



ASSIGNMENT OF MASTER'S THESIS

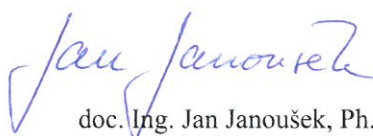
Title: Subgraph Isomorphism Algorithm Based on Color Coding
Student: Bc. Josef Malik
Supervisor: RNDr. Ondřej Suchý, Ph.D.
Study Programme: Informatics
Study Branch: System Programming
Department: Department of Theoretical Computer Science
Validity: Until the end of summer semester 2016/17

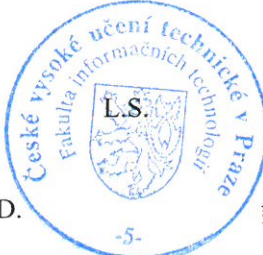
Instructions

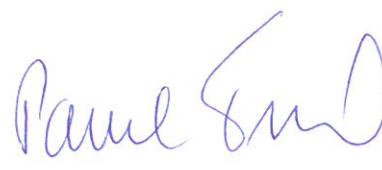
Survey known results for the Subgraph Isomorphism problem.
Familiarize yourself with the Color Coding algorithm for Subgraph Isomorphism.
Survey known algorithms for finding tree decomposition of graphs.
Implement in C a selected algorithm for tree decomposition of graphs.
Implement in C an algorithm to convert a given tree decomposition to a nice one.
Propose suitable modifications of the Color Coding algorithm in order to reduce the time and memory requirements.
Implement in C the modified Color Coding algorithm for Subgraph Isomorphism and further optimize the code possibly using suitable libraries.
Test the resulting program on suitable instances and compare different variants of it with each other and with other available implementations of algorithms for the problem, if there are any.

References

Noga Alon, Raphael Yuster, Uri Zwick: Color-Coding. J. ACM 42(4): 844-856 (1995)
Hans L. Bodlaender, Fedor V. Fomin, Aric M. C. A. Koster, Dieter Kratsch, Dimitrios M. Thilikos: On exact algorithms for treewidth. ACM Transactions on Algorithms 9(1): 12 (2012)


doc. Ing. Jan Janoušek, Ph.D.
Head of Department




prof. Ing. Pavel Tvrdík, CSc.
Dean

Prague February 8, 2016

CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF THEORETICAL COMPUTER SCIENCE



Master's thesis

Subgraph Isomorphism Algorithm Based on Color Coding

Bc. Josef Malík

Supervisor: RNDr. Ondřej Suchý, Ph.D.

9th May 2016

Acknowledgements

In the first place, I would like to express my gratitude to my supervisor RNDr. Ondřej Suchý, Ph.D. for his knowledgeable guidance and an extraordinary amount of effort he put towards helping me with this thesis. Also, I would like to thank my family and my friends for supporting me throughout the studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the “Work”), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 9th May 2016

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2016 Josef Malík. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Malík, Josef. *Subgraph Isomorphism Algorithm Based on Color Coding*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstract

This thesis describes a solution to the subgraph isomorphism problem using the color coding technique. The subgraph isomorphism problem, its variants, and its applications are shown. The problem of constructing a nice tree decomposition of optimal width for a given graph is addressed, as such structure is required for the corresponding approach. As a main part of the thesis, several modifications and optimizations of the original color coding algorithm are proposed. Practical result of this work is a module implemented in C designated to solve large instances of the subgraph isomorphism problem along with the enumeration of the results.

Keywords subgraph isomorphism problem, color coding, tree decomposition, treewidth

Abstrakt

Tato práce popisuje řešení problému izomorfismu podgrafů pomocí techniky barevného kódování. V práci je popsán problém izomorfismu podgrafů, jeho varianty a jeho aplikace. Dále je rozebrán problém tvorby hezkého stromového rozkladu optimální šířky pro zadaný graf, jelikož takováto struktura je pro daný přístup vyžadována. Jako hlavní část práce je popsáno několik modifikací a optimalizací původního algoritmu založeného na barevném kódování. Praktickým výsledkem práce je modul implementovaný v jazyce C, který je navržen pro řešení velkých instancí problému izomorfismu podgrafů včetně výpisu nalezených výsledků.

