Efficient Implementation of Color Coding Algorithm for Subgraph Isomorphism Problem

Josef Malík(✉), Ondřej Suchý, and Tomáš Valla

Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, Prague, Czech Republic
{josef.malik,ondrej.suchy,tomas.valla}@fit.cvut.cz

Abstract. We consider the subgraph isomorphism problem where, given two graphs G (source graph) and F (pattern graph), one is to decide whether there is a (not necessarily induced) subgraph of G isomorphic to F. While many practical heuristic algorithms have been developed for the problem, as pointed out by McCreesh et al. [JAIR 2018], for each of them there are rather small instances which they cannot cope. Therefore, developing an alternative approach that could possibly cope with these hard instances would be of interest.

A seminal paper by Alon, Yuster and Zwick [J. ACM 1995] introduced the color coding approach to solve the problem, where the main part is a dynamic programming over color subsets and partial mappings. As with many exponential-time dynamic programming algorithms, the memory requirements constitute the main limiting factor for its usage. Because these requirements grow exponentially with the treewidth of the pattern graph, all existing implementations based on the color coding principle restrict themselves to specific pattern graphs, e.g., paths or trees. In contrast, we provide an efficient implementation of the algorithm significantly reducing its memory requirements so that it can be used for pattern graphs of larger treewidth. Moreover, our implementation not only decides the existence of an isomorphic subgraph, but it also enumerates all such subgraphs (or given number of them).

We provide an extensive experimental comparison of our implementation to other available solvers for the problem.

Keywords: Subgraph isomorphism · Subgraph enumeration · Color coding · Tree decomposition · Treewidth

1 Introduction

Many real-world domains incorporate large and complex networks of interconnected units. Examples include social networks, the Internet, or biological and

J. Malík—Supported by grant 17-20065S of the Czech Science Foundation.
O. Suchý and T. Valla—The author acknowledges the support of the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics".

I. Kotsireas et al. (Eds.): SEA^2 2019, LNCS 11544, pp. 283–299, 2019.
https://doi.org/10.1007/978-3-030-34029-2_19

chemical systems. These networks raise interesting questions regarding their structure. One of those questions asks whether a given network contains a particular pattern, which typically represents a specific behaviour of interest [1,4,12]. The problem of locating a particular pattern in the given network can be restated as a problem of locating a subgraph isomorphic to the given pattern graph in the network graph.

Formally, the Subgraph Isomorphism (SubIso) problem is, given two undirected graphs G and F, to decide whether there is a (not necessarily induced) subgraph of G isomorphic to F. Or, in other words, whether there is an adjacency-preserving injective mapping from vertices of F to vertices of G. Since we do not require the subgraph to be induced (or the mapping to preserve non-adjacencies), some authors call this variant of the problem Subgraph Monomorphism.

For many applications it is not enough to just learn that the pattern does occur in the network, but it is necessary to actually obtain the location of an occurrence of the pattern or rather of all occurrences of the pattern [18,25]. Because of that, we aim to solve the problem of subgraph enumeration, in which it is required to output all subgraphs of the network graph isomorphic to the pattern graph. In Subgraph Enumeration (SubEnum), given again two graphs G and F, the goal is to enumerate all subgraphs of G isomorphic to F. Note, that SubEnum is at least as hard as SubIso. We call the variants, where the problem is required to be induced IndSubIso and IndSubEnum, respectively.

As Clique, one of the problems on the Karp's original list of 21 NP-complete problems [15], is a special case of SubIso, the problem is NP-complete. Nevertheless, there are many heuristic algorithms for SubEnum, many of them based on ideas from constraint programming (see Sect. 1.1), which give results in reasonable time for most instances. However, for each of them there are rather small instances which they find genuinely hard, as pointed out by McCreesh et al. [23]. Therefore, developing an alternative approach that could possibly cope with these hard instances would be of interest.

In this paper we focus on the well known randomized color coding approach [2], which presumably has almost optimal worst case time complexity. Indeed, its time complexity is $\mathcal{O}\big(n_G^{\mathrm{TW}(F)+1}2^{\mathcal{O}(n_F)}\big)$ with memory requirements of $\mathcal{O}\big(n_G^{\mathrm{TW}(F)+1}\mathrm{TW}(F)n_F2^{n_F}\big)$, where n_G and n_F denote the number of vertices in the network graph G and the pattern graph F, respectively, and $\mathrm{TW}(F)$ is the treewidth of graph F—a measure of tree-likeness (see Sect. 1.2 for exact definitions). Moreover, we presumably cannot avoid the factor exponential in treewidth in the worst case running time, as Marx [21] presented an ETH[1]-based lower bound for Partitioned Subgraph Isomorphism problem.

Proposition 1 (Marx [21]). *If there is a recursively enumerable class $\mathcal{F}$ of graphs with unbounded treewidth, an algorithm $\mathcal{A}$, and an arbitrary function f such that $\mathcal{A}$ correctly decides every instance of* Partitioned Subgraph

[1] Exponential Time Hypothesis [14].

Isomorphism *with the smaller graph* F *in* $\mathcal{F}$ *in time* $f(F)n_G^{o(\mathrm{TW}(F)/\log \mathrm{TW}(F))}$, *then ETH fails.*

As the memory requirements of the color coding approach grow exponentially with treewidth of the pattern graph, existing implementations for subgraph enumeration based on this principle restrict themselves to paths [13] or trees [25], both having treewidth 1. As the real world applications might employ networks of possibly tens to hundreds of thousands of vertices and also pattern graphs with structure more complicated than trees, we need to significantly reduce the memory usage of the algorithm.

