Research Article

Jurij Mihelič* and Uroš Čibej

An experimental evaluation of refinement techniques for the subgraph isomorphism backtracking algorithms

https://doi.org/10.1515/comp-2020-0149
Received Feb 27, 2020; accepted Apr 19, 2020

Abstract: In this paper, we study a well-known computationally hard problem, called the subgraph isomorphism problem where the goal is for a given pattern and target graphs to determine whether the pattern is a subgraph of the target graph. Numerous algorithms for solving the problem exist in the literature and most of them are based on the backtracking approach. Since straightforward backtracking is usually slow, many algorithmic refinement techniques are used in practical algorithms. The main goal of this paper is to study such refinement techniques and to determine their ability to speed up backtracking algorithms. To do this we use a methodology of experimental algorithmics. We perform an experimental evaluation of the techniques and their combinations and, hence, demonstrate their usefulness in practice.

Keywords: graph, subgraph isomorphism, backtracking, algorithm, experimental algorithmics

1 Introduction

Data structured in the form of graphs appear in many diverse disciplines such as chemistry, biology, social networks, and document analysis. A graph is a fundamental data structure that offers its users the ability to represent structure often present in practical problems. To do this, a graph interface supports the representation of objects, called *vertices*, and relations, called *edges* among them. As the amount of data is increasing, its analysis requires better and better algorithms both in terms of performance and quality of solutions obtained.

In this paper, we focus on a well-known $\mathcal{NP}$-hard combinatorial problem from graph theory, called the *subgraph isomorphism problem*, where the goal is for two graphs to determine whether the first one is a subgraph of the second one.

The problem has numerous applications [1–4] and a plethora of algorithms for solving the problem exactly exists in the literature. Most of thse algorithms are based on the *backtracking* approach which explores the search tree of all possible solutions. Since the problem is $\mathcal{NP}$-hard, all such algorithms exhibit exponential running time in the worst case. Nevertheless, in many practical cases, the problem can be very efficiently solved if advanced techniques of pruning the search tree and other optimizations are employed.

Many such refinement techniques exist in the literature where each algorithm is employing a subset of techniques. In this paper, we do not present another algorithm for solving the subgraph isomorphism problem, but rather focus on refinement techniques intended for backtracking algorithms which may provide a significant performance boost to algorithms especially if used in a combination with each other.

Most of the techniques are based on heuristic principles in order to improve the average-case complexity of the algorithms. Nevertheless, several use optimization and tuning of algorithms. The general area on which this paper is focused is called *algorithm engineering* and strives to bridge the gap between theory and practice by using theoretical results to improve the practical performance of algorithms [5, 6]. To establish the practical performance, approaches from *experimental algorithmics* are used. Most of the techniques presented in this paper originate from the field of constraint satisfaction [7].

The most widely used and well-known practical subgraph isomorphism algorithms are Ullmann's algorithm [8, 9], VF variants [10–12], RI [13], Glasgow subgraph solver [14], FocusSearch [15], and LAD [16]. The techniques pre-

***Corresponding Author: Jurij Mihelič:** Faculty of Computer and Information Science, University of Ljubljana, Večna pot 113, Ljubljana, 1000, Slovenia; Email: jurij.mihelic@fri.uni-lj.si
Uroš Čibej: Faculty of Computer and Information Science, University of Ljubljana, Večna pot 113, Ljubljana, 1000, Slovenia; Email: uros.cibej@fri.uni-lj.si

sented in this paper mostly originate from the study of these algorithms. Our main result is the experimental comparison and evaluation of the presented techniques. Nevertheless, many similar but different approaches to solving the problem, as well as the subgraph isomorphism related problems, are studied in the literature [17–20].

In the rest of the paper, we first present the methodology used to perform our research. We describe the experimental setup and principles used in our experimental study. Afterward, in Section 3 we describe the basic notion used in the rest of the paper. We define graphs, morphisms and the problem studied as well as the algorithmic framework for solving the problem using the backtracking approach. Section 4 is the main section of the paper and describes refinement techniques. We give a description of each technique as well as present the results of its experimental evaluation. Finally, in Section 5 we conclude the paper.

2 Methodology

In this section we describe the research methodology that we used to conduct the experimental evaluation of refinement techniques for backtracking algorithms used in solving the subgraph isomorphism problem.

2.1 Experimental Algorithmics

The discipline called *experimental algorithmics* focuses on scientific experiments with algorithms as well as the theory behind them [5, 21]. It studies algorithms as laboratory subjects via various experimental techniques such as control of parameters and isolation of components. It joins the best from both theoretical and empirical fields, and also tries to overcome their main weaknesses by using realistic models of computations as well as careful planning and execution of the experiments.

To conduct the experiments we followed the process [5, 6] depicted in Figure 1. This process consists of two phases: *planning* and *execution* of an experiment. The first phase begins with the formulation of the goals of the experiment, and is followed by the definition of measures and factors of influence, preparation of tests and setting up the experimental tools. Afterward, the experiment is executed and the obtained results are analyzed. Finally, the results, if beneficial, are reported.

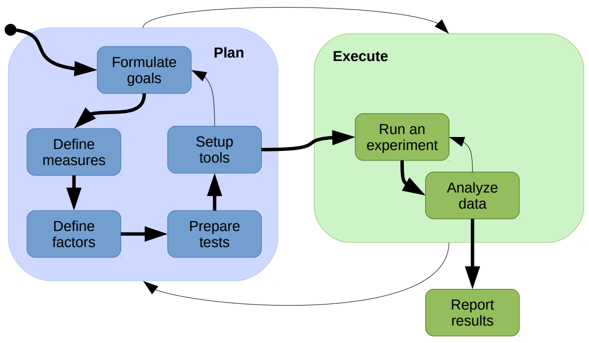


Figure 1: Experimental process.