Klíčová slova problém izomorfismu podgrafů, barevné kódování, stromový rozklad, stromová šířka

Contents

| | |
|---|-----------|
| Introduction | 1 |
| Goals of the thesis | 1 |
| Thesis structure | 2 |
| 1 Subgraph isomorphism problem | 3 |
| 1.1 Applications of subgraph isomorphism | 3 |
| 1.2 Graph theory notation | 4 |
| 1.3 Subgraph isomorphism problem statement and its variations . . | 5 |
| 1.4 Approaches to solving the subgraph isomorphism problem . . . | 8 |
| 1.5 Chosen problem variant and approach to it | 8 |
| 2 Tree decomposition and treewidth | 11 |
| 2.1 Definitions related to tree decomposition | 11 |
| 2.2 Tree decomposition construction | 13 |
| 2.3 Elimination ordering construction | 19 |
| 2.4 Nice tree decomposition construction | 22 |
| 3 Color coding algorithm | 29 |
| 3.1 Color coding technique | 29 |
| 3.2 Main algorithm idea | 30 |
| 3.3 Original top-down algorithm | 32 |
| 3.4 Modified bottom-up algorithm | 35 |
| 3.5 Final form of the algorithm and its properties | 55 |
| 4 Implementation | 59 |
| 4.1 Chosen technologies | 59 |
| 4.2 Licensing and availability | 59 |
| 4.3 LibUCW library | 59 |
| 4.4 Structure of the module | 60 |
| 4.5 Usage of the module | 61 |

| | |
|--|-----------|
| 5 Results and performance | 63 |
| 5.1 Testing enviroment | 63 |
| 5.2 Testing data | 63 |
| 5.3 Qualitative tests | 64 |
| 5.4 Quantitative tests | 64 |
| 5.5 Modification tests | 70 |
| 5.6 Discussion of results and comparison to related work | 71 |
| Conclusion | 73 |
| Future work | 73 |
| Bibliography | 75 |
| A Acronyms | 77 |
| B Contents of enclosed CD | 79 |
| C Module installation guide | 81 |
| C.1 Prerequisites | 81 |
| C.2 Installation | 81 |

List of Figures

| | | |
|-----|---|----|
| 1.1 | Difference between induced and non-induced subgraph | 5 |
| 1.2 | Example of two isomorphic graphs | 6 |
| 2.1 | Example of a tree decomposition of a graph | 12 |
| 2.2 | Introduce node example | 23 |
| 2.3 | Forget node example | 23 |
| 2.4 | Join node example | 23 |
| 2.5 | Start node example | 23 |
| 2.6 | Nice tree decomposition root construction | 24 |
| 2.7 | Nice tree decomposition node connecting | 24 |
| 2.8 | Nice tree decomposition degree reduction | 25 |
| 3.1 | Top-down dynamic programming table | 32 |
| 3.2 | Bottom-up dynamic programming list | 36 |
| 3.3 | Representation of partial mappings | 39 |
| 3.4 | Serialization of mappings from a dynamic programming list | 41 |
| 3.5 | Disruption in the order of mappings in an introduce node | 45 |
| 3.6 | Disruption in the order of mappings in a forget node | 47 |
| 3.7 | Mapping expansion modification based on edge consistency | 49 |
| 3.8 | Mapping expansion modification based on eccentricity | 51 |
| 4.1 | Graph representation in files | 62 |

List of Tables

| | | |
|-----|---|----|
| 3.1 | Memory required by the top-down algorithm for $\text{TW}(F) = 1$ | 34 |
| 3.2 | Memory required by the top-down algorithm for $\text{TW}(F) = 2$ | 35 |
| 3.3 | Bits needed to encode integer numbers by the LibUCW variable length code | 42 |
| 3.4 | Number of the algorithm repetitions needed to achieve $\varepsilon = 0.5$. . | 55 |
| 3.5 | Number of the algorithm repetitions needed to achieve $\varepsilon = \frac{1}{e}$. . . | 56 |
| 3.6 | Number of the algorithm repetitions needed to achieve $\varepsilon = 0.01$. . | 56 |
| | | |
| 5.1 | Treewidth of pattern graphs | 64 |
| 5.2 | Performance of nice tree decomposition construction | 65 |
| 5.3 | Performance of a single run of the algorithm on ECOLI dataset . . | 67 |
| 5.4 | Performance of a single run of the algorithm on TRANS dataset . . | 68 |
| 5.5 | Performance of a single run of the algorithm on SLASH dataset . . | 69 |
| 5.6 | Time and space needed with uncompressed buffers | 70 |
| 5.7 | Time and space needed with compressed buffers | 70 |
| 5.8 | Compressed buffer space storage efficiency | 71 |
| 5.9 | Number of vertex candidates with/without mapping optimizations | 71 |

