General Graph Refinement with Polynomial Delay

Jan Ramon
K.U.Leuven
Celestijnenlaan 200 A
Leuven, Belgium

Siegfried Nijssen
K.U.Leuven
Celestijnenlaan 200 A
Leuven, Belgium

{jan.ramon,siegfried.nijssen}@cs.kuleuven.be

1. INTRODUCTION

Of many graph mining algorithms an essential component is its procedure for enumerating graphs such that no two enumerated graphs are isomorphic. All frequent subgraph miners require such a component [14, 5, 1, 6], but also other data mining algorithms, such as for instance [7] require such a procedure, which is often called a "refinement operator" or a "join operator" depending on the enumeration (or candidate generation) strategy. Consequently, in the data mining literature a huge amount of algorithms have been presented for enumerating graphs without isomorphisms. None of these algorithms, however, have been shown to run with polynomial delay, i.e. when enumerating the graphs, even without accessing any data, in the worst case it may take exponential time to list the next graph. From a theoretical perspective, this is a disappointing result, as Goldberg [3, 4] showed already in the early nineties that enumerating all connected graphs without isomorphs can be done with $O(n^6)$ polynomial delay. In this paper, we address this problem. We show that there is a graph enumeration algorithm that can be used in graph mining algorithms and has $O(n^5)$ polynomial delay. Thus, we not only propose an algorithm to enumerate (the interesting) parts of the class of connected graphs, but also improve the earlier bound on graph enumeration that was shown by Goldberg. At the same time, our algorithm is general enough to be also applicable for other types of graphs than connected graphs. For instance, it can also be used for enumerating unconnected graphs, planar graphs, outerplanar graphs, and many other types of graphs, with polynomial delay. The datastructure that is produced by the algorithm allows membership queries to be executed in polynomial time: for any given graph, independent of its representation, we can determine in polynomial time if it is part of a set of enumerated subgraphs. In particular, in our algorithm it is not necessary to compute a canonical form. However, if this is desirable, we can use our algorithm to enumerate graphs in several canonical forms, for instance, the DFS and BFS canonical forms used in gSpan [14] and MoFA [1].

.

2. ALGORITHM

In this section we will briefly illustrate the main points of our algorithm. Due to space limitations, we will not present the fully general algorithm that works for arbitrary classes of graphs with what we call "dense" refinement schemas. Instead, we will limit ourselves to the problem of enumerating unlabeled, connected graphs with at least one edge. In this case, we can describe a graph by only listing its edges; it is not necessary to explicitly list nodes. For instance, graph G_{51} of Figure 1, can be described by $G_{51} = \{(1,2),(2,3),(3,4),(4,5),(5,2)\}$. We assume that in a graph with n nodes the nodes are numbered from 1 till n. The main problem that we are studying is that two graphs can be isomorphic. Given two graphs with equal numbers of nodes n, an isomorphism is a bijection $\varphi : \{1 \dots n\} \rightarrow \{1 \dots n\}$ such that one graph becomes equal to the other. For instance, $G_{21} = \{(1,2),(2,3)\}$ and $G_{11} \cup \{1,3\} = \{(1,2),(1,3)\}$ (where G_{21} and G_{11} are taken from Figure 1 again) are isomorphic for the permutation with $\varphi(1) = 2$, $\varphi(2) = 3$ and $\varphi(3) = 1$ as $\varphi(\{(1,2),(1,3)\}) = \{(\varphi(1),\varphi(2)),(\varphi(1),\varphi(3))\} = \{(1,2),(2,3)\}$. As all nodes in a graph with n nodes are numbered from 1 to n, the bijection becomes a permutation and we can shorten this description of our φ permutation to a vector of numbers: $\varphi = 231$.

A graph G' is subgraph isomorphic with a graph G iff after the removal of some edges of G one obtains a graph isomorphic to G'. In Figure 1, G_{42} is a subgraph of G_{51}, as we can first apply $\varphi = 51234$ to G_{51}, followed by a removal of $(1,5)$ (which corresponds to a removal of $(1,2)$ in G_{51}), to obtain G_{42}.