2.2 Experimental Setup

Our main reason to conduct experiments with refinement techniques for backtracking algorithms for solving the subgraph isomorphism problem is to obtain insights into which techniques or their combination have an impact on practical algorithm efficiency, as well as to measure this impact.

Our motivation is twofold:

- to provide experimental background for using a particular refinement technique,
- to determine combinations of refinements that perform particularly well together.

To achieve these objectives we systematically performed experiments with two straightforward implementations of backtracking, namely forward and backward checking. Following the principle of component isolation, we separately introduced various refinements into these two implementations. Similar experimental evaluations already exist in the literature [22].

After the implementations were finished we performed experiments with a well-know database of test instances for the subgraph isomorphism [23]. Our selected test scenarios consisted of random graphs and bounded valence graphs available in the database. In particular, random graphs include Erdős -Rényi graphs with edge probability from 0.01 to 0.1, and bounded valence graphs where the maximum degree of vertices in graph is bounded. The size of the pattern graph is 20%, 40% or 60% of the size of the target graph.

All algorithms in the particular experiment were executed on the same test inputs and the same computer architecture. To do this, we developed a dedicated test program that reads the input, *i.e.* the pattern and target graph, performs the algorithm and reports the results. The program was written in the C++ programming language and compiled with GCC 8.2.0 with flags `-O3` and `-march=native`. The experiments were performed with quad-core Intel Core i7-4771 computer running at 3.5 GHz and having 16 GB of memory (8 MB L3 cache). The operating system used was Arch Linux (kernel 4.18.1). The main indicators of an algorithm performance that we used were the number of subgraph isomorphisms between pattern and target graph, as well as the processor time to count them all. If the algorithm was not able to find all subgraph isomorphisms in the specified time limit we terminated its execution.

To graphically report the result we use line plots where the x-axis represents time and the y-axis gives the (cumulative) number of instances the algorithm can solve in that amount of time.

2.3 Experimental Principles

When planning the experiments we followed well-defined experimental principles [5]. In what follows we enlist these principles and discuss their particularities regarding our experimental evaluation.

Reproducibility *Retaking the same experiment should produce similar results.* In our experiments, we used a public database of test cases and our implementations of algorithms are publicly available[1]. The experimental setting is well described in this paper.

Correctness *Indicators obtained from the experiment must accurately reflect the properties being studied.* Our experiments were performed on the dedicated computer running only test programs and basic operating systems. We also took care to produce correct implementations and to test for the equality of solutions obtained by different algorithms.

Validity *Conclusions drawn from the experimental results are based on the correct interpretations of data.* Our experimental test set is diverse and large enough to exclude the possibility of spurious or pathological results. We also applied the principle of isolation of components; particularly, we separately tested each refinement based on the basic forward and backward backtracking.

Generality *Analysis of the results and conclusions should apply broadly.* The measures and indicators used in our experiments are standard, *i.e.* execution time. Moreover, the results are comparable to other similar experiments.

Efficiency *Produce correct results without wasting time and other resources.* Our experiments are performed through scripting; the process is mostly automatic.

Newsworthiness *Produce interesting results and conclusions.* The subgraph isomorphism problem is well-studied. Hence, to make progress towards faster algorithms, a careful study of various approaches and techniques is a very interesting research question that may lead to better algorithms in the future.

1 See online repository at https://git.sr.ht/~xnevs/graph-monomorphism-experiment

3 Basic Notions

In this section, we give the basic notions used in this paper. In particular, we define graph notions, various morphisms on them, and formally define the subgraph isomorphism problem. Moreover, we present a general framework for solving the problem using the algorithms based on the backtracking approach.

3.1 Graphs

First, let us present definitions of several notions from graph theory. A *graph* is a pair $G = (V_G, E_G)$, where $V_G = \{1, 2, \ldots, n_G\}$ is a finite set of *vertices* and $E_G \subseteq V_G \times V_G$ is a set of vertex pairs representing *edges*. If edges are unordered or ordered then the graph is undirected or directed, respectively. In this paper, our discussion is mostly focused on undirected graphs, but approaches and techniques may as well be used on the directed ones.

For an undirected graph $G = (V_G, E_G)$ and a vertex $u \in V_G$ we define the *neighborhood* of u as

$$\mathcal{N}_G(u) = \{v \in V_G \mid \langle u, v\rangle \in E_G\},$$

and its *degree* as

$$d_G(u) = |\mathcal{N}_G(u)|.$$

Consider an undirected or directed graph $H = (V_H, E_H)$. A graph $G = (V_G, E_G)$ is a *subgraph* of a graph H if

$$V_G \subseteq V_H \quad \text{and} \quad E_G \subseteq E_H.$$

The graph G is called an *induced subgraph* of the graph H if

$$\forall u, v \in V_G : \langle u, v\rangle \in E_G \iff \langle u, v\rangle \in E_H.$$

3.2 Morphisms

Let $G = (V_G, E_G)$ and $H = (V_H, E_H)$ be two graphs. A bijective mapping $\phi : V_G \to V_H$ is called a *graph isomorphism* if

$$\forall u, v \in V_G : \langle u, v\rangle \in E_G \iff \langle \phi(u), \phi(v)\rangle \in E_H.$$

An injective mapping $\phi : V_G \to V_H$ is called *graph monomorphism* if

$$\forall u, v \in V_G : \langle u, v\rangle \in E_G \implies \langle \phi(u), \phi(v)\rangle \in E_H.$$

Instead of the term graph monomorphism the term *(ordinary) subgraph isomorphism* is often used since the mapping ϕ is an isomorphism between G and the subgraph $\phi(G)$ of the graph H. Here, $\phi(G)$ represents a subgraph of H to which G is mapped, *i.e.* the vertices V_G of G are mapped using $\phi(v)$ for each $v \in V_G$.