Introduction

Many real-world domains incorporate large and complex networks of interconnected units. Namely speaking, we can see such networks represented in many examples – social networks, the Internet, or biological and chemical systems. Among with these networks arise interesting questions regarding their structure. One of those questions may ask whether a given network contains a particular pattern. By successfully answering this question, we can gain a very needed piece of information about the network we conduct the search in. Most notably, we can learn about the regularity of the given pattern in the structure of the network.

To approach the situation we can naturally represent both networks and patterns as graphs. The problem of locating a particular pattern in the given network can then be restated as a problem of locating a subgraph isomorphic to the given pattern graph in the network graph. Unfortunately, it is well known that such a problem is NP-complete. If we combine this fact with the possible sizes of the networks to be searched (possibly thousands of nodes), it becomes clear, that to solve this problem, we must employ advanced techniques. One of these techniques, which is described and used in this thesis, is the color coding technique.

Goals of the thesis

The main goal of this thesis is to develop a module which is capable of solving the subgraph isomorphism problem by using color coding technique for networks of large sizes. In practice it means that a suitable approach needs to be surveyed and specifically augmented for our usage. For the best results, the study has to be conducted from both theoretical and implementational points of the view.

Final implementation of the module is to be benchmarked by qualitative and performance-related measures and compared to results of other related work.

Thesis structure

Chapter 1 consists of precise formulation and survey of the subgraph isomorphism problem and analysis of possible algorithmic approaches to solving it in a non-trivial way. We also explain, why the approach using the color coding technique has been chosen for this thesis. Chapter 2 describes tree decomposition and treewidth, which are very important concepts for the used algorithm. As the algorithm requires a computation of a tree decomposition of the searched pattern, we also theoretically describe how is the computation carried out in the implementation. Chapter 3 focuses on the analysis of the main algorithm used to solve the subgraph isomorphism problem and on the description of its undertaken modifications. Chapter 4 addresses the implementational view of the algorithm and describes the choice of libraries used to implement the module. Chapter 5 consists of benchmarks of the final module, discussion of the results, and of its comparison to the related work.

Appendices contain a list of used acronyms, a description of the contents of the enclosed CD and details about the implemented module and its install guide.

Subgraph isomorphism problem

In this chapter we describe the subgraph isomorphism problem and its applications in various fields. We also address possible variants of this problem and survey known approaches to solving either of the variants. Finally, we show a time bound for solving this problem that is based on the treewidth of the searched pattern and describe why the algorithm based on the color coding technique has been chosen for the implementation in this thesis.

In order to precisely formulate problem statement, we also introduce the graph theory notation we use and some basic graph theory terms. It is useful to do so, as the described notation and terms are also to be found throughout the whole thesis.

1.1 Applications of subgraph isomorphism

Firstly, we present the applications of the subgraph isomorphism problem without precisely formulating its problem statement. This serves several purposes, as we want both to show how widely is this problem applicable to the real world problems and also how we arrived to the actual form of the problem statement, on which we base this thesis.

In layman's terms, the subgraph isomorphism problem aims to detect whether a given network pattern is present in some other network. Just from this inexact description it is clear, that there are many areas for which a network pattern detection is a relevant problem. This follows from the fact that almost in every system there are agents who interact with each other. These interactions naturally form a network containing information about the system itself. Some of the domains for which the subgraph isomorphism is applicable are:

- Biological systems – In biological systems there is a large number of interactions on a molecular level, e.g., protein-protein interaction. To properly understand such a system, it is mostly required to identify

topological similarities, from which it is possible to understand the underlying mechanisms. Detailed information about the applications of the subgraph isomorphism problem in this domain can be found in [6].