We are interested in enumerating *all* graphs that satisfy an anti-monotonic constraint p. A constraint p is anti-monotonic if for all G: $p(G) = true$ implies that $p(G') = true$ for all G' that are a subgraph of G, or, in other words, if $p(G') = false$ then $p(G) = false$ for all supergraphs G of G'. Anti-monotonic constraints are often used in data mining. Given a multiset of graphs $\mathcal{G}$ and a threshold θ, the most popular predicate in data mining is

$$p(G) = (|\{G' \in \mathcal{G} | G \text{ is a subgraph of } G'\}| \geq \theta),$$

which expresses that a subgraph must occur frequently in a database. The problem of finding *all* subgraphs that are frequent, is called the frequent subgraph mining problem. When enumerating these graphs, it is desirable out of efficiency concerns that no two graphs are enumerated that are isomorphic. To be able to exploit anti-monotonic constraints, it is furthermore desirable that a graph is only enumerated after its subgraphs have been enumerated, and that

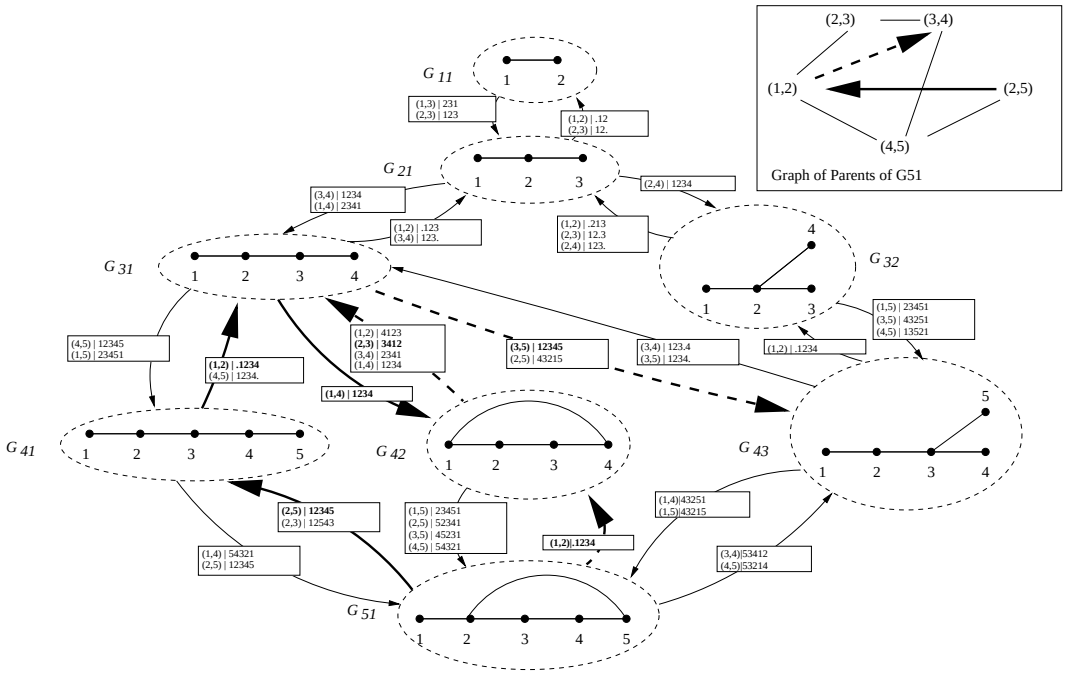

Figure 1: The lattice of all subgraphs of G_{51}

when generating a graph, we can retrieve whether the predicate p was satisfied for each of the subgraphs; this could allow us to prune the graph from the search space. With our algorithm, given a graph of size n, it is possible to look up the value of p for all subgraphs of size $n-1$ in polynomial time.

The most straightforward way to enumerate graphs, is to iteratively use a downward refinement operator. A downward refinement operator $\rho(G)$ is in this case an operator which lists all edges that can be added to the graph G. For instance, $\rho(\{(1,2)\}) = \{(1,3),(2,3)\}$. Each of the edges $e \in \rho(\{(1,2)\})$ corresponds to an extended graph $\{(1,2)\} \cup e$; however, obviously, two graphs that are obtained in this way, can be isomorphic. Our end goal is therefore to build a data structure such as in Figure 1. In this data structure, all subgraphs that satisfy an anti-monotonic constraint p are contained exactly once[1]. Associated to every subgraph G we store a set of downward pointers. A pointer consists of an edge $e \in \rho(G)$, a reference to the location in memory of the single representant of $G \cup e$, and a permutation ϖ such that $\varphi(G \cup e) = G'$ yields the node numbering in the single representant G' of the supergraph $G \cup e$.