Using the principle of inclusion-exclusion, Amini et al. [3, Theorem 15] suggested a modification of the color coding algorithm, which can decide whether the pattern F occurs in the graph G in expected time $\mathcal{O}\big(n_G^{\mathrm{TW}(F)+1}2^{\mathcal{O}(n_F)}\big)$ with memory requirements reduced to $\mathcal{O}\big(n_G^{\mathrm{TW}(F)+1}\log n_F\big)$.[2] While single witnessing occurrence can be found by means of self-reduction (which is complicated in case of randomized algorithm), the inclusion-exclusion nature of the algorithm does not allow to find all occurrences of pattern in the graph, which is our main goal.

Therefore, our approach rather follows the paradigm of generating only those parts of a dynamic programming table that correspond to subproblems with a positive answer, recently called "positive instance driven" approach [28]. This further prohibits the use of the inclusion-exclusion approach of Amini et al. [3], since the inclusion-exclusion approach tends to use most of the table and the term $\mathcal{O}\big(n_G^{\mathrm{TW}(F)+1}\big)$ is itself prohibitive in the memory requirements for $\mathrm{TW}(F) \geq 2$.

Because of the time and memory requirements of the algorithm, for practical purposes we restrict ourselves to pattern graphs with at most 32 vertices.

Altogether, our main contribution is twofold:

- We provide a practical implementation of the color coding algorithm of Alon, Yuster, and Zwick [2] capable of processing large networks and (possibly disconnected) pattern graphs of small, yet not a priory bounded, treewidth.
- We supply a routine to extract the occurrences of the subgraphs found from a run of the algorithm.

It is important to note that all the modifications only improve the practical memory requirements and running time. The theoretical worst case time and space complexity remain the same as for the original color coding algorithm and the algorithm achieves these, e.g., if the network graph is complete. Also, in such a case, there are $n_G^{\Theta(n_F)}$ occurrences of the pattern graph in the network implying a lower bound on the running time of the enumeration part.

In Sect. 2 we describe our modifications to the algorithm and necessary tools used in the process. Then, in Sect. 3, we benchmark our algorithm on synthetic and realistic data and compare its performance with available existing implementations of algorithms for subgraph isomorphism and discuss the results obtained.

[2] While the formulation of Theorem 15 in [3] might suggest that the algorithm actually outputs a witnessing occurrence, the algorithm merely decides whether the number of occurrences is non-zero (see the proof of the theorem).

Section 4 presents future research directions. Parts of the paper not present in this extended abstract due to space restrictions, can be found it the ArXiv preprint [20] or in the full version of the paper.

1.1 Related Work

There are several algorithms tackling SubIso and its related variants. Some of them only solve the variant of subgraph counting, our main focus is however on algorithms actually solving SubEnum. Following Carletti et al. [7] and Kimmig et al. [16], we categorize the algorithms by the approach they use (see also Kotthoff et al. [17] for more detailed description of the algorithms). Many of the approaches can be used both for induced and non-induced variants of the problem, while some algorithms are applicable only for one of them.

Vast majority of known algorithms for the subgraph enumeration problem is based on the approach of representing the problem as a searching process. Usually, the state space is modelled as a tree and its nodes represent a state of a partial mapping. Finding a solution then typically resorts to the usage of DFS in order to find a path of mappings in the state space tree which is compliant with isomorphism requirements. The efficiency of those algorithms is largely based on early pruning of unprofitable paths in the state space. Indeed, McCreesh et al. [23] even measure the efficiency in the number of generated search tree nodes. The most prominent algorithms based on this idea are Ullmann's algorithm [29], VF algorithm and its variants [5,7,9,10] (the latest VF3 [5] only applies to IndSubEnum) and RI algorithm [4]. The differences between these algorithms are based both on employed pruning strategies and on the order in which the vertices of pattern graph are processed (i.e. in the shape of the state space tree).

Another approach is based on constraint programming, in which the problem is modelled as a set of variables (with respective domains) and constraints restricting simultaneous variable assignments. The solution is an assignment of values to variables in a way such that no constraint remains unsatisfied. In subgraph isomorphism, variables represent pattern graph vertices, their domain consists of target graph vertices to which they may be mapped and constraints ensure that the properties of isomorphism remain satisfied. Also in this approach, a state space of assignments is represented by a search tree, in which non-profitable branches are to be filtered. Typical algorithms in this category are LAD algorithm [26], Ullmann's bitvector algorithm [30], and Glasgow algorithm [22]. These algorithms differ in the constraints they use, the way they propagate constraints, and in the way they filter state space tree.

There are already some implementations based on the color coding paradigm, where the idea is to randomly color the input graph and search only for its subgraphs, isomorphic to the pattern graph, that are colored in distinct colors (see Sect. 2.1 for more detailed description). This approach is used in subgraph counting algorithms, e.g., in ParSE [31], FASCIA [24], and in [1], or in algorithms for path enumeration described in [25] or in [13]. Each of these algorithms, after the color coding step, tries to exploit the benefits offered by this technique in its own

way; although usually a dynamic programming sees its use. Counting algorithms as ParSE and FASCIA make use of specifically partitioned pattern graphs, which allow to use combinatorial computation. Weighted path enumeration algorithms [13,25] describe a dynamic programming approach and try to optimize it in various ways. However, to the best of our knowledge there is no color coding algorithm capable of enumerating patterns of treewidth larger than 1.