- Social networks – Human interaction can be represented by social networks. Especially nowadays, there is a lot of information about human interaction gained from the the internet. In these human interaction networks, it is, as in the previous case, possible to obtain information from the contained network patterns. Most notably, the ability to search for patterns in social networks can be used in recommendation engines.
- Fraud detection – Any type of fraud, let it be e-mail fraud, bussiness fraud, or possibly even a terrorist attack, admits particular types of patterns on many levels. For example, fraud e-mails are likely to be sent from hubs specialized for sending fraud e-mails. By elaborating a network of computers, we might detect these hubs, as there are typical patterns of computer networks specialized for sending such content.

In general, the applicability of the subgraph isomorphism follows from the fact, that any network can be represented as a graph. This fact gives us a powerful framework to operate with networks, as it is in the case of the subgraph isomorphism problem.

From the described applications, we can see that real world graphs might consist of a very large number of vertices. Because of that, to be able to solve the subgraph isomorphism problem for a real-world application, we need to devise a method, which is able to handle even very large input graphs.

We now describe variants of the subgraph isomorphism problem, together with graph theory needed to precisely elaborate the problem.

1.2 Graph theory notation

Basic graph notation that we use in this thesis contains the following terms:

- $V(G)$: Set of vertices of a graph G . Moreover, we denote $|V(G)|$ by n_G .
- $E(G)$: Set of edges of a graph G . Moreover, we denote $|E(G)|$ by m_G .
- $\deg_G(x)$: Degree of a vertex x in a graph G .
- $d_G(x, y)$: Distance between vertices x and y in a graph G .
- $\text{exc}_G(x)$: Eccentricity of a vertex x in a graph G .
- $N_G(x)$: Set of neighbors of a vertex x in a graph G .
- $N_G[x]$: Set of neighbors of a vertex x in a graph G and x itself.

- $G[A]$: Subgraph of graph G induced by A , where $A \subseteq V(G)$.

All graphs considered in this thesis are undirected and do not contain self loops or multiple edges. We also suppose that every such a graph consists of a single connected component.

1.3 Subgraph isomorphism problem statement and its variations

Before stating the problem, we need to properly define what a subgraph and an induced subgraph is and describe differences between them.

1.3.1 Graph definitions related to subgraph isomorphism

Definition 1.1 *Graph G' is a subgraph of a graph G if $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$.*

Definition 1.2 *Graph G' is a subgraph of a graph G induced by $A \subseteq V(G)$, if G' is a subgraph of G , $V(G') = A$, and there is an edge $e \in E(G')$ between a pair of vertices $x, y \in V(G')$ if and only if there is an edge $f \in E(G)$ between x and y .*

We would like to emphasize the difference between induced and non-induced subgraphs, because it is heavily noticeable in the context of the subgraph isomorphism problem. A simple example of this difference is illustrated in Figure 1.1. We shall discuss the difference more in detail in the further sections.

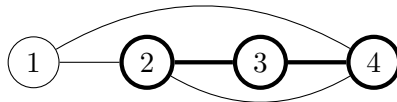


Figure 1.1: Difference between induced and non-induced subgraph. Marked graph (which is a path of length 3, shortly \mathcal{P}_3) is indeed a subgraph of the whole graph in the figure, but it isn't a subgraph induced by set of vertices $\{2, 3, 4\}$, because it is short of one edge $(2, 4)$.

The most important definition for our problem is the definition of graph isomorphism. An example showing that the planar embedding of two isomorphic graphs can indeed look very differently is present in Figure 1.2.