To enable the computation of these downward pointers, furthermore, we also include in the data structure upward pointers that correspond to removals of edges from graphs. For graph G and an edge $e \in G$, a permutation is stored such that $\varphi(G) - \varphi(e) = G'$, where G' is the single stored graph that is isomorphic with $G - e$. In Figure 1, for instance, we store associated to G_{51} an upward pointer to G_{43} with edge $(3,4) \in G_{51}$ and permutation 53214. Removal of edges is called upward refinement. Not all edges of a graph G can be removed. For instance, removal of $(2,3)$ from G_{31} would yield an unconnected graph. Therefore, the set of all upward refinements $\rho^-(G)$ that is stored has $\rho^-(G) \subseteq G$.

Finally, to every graph that is stored, we associate a so-called BSGS. More details about this are given later.

The main contribution of our work is a method with polynomial delay that computes all upward and downward pointers and all BSGSs, such that no two isomorphic graphs need to be stored.

We will illustrate this method using the example of Figure 1. Assume that we have already computed all pointers upto the 4th level, i.e. we have all pointers except those pointing from and to G_{51}. Then we proceed as follows.

We assume that all graphs that have not been (completely) downward refined yet are stored in a FIFO queue, and that G_{41} is the first unrefined graph in this queue, which we are going to output and refine.

Assume that $(2,5)$ is the first downward refinement that we generate for G_{41}. Then we store the resulting graph $G_{51} = G_{41} \cup (2,5)$ and a pointer from G_{41} to G_{51}, including its corresponding permutation 12345. Next, we fill in upward pointers from G_{51} to all its upward refinements. For one upward refinement, $(2,5)$, this is straightforward — we can invert the downward refinement that we just performed. For the others, it is more complicated. For each upward refinement $e \in \delta(G_{51})$ $(e \neq (2,5))$ one of these two cases applies:

- we can remove both $(2,5)$ and e from G_{51} without disconnecting G_{51}; for example, this is the case for $e = (1,2)$;
- if we remove $(2,5)$ and e we disconnect G_{51}; for example, this is the case for $e = (3,4)$.

In the first case, as illustrated in Figure 1 for $e = (1,2)$, we can reach a representative for $G_{51} - e$ by first removing $(2,5)$

[1]In thise case, the anti-monotonic constraint is that G is frequent in a database containing only graph G_{51}; $\theta = 1$.

from G_{51}, then removing the edge that corresponds to $(1,2)$ in G_{31} and finally adding the edge that corresponds to $(2,5)$ in G_{31} again — these correspondences can easily be determined using the stored permutations. By concatenating all the stored permutations, we obtain a permutation to G_{43}. In this case, we thus exploit that edges $(1,2)$ and $(2,5)$ can be removed in an arbitrary order; there is a 'diamond' in the datastructure for which we determine the last missing edge.

In the second case, we cannot remove $(3,4)$ from $G_{51} - (2,5)$; however, we can remove $(3,4)$ from $G_{51} - (1,2)$, of which we now know the representant G_{43}. By concatenating the known permutations in the diamond for $(3,4)$ and $(1,2)$, we can obtain a permutation to G_{43}:

$G_{51} \rightarrow G_{42}$:	51234
$G_{42} \rightarrow G_{31}$:	34125
$G_{31} \rightarrow G_{43}$:	12345
$G_{51} \rightarrow G_{43}$:	53412

In the general case, we can create a *graph of parents* for a graph G, in which every upward refinement of G is a node. There is an edge between nodes in the graph of parents if the two upward refinements can both be applied at the same time without disconnecting the graph G. To show that we can obtain an upward pointer for every upward refinement, we need to show that the graph of parents is connected, as this means that from the edge that is initially removed, we can build a path to every other upward refinement by repeatedly applying the diamond property. We can prove that for the search space of connected graphs, the graph of parents is connected. Obviously, this procedure is polynomial as the number of upward refinements is at most $O(n^2)$ for a graph with n nodes.