Thus, a graph monomorphism is an *injective mapping* preserving adjacency relation where the images of vertices of the graph G can have some additional edges in H not present in G. A more restricted version of the notion is called an *induced subgraph isomorphism* and is defined as an injective mapping $\phi : V_G \to V_H$ where

$$\forall u, v \in V_G : \langle u, v\rangle \in E_G \iff \langle \phi(u), \phi(v)\rangle \in E_H.$$

Consequently, the corresponding edges must be present in both graphs G and H. Consider a simple example in Figure 2 demonstrating the difference between the ordinary and induced version of the subgraph isomorphism.

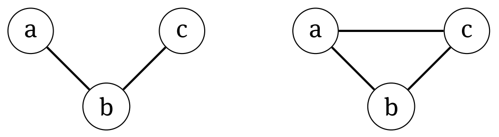


Figure 2: The left graph G has no induced subgraph isomorphisms in the right graph H while it has six ordinary subgraph isomorphisms. Here, in the latter, the vertices abc of G may map to abc, cba, bca, acb, bac, and cab of H.

3.3 Problem Definitions

Now, let us describe several versions of the subgraph isomorphism problem. The input of the problem is two graphs G and G: here, the graph G is usually called a *pattern* and the graph H a *target*.

Several versions of the subgraph isomorphism problem exist:

Decision problem Given graphs G and H, is there a subgraph isomorphism between the two?

Enumeration problem Given graphs G and H, list all the subgraph isomorphisms between the two.

Counting problem Given graphs G and H, count the number of subgraph isomorphisms between the two.

All these problems can ask for ordinary as well as induced subgraph isomorphisms. Notice that, in practice, other constraints may also be introduced into the problem, such as matching of labels of vertices or edges.

To show the $\mathcal{NP}$-completeness of the decision version of the problem, a reduction from the well-known clique or Hamiltonian cycle problems is usually employed. See also [24] for an overview of the problem.

3.4 Backtracking Algorithms

Many of the algorithms for solving the subgraph isomorphism problem are based on the backtracking approach. In particular, the mapping $\phi : V_G \to V_H$ is constructed gradually where at each step one assignment of a vertex $u \in V_G$ is resolved, *i.e.* $\phi(u)$ is assigned to some $u' \in V_H$. Additionally, after each step, problem constraints may be checked to determine the feasibility of the solution. If constraints are satisfied the algorithm proceeds to the next not yet processed vertex of V_G, otherwise, it tries the next candidate vertex of V_H for the assignment. When there are no more candidates left, the algorithm *backtracks*, *i.e.* it undoes the changes and steps back to the previous step. Without loss of generality, we assume (if not otherwise noted) that the vertices of V_G are processed in the order of their labels, *i.e.* $1, 2, \ldots, n_G$.

A state where some vertices of V_G are already assigned is called a *partial solution*. In particular, at step k, where $1 \le k \le n_G$, the algorithm resolves the vertex $k \in V_G$ and constructs a partial solution ϕ_k from ϕ_{k-1}. Here, $\phi_0(u) = \perp$ (*i.e.* undefined) and $\phi_k(u) = \phi_{k-1}(u)$ for all $u < k$. Finally, $\phi(u) = \phi_{n_G}(u)$.

A *consistency* of a partial solution ϕ_k may be checked by satisfying the following conditions:

(injectivity) $\phi(u) \neq \phi(v)$ for each $u, v \in V_G : u \neq v$,
(adjacency) $\langle \phi(u), \phi(v)\rangle \in E_H$ for all $\langle u, v\rangle \in E_G$, and
(non-adjacency) $\langle \phi(u), \phi(v)\rangle \notin E_H$ for all $\langle u, v\rangle \notin E_G$.

The listed conditions are basic constraints that determine the feasibility of the solution. In order to more efficiently prune the search tree, other constraints may be derived. The last constraint is only relevant to the induced version of the problem.

Usually, the straightforward backtracking approach to solving the subgraph isomorphism problem results in a recursive algorithm.

There are two general approaches to how to implement candidate checking in backtracking algorithms:

Backward checking. When a new candidate vertex $u' \in V_H$ is mapped to some vertex $u \in V_G$ the consistency of the partial solution is checked: if the check fails, then the algorithm backtracks, otherwise, it proceeds to the next vertex of G. When the last vertex is successfully mapped the algorithm finds a subgraph isomorphism.

Forward checking. Whenever a vertex $u \in V_G$ is mapped to $u' \in V_H$ we use this to form additional constraints on candidates for vertices not yet mapped. For example, due to injectivity, no vertex in V_G can be mapped to u'. Such additional constraints may be stored in a binary matrix where the u-th row gives the u's candidates.

All of the algorithms listed in the introduction utilize one of these two approaches and the refinement techniques described in the rest of the paper are generally applicable to both approaches unless otherwise noted.