Our aim is to make step towards competitive implementation of color coding based algorithm for SubEnum, in order to see, where this approach can be potentially beneficial against the existing algorithms. To this end, we extend the comparisons of SubEnum algorithms [6,17,23] to color coding based algorithms, including the one proposed in this paper.

1.2 Basic Definitions

All graphs in this paper are undirected and simple. For a graph G we denote $V(G)$ its vertex set, n_G the size of this set, $E(G)$ its edge set, and m_G the size of its edge set.

As already said, we use the color coding algorithm. The algorithm is based on a dynamic programming on a nice tree decomposition of the pattern graph. We first define a tree decomposition and then its nice counterpart.

Definition 1. *A* tree decomposition *of a graph F is a triple (T, β, r), where T is a tree rooted at node r and $\beta\colon V(T) \mapsto 2^{V(F)}$ is a mapping satisfying: (i) $\bigcup_{x \in V(T)} \beta(x) = V(F)$; (ii) $\forall \{u, v\} \in E(F)\ \exists x \in V(T)$, such that $u, v \in \beta(x)$; (iii) $\forall u \in V(F)$ the nodes $\{x \in V(T) \mid u \in \beta(x)\}$ form a connected subtree of T.*

We shall denote bag $\beta(x)$ as $\mathcal{V}_x$. The width of tree decomposition (T, β, r) is $\max_{x \in V(T)} |\mathcal{V}_x| - 1$. Treewidth $\mathrm{TW}(F)$ of graph F is the minimal width of a tree decomposition of F over all such decompositions.

Definition 2. *A tree decomposition of a graph F is* nice *if $\deg_T(r) = 1$, $\mathcal{V}_r = \emptyset$, and each node $x \in V(T)$ is of one of the following four types:*

- Leaf node—*x has no children and $|\mathcal{V}_x| = 1$;*
- Introduce node—*x has exactly one child y and $\mathcal{V}_x = \mathcal{V}_y \cup \{u\}$ for some $u \in V(F) \setminus \mathcal{V}_y$;*
- Forget node—*x has exactly one child y and $\mathcal{V}_x = \mathcal{V}_y \setminus \{u\}$ for some $u \in \mathcal{V}_y$;*
- Join node—*x has exactly two children y, z and $\mathcal{V}_x = \mathcal{V}_y = \mathcal{V}_z$.*

Note that for practical purposes, we use a slightly modified definition of nice tree decomposition in this paper. As the algorithm starts the computation in a leaf node, using the standard definition with empty bags of leaves [11] would imply that the tables for leaves would be somewhat meaningless and redundant. Therefore, we make bags of leaf nodes contain a single vertex.

Definition 3. *For a tree decomposition (T, β, r), we denote by $\mathcal{V}_x^*$ the set of vertices in $\mathcal{V}_x$ and in $\mathcal{V}_y$ for all descendants y of x in T. Formally $\mathcal{V}_x^* = \mathcal{V}_x \cup \bigcup_{y \text{ is a descendant of } x \text{ in } T} \mathcal{V}_y$.*

Note that, by Definition 3, for the root r of T we have $\mathcal{V}_r^* = V(F)$ and $F[\mathcal{V}_r^*] = F$.

2 Algorithm Description

In this section we first briefly describe the idea of the original color coding algorithm [2], show, how to alter the computation in order to reduce its time and memory requirements, and describe implementation details and further optimizations of the algorithm. Due to space restrictions, the way to obtain a nice tree decomposition of the pattern and the reconstruction of results are deferred to the full version of the paper.

2.1 Idea of the Algorithm

The critical idea of color coding is to reduce the problem to its colorful version. For a graph G and a pattern graph F, we color the vertices of G with exactly n_F colors. We use the randomized version, i.e., we create a random coloring $\zeta\colon V(G) \mapsto \{1, 2, \ldots, n_F\}$. After the coloring, the algorithm considers as valid only subgraphs G' of G that are colorful copies of F as follows.

Definition 4. *Subgraph G' of a graph G is a* colorful copy *of F with respect to coloring $\zeta\colon V(G) \mapsto \{1, 2, \ldots, n_F\}$, if G' is isomorphic to F and all of its vertices are colored by distinct colors in ζ.*

As the output of the algorithm heavily depends on the chosen random coloring of G, in order to reach some predefined success rate of the algorithm, we need to repeat the process of coloring several times. The probability of a particular occurrence of pattern graph F becoming colorful with respect to the random coloring is $\frac{n_F!}{n_F^{n_F}}$, which tends to e^{-n_F} for large n_F. Therefore, by running the algorithm $\mathrm{e}^{n_F \log \frac{1}{\varepsilon}}$ times, each time with a random coloring $\zeta\colon V(G) \mapsto \{1, 2, \ldots, n_F\}$, the probability that an existing occurrence of the pattern will be revealed in none of the runs is at most ε. While using more colors can reduce the number of iterations needed, it also significantly increases the memory requirements. Hence, we stick to n_F colors. Even though it is possible to derandomize such algorithms, e.g., by the approach shown in [11], in practice the randomized approach usually yields the results much quicker, as discussed in [25]. Moreover, we are not aware of any actual implementation of the derandomization methods.