Along with every upward pointer, we can also fill in one downward pointer for every parent: the removal of $(3,4)$ from G_{51} corresponds to adding $(1,4)$ to G_{43}. If at a later point all downward refinements of G_{43} are generated, we can therefore skip $(1,4)$ as its pointer is already filled in.

Up until now, however, we have ignored a major issue, which can be seen for instance for the example graph G_{42}. There are four possible downward refinements, all of which lead to the same graph. There is only one upward refinement which leads to G_{42}. We need to make sure that all downward pointers of G_{42} point to the same child. This is where we need a *base and a strong generating set* (BSGS) of G_{42}. BSGSs are a well-known concept in group theory [2]. Essentially, a BSGS is a set of permutations which allows one to generate every possible automorphism of a graph (an automorphism is an isomorphism of a graph to itself). When we are refining G_{42}, we can use the BSGS to determine for every pair of refinements if they are equivalent or not, and if so, we can derive a permutation which transforms one into the other.

Of importance, of course, is then what the complexity is of computing a BSGS. Given an arbitrary set of permutations that generates the automorphism group, an $O(n^5)$ algorithm is known to compute the BSGS. The main question is how to obtain the initial set of permutations. The main idea is here that we can compute this set of permutations by reusing the BSGS from an upward refinement. Due to lack of space we have to skip the details of this computation, but this can also be done in $O(n^5)$ time.

To conclude, therefore, we proceed as follows when refining a graph:

- we compute the BSGS of the graph
- we create every possible downward refinement
- we determine which downward refinements are equivalent from the BSGS
- consider every downward refinement in some predefined order:
  - if the pointer for the downward refinement has not been filled in yet:
    - if the downward refinement is equivalent with another downward refinement that has already been filled in, create a pointer to the supergraph of the other refinement
    - if the downward refinement is not equivalent with another downward refinement that has already been filld in, create a new supergraph and a pointer to the supergraph; add the supergraph to the FIFO queue; for the newly created supergraphs, fill in its pointers to its parents as previously described.

As we apply this procedure in a levelwise fashion, we can be sure that after having refined one level, we have filled in all downward pointers to the new level, and that the new level has appropriate upward pointers to the old level.

3. DISCUSSION

Our algorithm for enumerating graphs has several essential properties:

- it generates the set of subgraphs in a level-wise fashion;
- after it has finished generating a level of subgraphs, it has at least 3 levels of subgraphs in memory, including all possible upward and downward pointers between the 3 levels.

As the number of subgraphs of a certain number of edges can be exponential in the number of edges, our algorithm has exponential space complexity. We argue that it depends on the application if this is a problem.

In frequent subgraph mining, where one is interested in storing all subgraphs upto a certain size anyway, the drawback of our method is relatively limited. Indeed, our method for enumerating subgraphs can be seen as a drop-in replacement for the enumerion algorithms that have been used in breadth-first subgraph miners such as FSG [6] and AGM [5] that also work in a levelwise fashion and first generate subgraphs before counting their frequency in a database. The same argument applies to methods that use the frequent patterns afterwards as boolean features, and to several other data mining methods relying on a sparse set of frequent patterns.

If one is not interested in all subgraphs that satisfy the anti-monotonic constraint — for instance, one is only interested in the number of frequent subgraphs and not the subgraphs themselves — the space complexity of our approach may be prohibitive. The graph generation algorithms that have been used in gSpan and other depth-first graph miners [14], but also the well-known graph generation algorithm Nauty [8], have as main advantage that their space complexity is polynomially bounded by the size of the largest subgraph that is generated.

An important issue when our algorithm is used in frequent subgraph mining, is that in frequent subgraph mining not only graph isomorphism problems need to be solved, but also subgraph isomorphism problems: for every graph in the database (or a non-empty subset thereof), we need to check if it contains a generated subgraph. The subgraph isomorphism problem is NP complete; both theoretically and experimentally, it has been shown that the computation of subgraph isomorphisms therefore dominates the overall runtime of most algorithms [11]. If any speed-up can be obtained by using a polynomial graph generation algorithm, it will be relatively small.