Definition 1.3 *Graphs G and G' are isomorphic if there exists a bijective mapping $\varphi: V(G) \mapsto V(G')$ for which the following condition holds for all $x, y \in V(G)$: $(x, y) \in E(G) \Leftrightarrow (\varphi(x), \varphi(y)) \in E(G')$.*

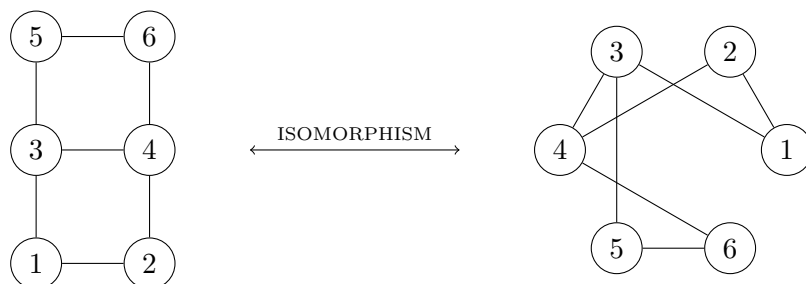


Figure 1.2: Example of two isomorphic graphs.

1.3.2 Subgraph isomorphism problem variations

Having shown all of the needed basic definitions, we can proceed to precisely formulate the subgraph isomorphism problem as a decision problem.

Problem statement SUBISO – Subgraph isomorphism problem

- **Input:** Two graphs G and F .
- **Output:** Decide whether there is a subgraph of G isomorphic to F .

This problem is NP-complete, which can be shown by a polynomial many-one reduction of the k -CLIQUE problem to the SUBISO problem. Such a reduction is sufficient, because it is well known that the k -CLIQUE is NP-complete problem. First, let us formally specify the k -CLIQUE problem:

Problem statement k -CLIQUE – Finding a clique of size k

- **Input:** Graph G and $k \in \mathbb{N}$
- **Output:** Decide whether graph G contains a complete graph \mathcal{K}_k on k vertices as a subgraph

Theorem 1.1 SUBISO is NP-complete.

Proof Clearly, SUBISO \in NP, because given a candidate solution as a certificate we can easily determine in polynomial time whether the solution is correct or not. To perform the reduction k -CLIQUE \leq_m SUBISO, for input G and k in the k -CLIQUE problem, it suffices to create a complete graph \mathcal{K}_k on k vertices. The result of the reduction is an instance of the SUBISO problem in the form of G and \mathcal{K}_k . \square

Naturally, another variant of the subgraph isomorphism problem is its restriction to induced subgraphs.

Problem statement SUBISOIND – Induced subgraph isomorphism problem

- **Input:** Two graphs G and F .
- **Output:** Decide whether there is an induced subgraph of G isomorphic to F .

So far we have only discussed decision variants of the subgraph isomorphism problem. A result of such a problem is already valuable and provides information about the input graph. But to gain even more information, we need to extend the problem statement. In particular, we would like to not only decide a presence of an isomorphic subgraph, but also, e.g., count the number of such subgraphs. From this follows a counting variant of the subgraph isomorphism problem.

Problem statement SUBISOCOUNT – Counting isomorphic subgraphs

- **Input:** Two graphs G and F .
- **Output:** Count the number of subgraphs of G isomorphic to F .

It comes as no surprise that the SUBISOCOUNT problem is at least as hard as the SUBISO problem. That is because an answer to this problem contains more information than an answer to the SUBISO problem, where any nonzero number of counted isomorphic subgraphs during the solution of the SUBISOCOUNT problem directly implies a positive answer to the the SUBISO problem.

To widen our knowledge about the input graphs even further, we need to opt to solve a possibly even harder problem. In such a problem, on top of counting, we also require an enumeration of the result subgraphs as the output of the problem. It is an enumeration variant of the subgraph isomorphism problem.

Problem statement SUBISOENUM – Enumeration of isomorphic subgraphs

- **Input:** Two graphs G and F .
- **Output:** Enumerate subgraphs of G isomorphic to F .

Again, we easily see that a solution to the SUBISOCOUNT can be easily deduced from a solution to the SUBISOENUM variant.

1.4 Approaches to solving the subgraph isomorphism problem

To solve the subgraph isomorphism problem, we could, of course, proceed in a naive way. That however consists of the enumeration of all possibilities, which becomes unfeasible even for graphs of a small size.

The most known approaches to the subgraph isomorphism problem are based on the representation of the problem as a searching process. The efficiency of those approaches is based on the pruning of unprofitable paths in the search space. Some of the algorithms that are based on this idea are Ullmann's algorithm [17], VF algorithm [7], or its successor VF2 algorithm [8]. The largest drawback of this style of approach is the fact that the algorithms are based on backtracking. Therefore, there exist some instances of the subgraph isomorphism problem, for which the algorithm solves the problem no better than the naive one.