4 Refinement Techniques

Here, we describe several refinement techniques applicable to backtracking algorithms for the subgraph isomorphism problem. We focus on the techniques which can provide practical speedups of the algorithms. To demonstrate this, we present the results of the experimental evaluation of the techniques.

4.1 Search Order

One of the most straightforward techniques for speeding up backtracking algorithms is the exploitation of the order in which the vertices are processed. In this section, we briefly explore this option while we refer the reader to [25] for a more in-depth analysis.

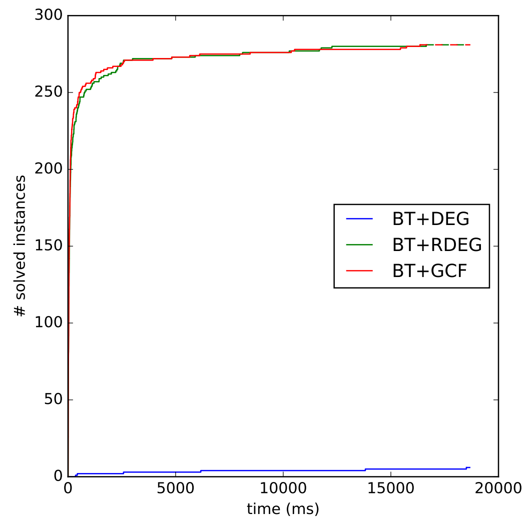
The backtracking algorithm correctness does not depend on the ordering, but the shape of a search tree does; smaller trees may result in much more efficient algorithms. In particular, the objective is to use such an order that leads to inconsistency as soon as possible in order to prune larger parts of the tree.

Here, we present several heuristics for ordering the vertices. Let $(u_1, u_2, \dots, u_{k-1})$, where $u_i \in V_G$, be a partial ordering of vertices; we also denote $U_{k-1} = \{u_1, u_2, \dots, u_{k-1}\}$ for the set in this order. In the following list we present several options for selecting the next vertex $u_k \in V_G \setminus U_{k-1}$.

DEG (degree) Vertices are non-increasingly ordered by degree, *i.e.*

$$u_k = \operatorname*{arg\,max}_{u \in V_G \setminus U_{k-1}} d_G(u).$$

RDEG (relative degree) Start with a vertex with the highest degree. Afterward, proceed with vertices which are adjacent to the maximal number of already selected ver-



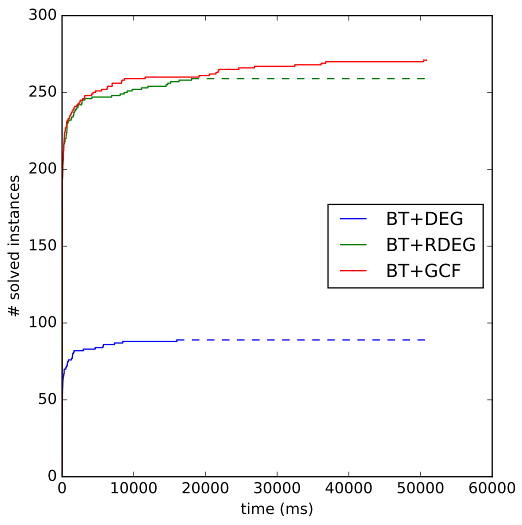


Figure 3: Results of the experiments with the backward-checking backtracking algorithm and various vertex orders on random graphs (left) and bounded valence graphs (right): x-axis specifies time-allowed for solving and y-axis gives the number of solved instances in the given time.

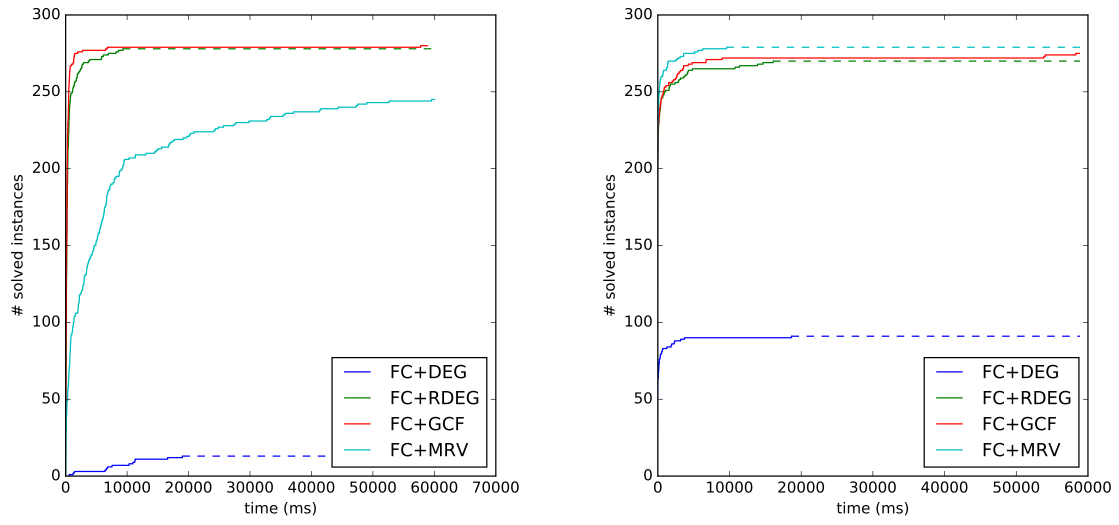


Figure 4: Results of the experiments with the forward-checking backtracking algorithm and various vertex orders on random graphs (left) and bounded valence graphs (right): x-axis specifies time-allowed for solving and y-axis gives the number of solved instances in the given time.

tices, *i.e.*

$$u_k = \underset{u \in V_G \setminus U_{k-1}}{\arg\max} |\mathcal{N}_G(u) \cap U_{k-1}|.$$

If several vertices match in the above equation, select one with a higher degree.