The main computational part of the algorithm is a dynamic programming. The target is to create a graph isomorphism $\Phi\colon V(F) \mapsto V(G)$. We do so by traversing the nice tree decomposition (T, β, r) of the pattern graph F and at each node $x \in V(T)$ of the tree decomposition, we construct possible partial mappings $\varphi\colon \mathcal{V}_x^* \to V(G)$ with regard to required colorfulness of the copy. Combination of partial mappings consistent in colorings then forms a desired resulting mapping.

The semantics of the dynamic programming table is as follows. For any tree decomposition node $x \in V(T)$, any partial mapping $\varphi\colon \mathcal{V}_x \mapsto V(G)$ and any color subset $C \subseteq \{1, 2, \ldots, n_F\}$, we define $\mathcal{D}(x, \varphi, C) = 1$ if there is an isomorphism Φ of $F[\mathcal{V}_x^*]$ to a subgraph G' of G such that:

(i) for all $u \in \mathcal{V}_x, \Phi(u) = \varphi(u)$;
(ii) G' is a colorful copy of $F[\mathcal{V}_x^*]$ using exactly the colors in C, that is, $\zeta(\Phi(\mathcal{V}_x^*)) = C$ and ζ is injective on $\Phi(\mathcal{V}_x^*)$.

If there is no such isomorphism, then we let $\mathcal{D}(x, \varphi, C) = 0$. We denote all configurations (x, φ, C) for which $\mathcal{D}(x, \varphi, C) = 1$ as *nonzero* configurations.

The original version of the algorithm is based on top-down dynamic programming approach with memoization of already computed results. That immediately implies a big disadvantage of this approach—it requires the underlying dynamic programming table (which is used for memoization) to be fully available throughout the whole run of the algorithm. To avoid this inefficiency in our modification we aim to store only nonzero configurations, similarly to the recent "positive instance driven" dynamic programming approach [28].

2.2 Initial Algorithm Modification

In our implementation, we aim to store only nonzero configurations, therefore we need to be able to construct nonzero configurations of a parent node just from the list of nonzero configurations in its child/children.

We divide the dynamic programming table $\mathcal{D}$ into lists of nonzero configurations, where each nice tree decomposition node has a list of its own. Formally, for every node $x \in V(T)$, let us denote by $\mathcal{D}_x$ a list of all mappings φ with a list of their corresponding color sets C, for which $\mathcal{D}(x, \varphi, C) = 1$. The list $\mathcal{D}_x$ for all $x \in V(T)$ is, in terms of contained information, equivalent to maintaining the whole table $\mathcal{D}$—all configurations not present in the lists can be considered as configurations with a result equal to zero.

Dynamic Programming Description. We now describe how to compute the lists $\mathcal{D}(x, \varphi, C)$ for each type of a nice tree decomposition node.

For a *leaf* node $x \in T$, there is only a single vertex u in $\mathcal{V}_x^*$ to consider. We can thus map u to all possible vertices of G, and we obtain a list with n_G partial mappings φ, in which the color list for each mapping contains a single color set $\{\zeta(\varphi(u))\}$.

For an *introduce* node $x \in T$ and its child y in T, we denote by u the vertex being introduced in x, i.e., $\{u\} = \mathcal{V}_x \backslash \mathcal{V}_y$. For all nonzero combinations of a partial mapping and a color set (φ', C') in the list $\mathcal{D}_y$, we try to extend φ' by all possible mappings of the vertex u to the vertices of G. We denote one such a mapping as φ. We can consider mapping φ as correct, if (i) the new mapping $\varphi(u)$ of the vertex u extends the previous colorset C', that is, $C = C' \cup \{\zeta(\varphi(u))\} \neq C'$, and (ii) φ is *edge consistent*, that is, for all edges $\{v, w\} \in E(F)$ between currently mapped vertices, i.e., in our case $v, w \in \mathcal{V}_x$, there must be an edge $\{\varphi(v), \varphi(w)\} \in E(G)$. However, because φ' was by construction already edge consistent, it suffices to check the edge consistency only for all edges in $F[\mathcal{V}_x]$ with u as one of their endpoints, i.e., for all edges $\{u, w\} \in E(F[\mathcal{V}_x])$ with $w \in N_{F[\mathcal{V}_x]}(u)$. After checking those two conditions, we can add (φ, C) to $\mathcal{D}_x$.

Due to space restrictions, the computation in forget and join nodes is deferred to the full version of the paper.

Because we build the result from the leaves of the nice tree decomposition, we employ a recursive procedure on its root, in which we perform the computations in a way of a post-order traversal of a tree. From each visited node, we obtain a bottom-up dynamic programming list of nonzero configurations. After the whole nice tree decomposition is traversed, we obtain a list of configurations, that were valid in its root. Such configurations thus represent solutions found during the algorithm, from which we afterwards reconstruct results. Note that as we prepend a root with no vertices in its bag to the nice tree decomposition, there is a nonzero number of solutions if and only if, at the end of the algorithm, the list $\mathcal{D}_r$ contains a single empty mapping using all colors.

2.3 Further Implementation Optimizations

Representation of Mappings. For mapping representation, we suppose that the content of all bags of the nice tree decomposition stays in the same order during the whole algorithm. This natural and easily satisfied condition allows us to represent a mapping $\varphi\colon \mathcal{V}_x \mapsto V(G)$ in a nice tree decomposition node x simply by an ordered tuple of $|\mathcal{V}_x|$ vertices from G. From this, we can easily determine which vertex from F is mapped to which vertex in G. Also, for a mapping in an introduce or a forget node, we can describe a position in the mapping, on which the process of introducing/forgetting takes place.