A special type of algorithm is the Nauty algorithm [14], which firstly transforms the searched graph into its canonical form. However, as mentioned in [8], there also exist instances of the subgraph isomorphism problem for which it behaves exponentially.

Another family of approaches incorporates a technique called color coding. Its 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. This approach is described in, e.g., [1], [19] or [11].

The algorithm for subgraph isomorphism problem described in [1] makes use of tree decompositions and its complexity is thus related to treewidth (defined in Chapter 2). There is a result proposed by [13], which links the bound on the time needed to solve the subgraph isomorphism problem with ETH (Exponential Time Hypothesis). The time bound is also related to treewidth, which makes [1] a very reasonable candidate to be used for solving the subgraph isomorphism problem. The bound is shown for a special type of the subgraphs isomorphism problem, which is the partitioned subgraph isomorphism problem.

Theorem 1.2 (Dániel Marx [13]) *If there is a recursively enumerable class \mathcal{G} 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 Isomorphism with the smaller graph F in \mathcal{G} in time $f(F)n_G^{o\left(\frac{\text{tw}(F)}{\log \text{tw}(F)}\right)}$, then ETH fails.*

1.5 Chosen problem variant and approach to it

With the knowledge of all variants of the subgraph isomorphism problem and approaches to solve it, we can now specify the problem that we will tackle in

this thesis and the algorithm we will use to solve it.

In Section 1.3, we defined both induced and non-induced variants of the problem. There are many reasons to solve the non-induced variant. For example, the recovery of input graphs from biological systems is indeed not errorless. If we solve only the induced variant of the problem, we might miss an important subgraph just because an information about a single connection in the network was lost during the recovery of the input graph.

Also, based on the discussed applications, we would like to solve the hardest version of the problem – the SUBISOENUM problem, as it allows us to recover the most information from a network.

From the last section we can see, that the color coding algorithm from [1] is theoretically very suitable to solve the SUBISO problem. We will thus use it as a base element of our approach and properly extend it to solve the SUBISOENUM problem.

It is clear that the subgraph isomorphism problem is solved mainly on graphs of a very large size. As the chosen algorithm is exponential in the size of the searched pattern (as later shown in its description in Chapter 3), we must make a compromise. Because of that, we will restrict the domain of graph on which we will solve the problem to pattern graphs of a very small size, i.e., at most 20 vertices.

Tree decomposition and treewidth

This chapter consists of description of treewidth related terminology and problems. We examine this domain, because as we've shown in the previous chapter, the subgraph isomorphic problem complexity relates to the treewidth of the searched pattern. For the chosen algorithm based on the color coding technique, we also propose a suitable way to compute so-called nice tree decomposition of the searched pattern. It turns out that it is a non-trivial task that requires advanced graph theory concepts.

2.1 Definitions related to tree decomposition

For the beginning we need to define what a tree decomposition is.

Definition 2.1 *A tree decomposition of a graph G is a pair (T, β) , where T is a rooted tree and β is a mapping $\beta: V(T) \mapsto 2^{V(G)}$ and for which the following conditions hold:*

- (i) $\bigcup_{x \in V(T)} \beta(x) = V(G)$;
- (ii) for all $(u, v) \in E(G)$ there is $x \in V(T)$, such that $u, v \in \beta(x)$;
- (iii) for all $u \in V(G)$ the nodes $\{x \in V(T) \mid u \in \beta(x)\}$ form a connected subtree of T .

Informally, we could describe a tree decomposition of graph G as a rooted tree of nodes, each of which contains a *bag* of some vertices from G . Contents of such bags need to meet requirements stated above. We shall denote bag $\beta(x)$ as \mathcal{V}_x . Illustration of a tree decomposition of a particular graph can be seen in Figure 2.1.