GCF (greatest constraint first) This order (see also [13]) is similar to RDEG but equalities are resolved differently. Let M be the set of vertices with at least one neighbor in the partial order, *i.e.*

$$M = \{u \in V_G \setminus U_{k-1} \mid \mathcal{N}_G(u) \cap U_{k-1} \neq \emptyset\}$$

and the next selected vertex

$$u_k = \underset{u \in V_G \setminus U_{k-1}}{\arg\max} (|\mathcal{N}_G(u) \cap U_{k-1}|, |\mathcal{N}_G(u) \cap M|, d_G(u)).$$

MRV (minimal remaining values) In the forward checking algorithms we can, for each vertex $u \in V_G$, store its candidate vertices from V_H. Using this information we can, in each state of the search tree, select such a vertex that has a minimum number of candidates. The goal is to minimize the branching of the search tree.

To compare the performance of the above orders we performed an experiment. See Figure 3 for the comparison of orders for backward-checking algorithms and Figure 4 for the comparison of orders for forward-checking algorithms. Observe that DEG order (which represents a good starting point) performs inferior to the other orders. Using more advanced orders significantly improves the performance of both backward and forward-checking algorithms.

4.2 Degree Constraints

Another approach to pruning the search tree is to derive additional problem constraints. To do this we use basic (sufficient) constraints given by the problem definition. The goal is that the derived constraints can be efficiently checked and if they are not satisfied we can prune the tree.

First, let us derive the *vertex degree* constraints. Since the mapping of vertices is an injection, we can see that

$$d_G(u) \leq d_H(\phi(u)) \quad \text{for each } u \in V_G.$$

Indeed, the neighborhood of $u \in V_G$ must be mapped to the neighborhood of $\phi(u)$. Thus, $\phi(\mathcal{N}_G(u)) \subseteq \mathcal{N}_H(\phi(u))$ and $|\mathcal{N}_G(u)| \leq |\mathcal{N}_H(\phi(u))|$.

Explicitly storing vertex degrees in the graph data structure enables us to very efficiently check degree constraints. Notice that, in backward checking the constraint is easily checked for every new mapping made while in forward checking the constraint is included in the preprocessing phase.

Degree checking can also be extended to a neighborhood of a vertex. First, we define a *neigborhood degree sequence* $Z_G(u)$ for a vertex $u \in V_G$ as

$$Z_G(u) = (d_G(v))_{v \in \mathcal{N}_G(u)} \quad \text{sorted nonincreasingly.}$$

Observe that, for each vertex $u \in V_G$ the following must hold

$$Z_G(u) \preceq Z_H(\phi(u)) \tag{1}$$

where $\preceq$ is the relation of lexicographical comparison of two sequences.

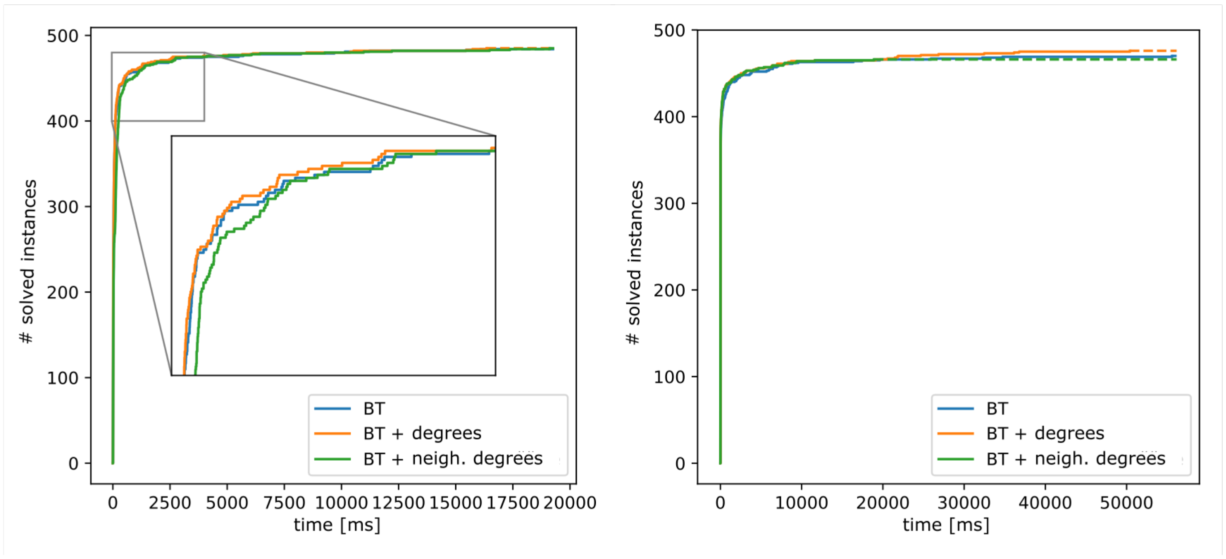


Figure 5: Results of the experiments for the algorithms using degree constraints on random graphs (left) and bounded valence graphs (right): Solved test instances – degree constraints: x-axis specifies time-allowed for solving and y-axis gives the number of solved instances in the given time.