However, the datastructure that is produced by our algorithm has merits on its own. In those applications where it is of interest to store all subgraphs, it can be desirable that one can also efficiently retrieve any information, such as frequency, that is associated to a stored subgraph. Our datastructure allows such retrievals to be answered in polynomial time. This follows from the observation that every graph can be described as a sequence of downward refinements, and we can follow any sequence of downward refinements in our datastructure. No other graph miner that we know of produces a datastructure which allows such polynomial retrieval of information.

Of theoretical interest is the observation that our algorithm can be used to generate graphs in several other normal forms, among which gSpan's normal form. A property of our algorithm is that it works for any possible order in which downward refinements are generated. In our example, we choose to generate $(5, 2)$ before $(4, 1)$, for instance. It can be shown that if we generate downward refinements in decreasing order (where the order among edges is as defined in [14]), that levelwise the generated graphs have the same order as they would have when sorted according to gSpan's normal form. In Figure 1, this is illustrated: for every subgraph, the nodes are numbered exactly in the order in which gSpan's normal form would order them.

Finally, our algorithm can be used to enumerate any class of graphs which can be described using an anti-monotonic predicate and for which the predicate can be evaluated in polynomial time. For instance, there are polynomial algorithms to determine if a graph is a tree, planar or outerplanar. As any subgraph of an (outer)planar graph is (outer)planar, we can test in polynomial time for every generated graph whether it violates the (outer)planarity property, and prune it if necessary. Of course, the resulting algorithm will have worse complexity than more specialized algorithms [12, 13].

4. CONCLUSIONS

We proposed a general algorithm that can enumerate with polynomial delay a wide range of graph classes and only requires a refinement schema satisfying certain properties and an antimonotonic predicate p. In this way, it can be used as a generic candidate pattern generator for a wide range of algorithms mining structured patterns, avoiding the need to research specialized canonical forms and enumeration strategies. The algorithm has some interesting theoretical properties and improves on existing enumeration algorithms in several ways. In future work we intend to further analyse the theoretical aspects (including an investigation of the question whether this algorithm can be used to solve some open enumeration problems such as for claw-free graphs); also, we intend to perform an empirical evaluation.

5. REFERENCES

[1] C. Borgelt. borgelt On Canonical Forms for Frequent Graph Mining In *Proceedings of the 3rd International Workshop on Mining Graphs, Trees and Sequences* (MGTS), pages 1–12, 2005.

[2] G. Butler. *Fundamental algorithms for permutation groups*, volume 559 of *LNCS*, 1991.

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

[4] L.A. Goldberg. Polynomial space polynomial delay algorithms for listing families of graphs. In *Proceedings of ACM symposium on Theory of computing*, pages 218–225, 1993.

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

[6] M. Kuramochi and G. Karypis. An Efficient Algorithm for Discovering Frequent Subgraphs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1038–1051, 2004.

[7] J. Leskovec, A. Singh, and J. Kleinberg. Patterns of Influence in a Recommendation Network. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (PAKDD), 2006.

[8] B.D. McKay. Isomorph-free exhaustive generation. *Journal of Algorithms*, 26:306–324, 1998.

[9] S. Nakano and T. Uno. Constant time generation of trees with specified diameter. In *Proceedings of the 30th International Workshop on Graph Theoretical Concepts in Computer Science*, volume 3353 of *LNCS*, pages 33–45, 2004.

[10] S.-H. Nienhuys-Cheng and R. De Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *LNCS*, 1997.

[11] S. Nijssen, J.N. Kok Frequent Subgraph Miners: Runtimes Don't Say Everything. In *Proceedings of the Workshop on Mining and Learning with Graphs* (MLG), 2006.

[12] C. Paul, A. Proskurowski, and J.A. Telle. Generating graphs of bounded branchwidth. In *Proceedings of the 32nd International Workshop on Graph Theoretical Concepts in Computer Science*, volume 4271 of *LNCS*, pages 206–216, 2006.

[13] R.A. Wright, B. Richmond, A. Odlyzko, and B.D. McKay. Constant time generation of free trees. *SIAM Journal on Computing*, 15(2):540–548, 1986.

[14] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM 2002)*, Japan, 2002. IEEE Computer Society.