Representation of Color Sets. We represent color sets as bitmasks, where the i-th bit states whether color i is contained in the set or not. For optimization purposes, we represent bitmasks with an integer number. As we use n_F colors in the algorithm and restricted ourselves to pattern graphs with at most 32 vertices, we represent a color set with a 32-bit number.

Compressing the Lists. Because we process the dynamic programming lists one mapping at a time, we store these lists in a compressed way and decompress them only on a mapping retrieval basis. Due to space restrictions, the exact way we serialize the records, the use of delta compression and a special library is deferred to the full version of the paper.

Masking Unprofitable Mappings. Our implementation supports an extended format of input graphs where one can specify for each vertex of the network, which vertices of the pattern can be mapped to it. This immediately yields a simple degree-based optimization. Before the run of the main algorithm, we perform a linear time preprocessing of input graphs and only allow a vertex $y \in V(F)$ to be mapped to a vertex $x \in V(G)$ if $\deg_G(x) \geq \deg_F(y)$.

Mapping Expansion Optimizations. The main "brute-force" work of the algorithm is performed in two types of nodes—leaf and introduce nodes, as we need to try all possible mappings of a particular vertex in a leaf node or all possible mappings of an introduced vertex in a introduce node to a vertex from G. We describe ways to optimize the work in introduce nodes in this paragraph.

Let x be an introduce node, u the vertex introduced and φ a mapping from a nonzero configuration for the child of x. We always need to check whether the new mapping of u is edge consistent with the mapping φ of the remaining vertices for the corresponding bag, i.e., whether all edges of F incident on u would be realized by an edge in G. Therefore, if u has any neighbors in $F[\mathcal{V}_x]$, then a vertex of G is a candidate for the mapping of u only if it is a neighbor of all vertices in the set $\varphi(N_{F[\mathcal{V}_x]}(u))$, i.e., the vertices of G, where the neighbors of u in F are mapped. Hence, we limit the number of candidates by using the adjacency lists of the already mapped vertices.

In the case $\deg_{F[\mathcal{V}_x]}(u) = 0$ we have to use different approach. The pattern graphs F tend to be smaller than the input graphs G by several orders of magnitude. Hence, if the introduced vertex is in the same connected component of F as some vertex already present in the bag, a partial mapping processed in an introduce node anchors the possible resulting component to a certain position in G. Due to space restrictions, the exact way to exploit that is deferred to the full version of the paper.

Only if there is no vertex in the bag sharing a connected component of F with u, we have to fall back to trying all possible mappings.

3 Experimental Results

The testing was performed on a 64-bit linux system with Intel Xeon CPU E3-1245v6@3.70GHz and 32 GB 1333 MHz DDR3 SDRAM memory. The module was compiled with `gcc` compiler (version 7.3.1) with `-O3` optimizations enabled. Implementation and instances utilized in the testing are available at http://users.fit.cvut.cz/malikjo1/subiso/. All results are an average of 5 independent measurements.

We evaluated our implementation in several ways. Firstly, we compare available implementations on two different real world source graphs and a set of more-or-less standard target graph patterns. Secondly, we compare available implementations on instances from ICPR2014 Contest on Graph Matching Algorithms for Pattern Search in Biological Databases [8] with suitably small patterns. We also adapt the idea of testing the algorithms on Erdős-Rényi random graphs [23].

3.1 Algorithm Properties and Performance

In the first two subsection we used two different graphs of various properties as target graph G. The first instance, IMAGES, is built from an segmented image, and is a courtesy of [27]. It consists of 4838 vertices and 7067 edges. The second

instance, TRANS, is a graph of transfers on bank accounts. It is a very sparse network, which consists of 45733 vertices and 44727 undirected edges. Due to space restrictions, the results on this dataset are deferred to the full version of the paper.

For the pattern graphs, we first use a standard set of basic graph patterns, as the treewidth of such graphs is well known and allows a clear interpretation of the results. In particular, we use paths, stars, cycles, an complete graphs on n vertices, denoted P_n, S_n, C_n, and K_n with treewidth 1, 1, 2, and $n-1$, respectively. We further used grids $G_{n,m}$ on $n \times m$ vertices, with treewidth $\min\{n, m\}$. Secondly, we use a special set of pattern graphs in order to demonstrate performance on various patterns. Patterns A, B, C, and D have 9, 7, 9, and 7 vertices, 8, 7, 12, 6 edges, and treewidth 1, 2, 2, and 2, respectively. Patterns A, B, and D appear in both dataset, pattern C in neither and pattern D is disconnected. Description of these pattern graphs is deferred to the full version of the paper.

Due to randomization, in order to achieve some preselected constant error rate, we need to repeat the computation more than once. The number of found results thus depends not only on the quality of the algorithm, but also on the choice of the number of its repetitions. Hence, it is logical to measure performance of the single run of the algorithm. Results from such a testing, however, should be still taken as a rough average, because the running time of a single run of the algorithm depends on many factors.

Therefore, we first present measurements, where we average the results of many single runs of the algorithm (Table 1). We average not only the time and space needed, but also the number of found subgraphs. To obtain the expected time needed to run the whole algorithm, it suffices to sum the time needed to create a nice tree decomposition and ℓ times the time required for a single run, if there are ℓ runs in total.

Table 1. Performance of a single run of the algorithm on IMAGES dataset.