The constraint is a consequence of the vertex degree constraint. In particular, we can see that for each $v \in \mathcal{N}_G(u)$ there must be a unique vertex $v' \in \mathcal{N}_H(\phi(u))$ such that $d_G(v) \leq d_H(v')$. Furthermore, checking this for the whole neighborhood corresponds to Equation (1).

To compare the performance of both refinements we performed an experiment with the backtracking algorithm. See Figure 5 for the results. Observe that using degree constraints slightly improves the performance of the algorithms while neighborhood degree constraints seem to be too much of an overhead to improve the total performance.

4.3 Bit arrays

In the subgraph-isomorphism backtracking algorithms, set operations such as union and intersection are often used, where sets are subsets of vertices of graph G (or H). One way to efficiently represent such sets is to use *bit ar-*

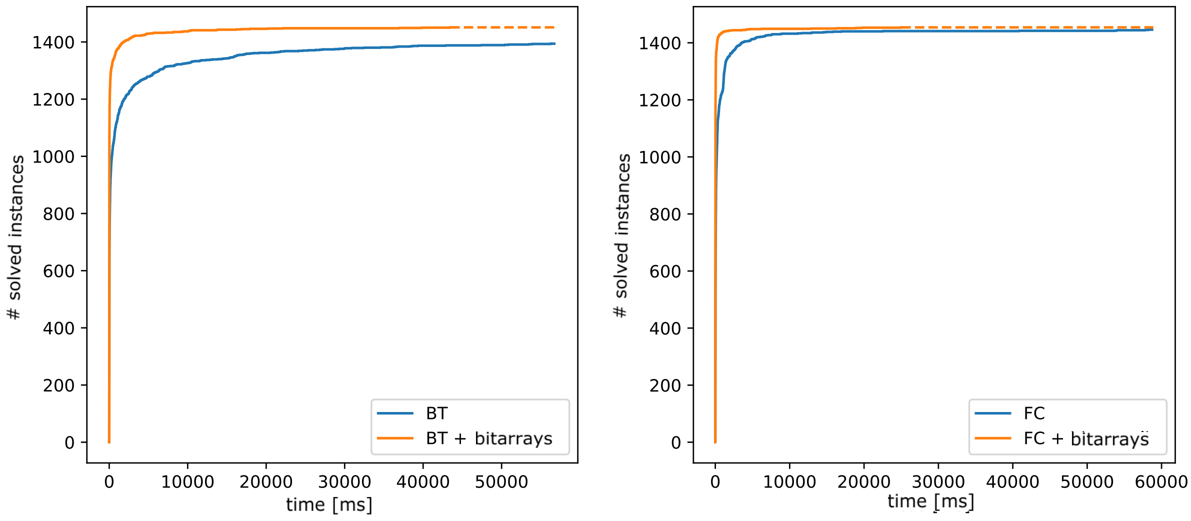


Figure 6: Results of the experiments on random graphs for the algorithms using bitarrays and backward-checking (left) and forward-checking (right): x-axis specifies time-allowed for solving and y-axis gives the number of solved instances in the given time.

rays, where the i-th bit of the array corresponds to the membership of the vertex i (we assume vertices are numbered from 1 to n). Now using a bit array representation, union correspond to bitwise or, intersection to bitwise and and difference to and and not.

A graph can be represented with bit arrays, where for each vertex a bit array is stored representing a set of the vertex's neighbors. To compute the set of candidates for the vertex $u \in V_G$ we consider only the vertices from the intersection of neighborhoods of already mapped neighbors of u, *i.e.*

$$\bigcap_{v \in \mathcal{N}_G(u), v<u} \mathcal{N}_H(\phi(v)).$$

To see the practical effect of using bit arrays see Figure 6. We observe that in both forward and backward checking algorithms the use of bit arrays has a positive effect on the running time.

4.4 Parents-based Candidate Selection

In this section, we describe a tuning technique that does not prune the search tree but it enables a faster selection of candidate vertices. Let $(u_1, u_2, \dots, u_{k-1})$, where $u_i \in V_G$, be a partial ordering of vertices and let $u_k \in V_G$ be the next vertex to be assigned a vertex $u'_k \in V_H$. Moreover, let u_k be adjacent to some vertex u_p appearing before u_k in the search order, *i.e.* $1 \leq p < k$. We call such a vertex u_p, a *parent* of u_k.

Since a vertex $u'_k \in V_H$ to be assigned to $u_k \in V_G$ must also be adjacent to a vertex $u'_p \in V_H$ that is assigned to $u_p \in V_G$, we may, when selecting u_k's candiate vertices, iterate only on the neighbors of u'_p, *i.e.* $u'_k \in \mathcal{N}_H(u'_p)$.

Notice that, u_k might not have a parent, so the algorithm must still check all the vertices. However, in connected graphs the parent almost always exists; in particular, this is true for the RDEG ordering which orders the vertices according to the number of parents. The results of the experimental evaluation are in Figure 8 and are described in the next subsection.

