_2 ↑^s r_1)), φ_f1)
5:     p ← φ_p((s − r_1) − φ_f1(r_2 ↑^s r_1))
6:     Let r'_1 ∈ ρ⁺(p) and φ_* : V(φ_p(r_1 ↑^s r_2)) → V(r'_1) such that (I_p ∪ φ_*)(φ_p(r_1 ↑^s r_2)) = r'_1
7:     if aug(p, r'_1) = nil then return nil
8:     else return ext(aug(p, r'_1), (I_p ∪ φ_*) ∘ φ_p)
9: end procedure
```

For each of these images of r, one can also compute one automorphism $\varphi_g \in \mathcal{A}ut(g)$ under which $\varphi_g(r)$ corresponds to the image, by following the edges of Q_i. Then, one can check for every $r' \in \rho^+(g)$ whether there is a mapping $\varphi_r : V(r) \to V(r')$ such that for $\varphi = \varphi_g \cup \varphi_r$ we have $\varphi(r) = r'$. Traversing Q_i and computing the automorphisms φ_g along the way is possible in time $O(|V(g)|^{i+1})$. As $i \leq S_V$ and we do this at most once for every $r \in \rho^+(g)$, the total time spent here is bounded by $O(|V(g)|^{S_V+1}|\rho^+(g)|)$.

Let us now consider the investigation of new children in lines 13-24 of Algorithm 3. For a particular child $s = g + r$, the algorithm first searches the parents $f = s - r'$ for all $r' \in \rho^-(s)$. As we explain below, if all parents of s satisfy the predicate p the algorithm is guaranteed to find all these parents. The $aug(f, r')$ variables have been assigned a value and depending on whether p holds for s itself, the $red(s, r')$ variables have been assigned a value and it is added to the data structures and to $RefQueue$.

Finding other parents of a proposed child $s = g + r$ is detailed in Algorithms 4 and 5. Before analysing this procedure and its consequences, we first explain some basic ideas. The key intuition that enables an efficient enumeration is that we can efficiently identify the (already enumerated) parents f_2 of the candidate graph s together with a suitable isomorphism mapping φ such that $s - r_2 \simeq_\varphi f_2$ for every reduction $r_2 \in \rho^-(s)$ by using the knowledge from Equation (6) that one can obtain $p = (s - r_1) - (r_2 \uparrow^s r_1)$ also by removing the parts in a different order: $p = (s - r_2) - (r_1 \uparrow^s r_2)$. Therefore, in Algorithm 5 we first remove some $r_1 \in \rho^-(s)$ to obtain a known parent f_1 of s, then go to a grand-parent p by removing a (translated) r_2 from f_1, and then go downwards

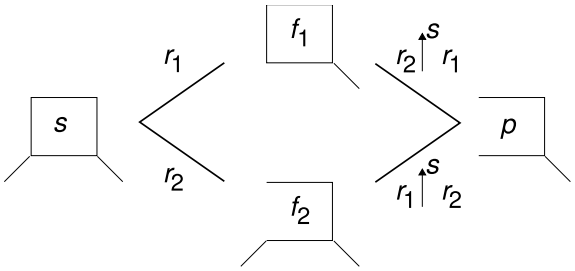

Figure 3: Searching for parents

again from p to the unknown parent f_2 (see Figure 3 for an illustration). In order to construct an isomorphism between $s - r_2$ and f_2, isomorphisms between $s - r_1$ and f_1, between $f_1 - (r_2 \uparrow^s r_1)$ and p and between $p + (r_1 \uparrow^s r_2)$ and f_2 are first extended to cover the full set of vertices of s (the $ext(\cdot,\cdot)$ function), and then composed (line 8).

In Algorithm 5 we have the following notation. Let φ_2 and φ_1 be bijections between sets of vertices. Then, $ext(\varphi_2, \varphi_1)$ is a bijection that maps any x for which $\varphi_1(x)$ is in the domain of φ_2 on $\varphi_2(\varphi_1(x))$ and maps any other x on a new vertex.

Now we return to the details of Algorithms 4 and 5. First, remember that for a dense augmentation schema, the graph of parents of a particular graph is connected. Therefore, it is possible in line 2 of Algorithms 4 to construct a spanning tree for it. Algorithm 4 attempts to construct in *s_par_list* a mapping from reductions $r \in \rho^-(s)$ to isomorphism mappings between the graphs $s - r$ and the representatives of their isomorphism class which were output before. We know already the isomorphism mapping between $s - r_0 = g$ and the representative of its isomorphism class, which is g itself (line 3). Now, if for some $r_1 \in \rho^-(s)$ we know an isomorphism mapping between $s - r_1$ and its representative f_1, and if for some other $r_2 \in \rho^-$, the graph $s - r_2$ satisfies p and has been listed earlier, then Algorithm 5 will provide us with an isomorphism mapping between $s - r_2$ and its representative f_2 as discussed above.

Then there are two possible cases. On the one hand, if there is a parent of s which did not fulfil the predicate p and hence was not listed earlier, we will not find that parent: line 7 of Algorithm 5 will detect this and return *nil*, which will cause also Algorithm 4 to return FALSE. On the other hand, if all parents of s fulfil predicate p, the search along the spanning tree will eventually provide an isomorphism mapping between s and the representatives of $s - r$ for all $r \in \rho^-(s)$.