Pattern	Comp. time [ms]	Comp. memory [MB]	Occurrences
$\mathcal{P}_5$	240	12.73	3488.21
$\mathcal{P}_{10}$	160	8.52	732.46
$\mathcal{P}_{15}$	90	10.54	76.18
$\mathcal{S}_5$	4	5.37	114.72
$\mathcal{C}_5$	20	7.24	239.17
$\mathcal{C}_{10}$	70	9.34	26.64
$\mathcal{K}_4$	5	6.46	0
$G_{3,3}$	90	13.42	0
Pattern A	80	9.14	292.48
Pattern B	10	7.17	6.85
Pattern C	10	5.30	0
Pattern D	40	10.14	426.76

3.2 Comparison on Real World Graphs and Fixed Graph Patterns

We compare our implementation to three other tools for subgraph enumeration: RI algorithm [4] (as implemented in [19]), LAD algorithm [26] and color coding algorithm for weighted path enumeration [13] (by setting, for comparison purposes, all weights of edges to be equal). The comparison is done on the instances from previous subsection and only on pattern graphs which occur at least once in a particular target graph.

In comparison, note the following specifics of measured algorithms. The RI algorithm does not support outputting solutions, which might positively affect its performance. LAD algorithm uses adjacency matrix to store input graphs, and thus yields potentially limited use for graphs of larger scale. Neither of RI or LAD algorithms supports enumeration of disconnected patterns.[3] Also we did not measure the running time of the weighted path algorithm on non-path queries and also on Trans dataset, as its implementation is limited to graph sizes of at most 32 000.

We run our algorithm repeatedly to achieve an error rate of $\varepsilon = \frac{1}{e}$. In order to be able to measure the computation for larger networks with many occurrences of the pattern, we measure only the time required to retrieve no more than first 100 000 solutions and we also consider running time greater than 10 min (600 s) as a timeout. Since we study non-induced occurrences (and due to automorphisms) there might be several ways to map the pattern to the same set of vertices. Other measured algorithms do count all of them. Our algorithm can behave also like this, or can be switched to count only occurrences that differ in vertex sets. For the sake of equal measurement, we use the former version of our algorithm.

From Table 2, we can see that RI algorithm outperforms all other measured algorithms. We can also say our algorithm is on par with LAD algorithm, as the results of comparison of running times are similar, but vary instance from instance. Our algorithm nevertheless clearly outperforms another color coding algorithm, which on one hand solves more complicated problem of weighted paths, but on the another, is still limited only to paths. Also, our algorithm is the only algorithm capable of enumerating disconnected patterns.

The weak point of the color coding approach (or possibly only of our implementation) appears to be the search for a pattern of larger size with very few (or possibly zero) occurrences. To achieve the desired error rate, we need to repeatedly run the algorithm many times. Therefore our algorithm takes longer time to run on some instances (especially close to zero-occurrence ones), which are easily solved by the other algorithms.

[3] When dealing with disconnected patterns, one could find the components of the pattern one by one, omitting the vertices of the host graph used by the previous component. However, this would basically raise the running time of the algorithm to the power equal to the number of components of the pattern graph.

Table 2. Comparison of running time on IMAGES dataset (in seconds).

Pattern	Our algorithm	RI algorithm	LAD algorithm	Weighted path
$\mathcal{P}_5$	31.12	0.11	28.86	362.41
$\mathcal{P}_{10}$	53.17	1.25	13.63	> 600
$\mathcal{P}_{15}$	104.30	3.7	8.18	> 600
$\mathcal{S}_5$	0.94	0.07	0.43	–
$\mathcal{C}_5$	4.98	0.14	35.18	–
$\mathcal{C}_{10}$	151.25	3.44	174.27	–
Pattern A	43.11	0.82	36.60	–
Pattern B	91.93	0.41	0.83	–
Pattern D	23.54	–	–	–

3.3 ICPR2014 Contest Graphs

To fully benchmark our algorithm without limitations on time or number of occurrences found, we perform a test on ICPR2014 Contest on Graph Matching Algorithms for Pattern Search in Biological Databases [8].

In particular, we focus our attention on a MOLECULES dataset, containing 10,000 (target) graphs representing the chemical structures of different small organic compounds and on a PROTEINS dataset, which contains 300 (target) graphs representing the chemical structures of proteins and protein backbones. Target graphs in both datasets are sparse and up to 99 vertices or up 10,081 vertices for MOLECULES and PROTEINS, respectively.

In order to benchmark our algorithm without limiting its number of iterations, we focus on pattern graphs of small sizes, which offer reasonable number of iterations for an error rate of $\frac{1}{e}$. Both datasets contain 10 patterns for each of considered sizes constructed by randomly choosing connected subgraphs of the target graphs. We obtained an average matching time of all pattern graphs of a given size to all target graphs in a particular dataset.

Table 3. Comparison of average running time on ICPR2014 graphs

Targets	Pattern size	Our algorithm	LAD algorithm	RI algorithm
MOLECULES	4	0.01	0.01	0.01
MOLECULES	8	0.67	0.14	0.01
XSC PROTEINS	8	19.45	8.83	0.51

From the results in Table 3, we can see our algorithm being on par with LAD algorithm, while being outperformed by RI algorithm. However, we mainly include these results as a proof of versatility of our algorithm. As discussed

in [23], benchmarks created by constructing subgraphs of target graphs do not necessarily encompass the possible hardness of some instances and might even present a distorted view on algorithms' general performance. Thus, in the following benchmark we opt to theoretically analyze our algorithm.