4.5 Neighborhood Adjacency Checking

Let $u \in V_G$ and $u' \in V_H$ be the two vertices so that u is mapped to u'. We present another approach to consistency-constraint checking which probes the vertices in the neighborhoods of u and v. In particular, we check all the vertices $v \in \mathcal{N}_G(u)$ that are already mapped; we denote with $v' \in$

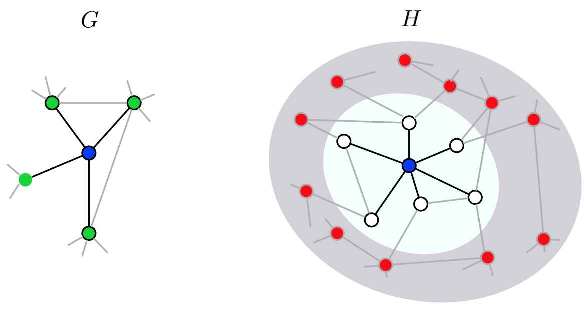


Figure 7: Filtering based on the selected pattern and target vertices (blue). The neighbors (green) of the pattern vertex are incompatible with the non-neighbors (red) of the target vertex (color online).

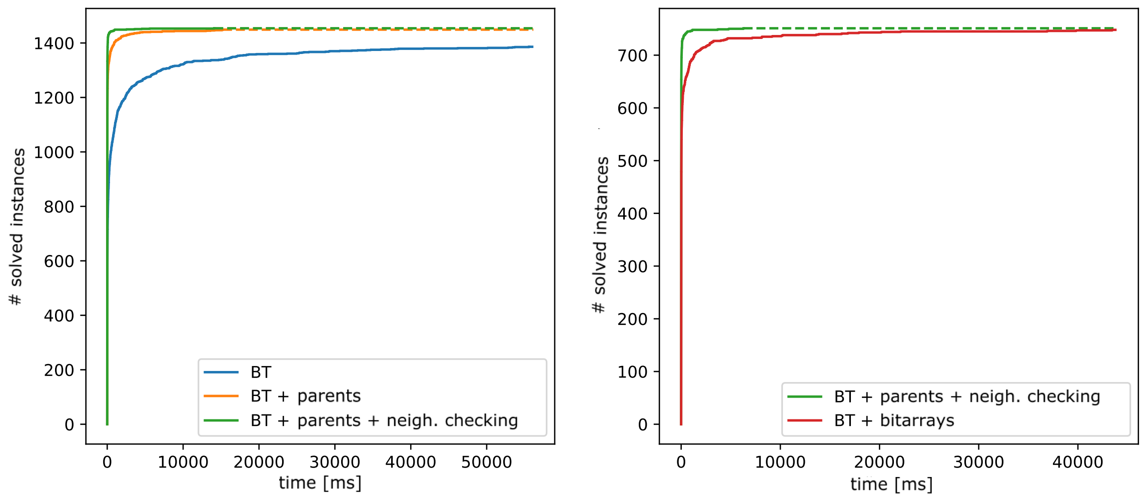


Figure 8: Results of the experiments on random graphs for the algorithms using various refinements and their combinations: x-axis specifies time-allowed for solving and y-axis gives the number of solved instances in the given time.

V_H a vertex in the target graph to which v is mapped. Notice that, a vertex v' (corresponding to v) must also be adjacent to u' (corresponding to u). See Figure 7 for a graphical representation of neighborhood adjacency checking. If this neighborhood adjacency check fails then the current mapping cannot represent a valid subgraph isomorphism.

Notice that, in the case of the induced subgraph isomorphism problem, the same constraint can also be checked from the perspective of the target vertex to pattern vertex mapping. To efficiently implement this technique we need to represent graphs with adjacency lists (for fast iteration of neighborhoods) as well as adjacency matrix (for fast checking of adjacency of two vertices). Moreover, we also have to store the inverse of the mapping function to efficiently obtain a vertex in the pattern which was mapped to a particular target vertex. See also [8] for details.

In Figure 8 we show the results of our experimental evaluation. Here, we compare the improvements of using parents as well as neighborhood adjacency checking and bit arrays. One can observe that the combination of using parents together with neighborhood checking performs the best. We can also observe that bit arrays (which are unfortunately incompatible with neighborhood checking) are slower than the parent-neighborhood checking combination.

4.6 Other Refinements

Of course, the survey and experimental evaluation of the refinements presented in this paper are not complete.

Many potentially interesting refinements exist, for example, pruning of the search tree based on the symmetries one can find in the pattern or target graph. Here, a promising pruning technique is based on the *exploratory equivalence* which is defined on the graph vertex set and exploits automorphisms in the pattern graph. For example, if the pattern graph is a clique graph of size q then the speed might be up to a factor of $q!$. See our previous papers [26–28] for more details on this topic.

Another interesting refinement is a reduction of memory consumption in forward-checking backtracking algorithms. Notice that such algorithms must store one bit of information representing whether a particular source vertex can be mapped to a particular target vertex. See [8] for details.

5 Conclusion

The focus of this paper was on the refinement techniques for backtracking algorithms for the subgraph isomorphism problem. Our main goal was to present the techniques, and, mainly, to experimentally evaluate them. Nevertheless, the refinements selected for presentation in this paper do not represent the complete spectrum. Our further research will focus on several not yet completely explored algorithms. To name just a few from the constraint satisfaction area there is the so-called *arc consistency* and *Hall sets* that seem promising as well as the ones mentioned at the end of the previous section.

Our experimental evaluation was performed through several separate experiments. The main reason for this is the multitude of refinements appearing in the literature. Consequently, our paper also serves as a partial survey on the topic.

Acknowledgement: Our gratitude goes to Sven Cerk who performed most of the experimental work several of the refinements. This work was partially supported by the Slovenian Research Agency and the projects "P2-0095 Parallel and distributed systems" and "N2-0053 Graph Optimisation and Big Data".