The complexity of this search can be assessed as follows: Algorithm 5 is executed once for every $r' \in \rho^-(s)$ and contains operations on permutations that can be performed in time $O(|V(s)|)$. Assuming that $\uparrow$ can be performed efficiently, that a good data structure is built on $\rho^+(p)$ in order to be able to perform line 6 of Algorithm 5 efficiently, that $|\rho^-(s)|$ can be bounded by $O(|\rho^-(g)|)$ and that $|V(s)|$ can be bounded by $O(|V(g)|)$, we can therefore conclude that Algorithm 4 can be performed in time $O(|\rho^-(g)| \cdot |V(g)|)$. These assumptions are not very strong and hold for our two example augmentation schemas $(\rho_a^+, \rho_a^-, \uparrow_a)$ and $(\rho_c^+, \rho_c^-, \uparrow_c)$. Algorithm 4 is ran at most once for every $r \in \rho^+(g)$, for a total complexity of $O(|\rho^+(g)| \cdot |V(g)| \cdot |\rho^-(g)|)$.

In summary, this appendix has provided an informal proof of the following.

Theorem 5 *Under the assumptions stated in Theorem 1, Algorithm 2 correctly lists for every isomorphism class of graphs g for which* $p(g) =$ TRUE *exactly one representative, and the time needed*

for outputting the next graph g is bounded by $O(|V(g)|^5 + |V(g)|^3 \cdot |\rho^-(g)| \cdot S_V! + S_V \cdot |V(g)|^{S_V+2} + |V(g)|^{S_V+1}|\rho^+(g)| + |\rho^+(g)| \cdot |V(g)| \cdot |\rho^-(g)|)$.

This bound is polynomial for a constant S_V.

For example, for enumerating connected graphs with the schema $(\rho_c^+, \rho_c^-, \uparrow_c)$, $S_V = 2$. Therefore, this algorithm enumerates classes of connected graphs in time $O(|V(g)|^5)$ for each output graph g in the worst case. A similar result can be shown for enumerating (a monotonic subset of) all graphs with $(\rho_a^+, \rho_a^-, \uparrow_a)$.

Our conjecture is that this complexity can be improved further to $O(|V(g)|^4)$.

References

Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, 1996.

Christian Borgelt and Michael R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *ICDM*, pages 51–58, 2002.

Gregory Butler. *Fundamental algorithms for permutation groups*, volume 559 of *Lecture Notes in Computer Science*. Springer-Verlag, 1991.

Yun Chi, Richard R. Muntz, Siegfried Nijssen, and Joost N. Kok. Frequent subtree mining - an overview. *Fundam. Inform.*, 66(1-2):161–198, 2005.

Leslie A. Goldberg. Efficient algorithms for listing unlabeled graphs. *Journal of Algorithms*, 13(1): 128–143, 1992.

Leslie A. Goldberg. Polynomial space polynomial delay algorithms for listing families of graphs. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing (STOC'93)*, pages 218–225, New York, NY, USA, 1993. ACM Press.

Tamás Horváth, Jan Ramon, and Stefan Wrobel. Frequent subgraph mining in outerplanar graphs. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 197–206, Philadelphia, PA, August 2006.

Jun Huan, Wei Wang, and Jan Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, pages 549–552. IEEE Computer Society, 2003.

Akihiro Inokuchi. Mining generalized substructures from a set of labeled graphs. In *ICDM*, pages 415–418. IEEE Computer Society, 2004.

Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. Complete mining of frequent patterns from graphs: Mining graph data. *Machine Learning*, 50(3):321–354, 2003.

Johannes Köbler, Uwe Schöning, and Jacobo Torán. *The Graph Isomorphism Problem: Its Structural Complexity*. Birkhäuser, 1993.

Michihiro Kuramochi and George Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Trans. Knowl. Data Eng.*, 16(9):1038–1051, 2004.

Jure Leskovec, Ajit Singh, and Jon M. Kleinberg. Patterns of influence in a recommendation network. In *Advances in Knowledge Discovery and Data Mining, 10th Pacific-Asia Conference (PAKDD)*, pages 380–389, 2006.

Brendan D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26:306–324, 1998.

Shin-ichi Nakano and Takeaki Uno. Constant time generation of trees with specified diameter. In Juraj Hromkovic, Manfred Nagl, and Bernhard Westfechtel, editors, *Proceedings of the 30th International Workshop on Graph Theoretical Concepts in Computer Science*, volume 3353 of *Lecture Notes in Computer Science*, pages 33–45. Springer-Verlag, 2004.

Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 647–652, 2004.

Christophe Paul, Andrzej Proskurowski, and Jan A. Telle. Generating graphs of bounded branchwidth. In *Proceedings of the 32nd International Workshop on Graph Theoretical Concepts in Computer Science*, volume 4271 of *Lecture Notes in Computer Science*, pages 206–216, 2006.

Jan Ramon and Siegfried Nijssen. General graph refinement with polynomial delay. In *Proceedings of the Workshop on Mining and Learning with Graphs (MLG'07)*, 2007.

Robert A. Wright, Bruce Richmond, Andrew Odlyzko, and Brendan D. McKay. Constant time generation of free trees. *SIAM Journal on Computing*, 15(2):540–548, 1986.

Xifeng Yan and Jiawei Han. gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, pages 721–724, Japan, 2002. IEEE Computer Society.