3.4 Erdős-Rényi Graph Setup

In order to precisely analyze the strong and weak points of our algorithm we measure its performance is a setting where both the pattern and the target are taken as an Erdős-Rényi random graph of fixed size with varying edge density and compare the performance of our algorithm with the analysis of McCreesh et al. [23], which focused on algorithms Glasgow, LAD, and VF2.

An Erdős-Rényi graph $G(n, p)$ is a random graph on n vertices where each edge is included in the graph independently at random with probability p. We measure the performance on target graph of 150 vertices and pattern graph of 10 vertices with variable edge probabilities. As our algorithm cannot be classified in terms of search nodes used (as in [23]), we measure the time needed to complete 10 iterations of our algorithm.

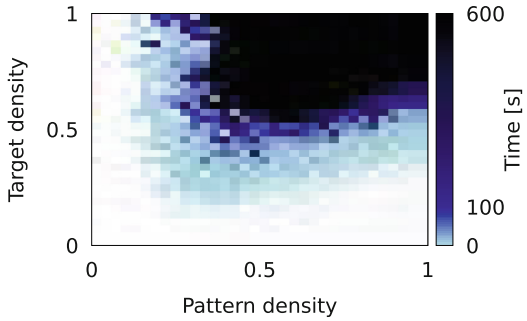

Fig. 1. Behavior for target graph of 150 vertices and pattern graph of 10 vertices. The x-axis is the pattern edge probability, the y-axis is the target edge probability, from 0 to 1 with step of 0.03. Graph shows the time required for our algorithm to complete 10 iterations (the darker, the more time is required). Black regions indicate instances on which a timeout of 600 s occurred.

From Fig. 3 we can see our algorithm indeed follows a well observed phase transition (transition between instances without occurrence of the pattern and with many occurrences of the pattern). If we compare our results from Fig. 1 to the results of [23], we can see that hard instances for our algorithm start to occur later (in terms of edge probabilities). However, due to the almost linear dependency of treewidth on edge probabilities (see Fig. 2), hard instances for our algorithm concentrate in the "upper right corner" of the diagram, which contains dense graphs with naturally large treewidth (Fig. 4).

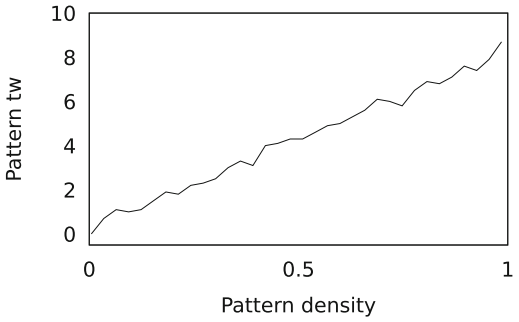

Fig. 2. Correspondence of treewidth to the edge probability of a pattern graph with 10 vertices.

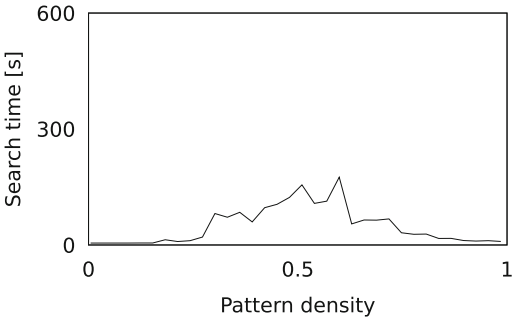

Fig. 3. Time needed to complete 10 iterations of our algorithm on a target graph of 150 vertices with edge probability of 0.5 and pattern graph of 10 vertices with variable edge probability.

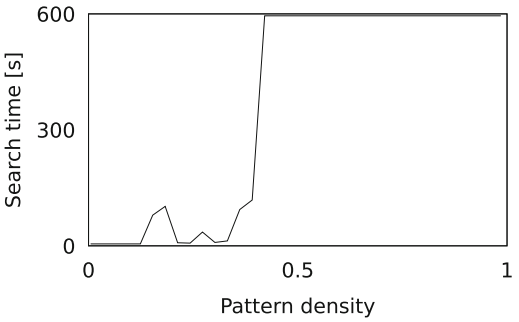

Fig. 4. Time needed to complete 10 iterations of our algorithm on a target graph of 150 vertices with edge probability of 0.8 and pattern graph of 10 vertices with variable edge probability.

Therefore, it seems that our algorithm complements the portfolio of algorithms studied by Kotthoff et al. [17] by an algorithm suitable just below the phase transition (in view of Fig. 1).

4 Conclusion

We described an efficient implementation of the well known color coding algorithm for the subgraph isomorphism problem. Our implementation is the first color-coding based algorithm capable of enumerating all occurrences of patterns of treewidth larger than one. Moreover, we have shown that our implementation is competitive with existing state-of-the-art solutions in the setting of locating small pattern graphs. As it exhibits significantly different behaviour than other solutions, it can be an interesting contribution to the portfolio of known algorithms [17,23].

As an obvious next step, the algorithm could be made to run in parallel. We also wonder whether the algorithm could be significantly optimized even further, possibly using some of the approaches based on constraint programming.