References

[1] Dimitris K Agrafiotis, Victor S Lobanov, Maxim Shemanarev, Dmitrii N Rassokhin, Sergei Izrailev, Edward P Jaeger, Simson Alex, and Michael Farnum. Efficient Substructure Searching of Large Chemical Libraries: The ABCD Chemical Cartridge. *J. Chem. Inf. Model.*, 2011.

[2] John M Barnard. Substructure searching methods: Old and new. *J. Chemical Information and Computer Sciences*, 33(4):532–538, 1993.

[3] Wenfei Fan. Graph pattern matching revised for social network analysis. In *Proc. 15th International Conference on Database Theory - ICDT '12*, page 8. ACM Press, March 2012.

[4] Jianzhuang Liu and Yong Tsui Lee. Graph-based method for face identification from a single 2D line drawing. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 23(10):1106–1119, 2001.

[5] Catherine C. McGeoch. *A guide to experimental algorithmics*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.

[6] Matthias Muller-Hannemann and Stefan Schirra, editors. *Algorithm Engineering: Bridging the Gap Between Algorithm Theory and Practice*. Springer-Verlag, Berlin, Heidelberg, 2010.

[7] Stéphane Zampelli. *A Constraint Programming Approach to Subgraph Isomorphism*. PhD thesis, Université catholique de Louvain, Belgium.

[8] Uroš Čibej and Jurij Mihelič. Improvements to Ullmann's algorithm for the subgraph isomorphism problem. *Interna-*

tional Journal of Pattern Recognition and Artificial Intelligence, 29(07):1550025, 2015.

[9] Julian R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23(1):31–42, January 1976.

[10] Vincenzo Carletti, Pasquale Foggia, Antonio Greco, Mario Vento, and Vincenzo Vigilante. Vf3-light: A lightweight subgraph isomorphism algorithm and its experimental evaluation. *Pattern Recognition Letters*, 125:591 – 596, 2019.

[11] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. Introducing vf3: A new algorithm for subgraph isomorphism. In Pasquale Foggia, Cheng-Lin Liu, and Mario Vento, editors, *Graph-Based Representations in Pattern Recognition*, pages 128–139, Cham, 2017. Springer International Publishing.

[12] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, Oct 2004.

[13] Vincenzo Bonnici, Rosalba Giugno, Alfredo Pulvirenti, Dennis Shasha, and Alfredo Ferro. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics*, 14(7), 2013.

[14] Ciaran McCreesh and Patrick Prosser. A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In Gilles Pesant, editor, *Principles and Practice of Constraint Programming*, pages 295–312, Cham, 2015. Springer International Publishing.

[15] Julian R. Ullmann. Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *Journal of Experimental Algorithmics*, 15:1.6:1.1–1.6:1.64, February 2011.

[16] Christine Solnon. AllDifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12-13):850–864, August 2010.

[17] Junhu Wang, Xuguang Rena, Shikha Anirbana, and Xin-Wen Wub. Correct filtering for subgraph isomorphism search in compressed vertex-labeled graphs. *Information Sciences*, 482:363–373, 2019.

[18] S. Sun and Q. Luo. Subgraph matching with effective matching order and indexing. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[19] Vincenzo Carletti, Pasquale Foggia, Pierluigi Ritrovato, Mario Vento, and Vincenzo Vigilante. A parallel algorithm for subgraph isomorphism. In Donatello Conte, Jean-Yves Ramel, and Pasquale Foggia, editors, *Graph-Based Representations in Pattern Recognition*, pages 141–151, Cham, 2019. Springer International Publishing.

[20] Shun Imai and Akihiro Inokuchi. Efficient supergraph search using graph coding. *IEICE Transactions on Information and Systems*, E103.D:130–141, 2020.

[21] Bernard M. E. Moret. Towards a discipline of experimental algorithmics. *Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges*, 59:197–213, 2002.

[22] Christine Solnon. Experimental evaluation of subgraph isomorphism solvers. In Donatello Conte, Jean-Yves Ramel, and Pasquale Foggia, editors, *Graph-Based Representations in Pattern Recognition*, pages 1–13, Cham, 2019. Springer International Publishing.

[23] Massimo De Santo, Pasquale Foggia, Carlo Sansone, and Mario Vento. A large database of graphs and its use for benchmarking graph isomorphism algorithms. *Pattern Recogn. Lett.*, 24(8):1067–1079, May 2003.

[24] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *International Journal of Pattern Recognition and Artificial Intelligence*, 2004.

[25] Uroš Čibej and Jurij Mihelič. Search strategies for subgraph isomorphism algorithms. In Prosenjit Gupta and Christos Zaroliagis, editors, *Applied Algorithms*, pages 77–88, Cham, 2014. Springer International Publishing.

[26] Luka Fürst, Uroš Čibej, and Jurij Mihelič. Maximum exploratory equivalence in trees. In *2015 Federated Conference on Computer Science and Information Systems, FedCSIS 2015, Łódź, Poland, September 13-16, 2015*, pages 507–518, 2015.

[27] Jurij Mihelič, Luka Fürst, and Uroš Čibej. Exploratory equivalence in graphs: Definition and algorithms. In *2014 Federated Conference on Computer Science and Information Systems, Fedcsis 2014, Warsaw, Poland, September 7-10, 2014*, pages 447–456, 2014.

[28] Uroš Čibej, Luka Fürst, and Jurij Mihelič. A symmetry-breaking node equivalence for pruning the search space in backtracking algorithms. *Symmetry*, 11(10), 2019.