References

1. Alon, N., Dao, P., Hajirasouliha, I., Hormozdiari, F., Sahinalp, S.: Biomolecular network motif counting and discovery by color coding. Bioinformatics **24**, 241–249 (2008)
2. Alon, N., Yuster, R., Zwick, U.: Color-coding. J. ACM **42**(4), 844–856 (1995)
3. Amini, O., Fomin, F.V., Saurabh, S.: Counting subgraphs via homomorphisms. SIAM J. Discrete Math. **26**(2), 695–717 (2012)
4. Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., Ferro, A.: A subgraph isomorphism algorithm and its application to biochemical data. BMC Bioinform. **14**, 1–13 (2013)
5. Carletti, V., Foggia, P., Saggese, A., Vento, M.: Introducing VF3: a new algorithm for subgraph isomorphism. In: Foggia, P., Liu, C.-L., Vento, M. (eds.) GbRPR 2017. LNCS, vol. 10310, pp. 128–139. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-58961-9_12
6. Carletti, V., Foggia, P., Vento, M.: Performance comparison of five exact graph matching algorithms on biological databases. In: Petrosino, A., Maddalena, L., Pala, P. (eds.) ICIAP 2013. LNCS, vol. 8158, pp. 409–417. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41190-8_44
7. Carletti, V., Foggia, P., Vento, M.: VF2 Plus: an improved version of VF2 for biological graphs. In: Liu, C.-L., Luo, B., Kropatsch, W.G., Cheng, J. (eds.) GbRPR 2015. LNCS, vol. 9069, pp. 168–177. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18224-7_17
8. Carletti, V., Foggia, P., Vento, M., Jiang, X.: Report on the first contest on graph matching algorithms for pattern search in biological databases. In: Liu, C.-L., Luo, B., Kropatsch, W.G., Cheng, J. (eds.) GbRPR 2015. LNCS, vol. 9069, pp. 178–187. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-18224-7_18
9. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: Performance evaluation of the VF graph matching algorithm. In: 10th International Conference on Image Analysis and Processing, ICIAP 1999. pp. 1172–1177. IEEE Computer Society (1999)

10. Cordella, L.P., Foggia, P., Sansone, C., Vento, M.: A (sub)graph isomorphism algorithm for matching large graphs. IEEE Trans. Pattern Anal. Mach. Intell. **26**(10), 1367–1372 (2004)
11. Cygan, M., et al.: Parameterized Algorithms. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21275-3
12. Dahm, N., Bunke, H., Caelli, T., Gao, Y.: Efficient subgraph matching using topological node feature constraints. Pattern Recogn. **48**(2), 317–330 (2015)
13. Hüffner, F., Wernicke, S., Zichner, T.: Algorithm engineering for color-coding with applications to signaling pathway detection. Algorithmica **52**(2), 114–132 (2008)
14. Impagliazzo, R., Paturi, R.: On the complexity of k-SAT. J. Comput. Syst. Sci. **62**(2), 367–375 (2001)
15. Karp, R.M.: Reducibility among combinatorial problems. In: Symposium on the Complexity of Computer Computations, COCO 1972, The IBMResearch Symposia Series, pp. 85–103. Plenum Press, New York (1972)
16. Kimmig, R., Meyerhenke, H., Strash, D.: Shared memory parallel subgraph enumeration. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp. 519–529. IEEE Computer Society (2017)
17. Kotthoff, L., McCreesh, C., Solnon, C.: Portfolios of subgraph isomorphism algorithms. In: Festa, P., Sellmann, M., Vanschoren, J. (eds.) LION 2016. LNCS, vol. 10079, pp. 107–122. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-50349-3_8
18. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. In: 2001 IEEE International Conference on Data Mining, pp. 313–320. IEEE Computer Society (2001)
19. Leskovec, J., Sosič, R.: Snap: a general-purpose network analysis and graph-mining library. ACM Trans. Intel. Syst. Technol. (TIST) **8**(1), 1 (2016)
20. Malík, J., Suchý, O., Valla, T.: Efficient implementation of color coding algorithm for subgraph isomorphism problem. CoRR abs/1908.11248 (2019)
21. Marx, D.: Can you beat treewidth? Theory Comput. **6**(1), 85–112 (2010)
22. McCreesh, C., Prosser, P.: A parallel, backjumping subgraph isomorphism algorithm using supplemental graphs. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 295–312. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23219-5_21
23. McCreesh, C., Prosser, P., Solnon, C., Trimble, J.: When subgraph isomorphism is really hard, and why this matters for graph databases. J. Artif. Intell. Res. **61**, 723–759 (2018)
24. Slota, G.M., Madduri, K.: Fast approximate subgraph counting and enumeration. In: ICPP 2013, pp. 210–219. IEEE Computer Society (2013)
25. Slota, G.M., Madduri, K.: Parallel color-coding. Parallel Comput. **47**, 51–69 (2015)
26. Solnon, C.: AllDifferent-based filtering for subgraph isomorphism. Artif. Intell. **174**(12–13), 850–864 (2010)
27. Solnon, C., Damiand, G., de la Higuera, C., Janodet, J.C.: On the complexity of submap isomorphism and maximum common submap problems. Pattern Recogn. **48**(2), 302–316 (2015)
28. Tamaki, H.: Positive-instance driven dynamic programming for treewidth. In: ESA 2017. LIPIcs, vol. 87, pp. 68:1–68:13. Schloss Dagstuhl (2017)
29. Ullmann, J.R.: An algorithm for subgraph isomorphism. J. ACM **23**(1), 31–42 (1976)
30. Ullmann, J.R.: Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. J. Exp. Algorithmics **15**, 1.6:1.1–1.6:1.64 (2011)
31. Zhao, Z., Khan, M., Kumar, V.S.A., Marathe, M.V.: Subgraph enumeration in large social contact networks using parallel color coding and streaming. In: ICPP 2010, pp. 594–603. IEEE Computer Society (2010)