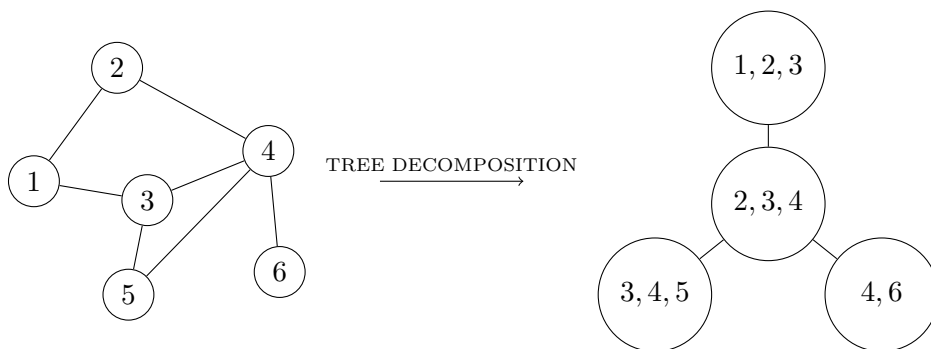


Figure 2.1: Example of a tree decomposition of a graph.

We can see that the tree decomposition definition allows the existence of multiple tree decompositions of one particular graph. As the matter of fact, we can trivially construct a tree decomposition for every graph by forming a tree of single node with all vertices from the original graph in its bag. It is clear that this approach contributes no new information about the original graph. To distinguish quality of tree decompositions, we introduce a metric called the width of a tree decomposition.

Definition 2.2 *Width of tree decomposition (T, β) equals $\max_{x \in V(T)} |\mathcal{V}_x| - 1$.*

Being able to determine width of a tree decomposition, we can ask, which of the tree decompositions of a particular graph is of the minimal width. We can see that we can no longer use trivial approaches to create a tree decomposition, if we want to minimize its width. To be able to describe such a minimal width, we speak of a treewidth of a particular graph.

Definition 2.3 *Treewidth $\text{TW}(G)$ of graph G equals the minimal width of a tree decomposition of G over all such decompositions.*

Treewidth of a graph can be described as a similarity of the graph to a tree. It is then clear, that such a property carries from a graph to its subgraphs.

Proposition 2.1 *For a graph G and its subgraph G' , $\text{TW}(G') \leq \text{TW}(G)$ holds.*

Proof We can easily see that the relation of being a subgraph implies both $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. It then suffices to declare a tree decomposition of G with minimal width (i.e., treewidth) as a tree decomposition of G' (with some adjustments), as neither of the structural differences in G' can increase the width of such tree decomposition. \square

As shown in Theorem 1.2, the computational complexity of subgraph isomorphism problem is related to a treewidth of the searched pattern. The dependency states that the computational complexity grows with the treewidth of the searched pattern (even in the cases when the dependency is indirect). It is then important to realize that the treewidth of a graph isn't generally known beforehand. Moreover, for a given graph G and an integer k , the problem of determining whether the treewidth of G is at most k , is NP-complete [2]. However, by constructing a tree decomposition of G that has a certain width, we retrieve an upper-bound for the treewidth of G . Therefore, all algorithms that use tree decomposition in their process are not only dependant on the treewidth of the input graph, but also on the quality of used tree decomposition, i.e., its width. This restriction also happens to limit us in our case, as described in Chapter 3 – the complexity of the chosen algorithm directly depends on the treewidth of the searched pattern. Because of that, we need to present an algorithm to construct a tree decomposition for a given graph whose width is as close to the optimal treewidth as possible. We elaborate our options in this issue in the next section.

2.2 Tree decomposition construction

For the discussion about the optimal algorithm for tree decomposition construction, we would like to emphasize, that we seek a practically usable algorithm for a restricted class of graphs. Precisely, as mentioned in Section 1.5, we only consider graphs that have at most 20 vertices.

2.2.1 Construction exponential only in treewidth

Most of the theoretical results about the construction of a tree decomposition rely on a supplied bound for treewidth. We can formalize such construction in a k -TREEWIDTH problem.

Problem statement k -TREEWIDTH

- **Input:** Graph G and $k \in \mathbb{N}$.
- **Output:** Decide whether $\text{TW}(G) \leq k$ and if so, output a tree decomposition of G which has width at most k .

The first result in solving this problem has been shown in [2] and is of the following form.

Theorem 2.1 (Arnborg et al. [2]) *There exists an algorithm which solves k -TREEWIDTH problem for a graph G in $\mathcal{O}(n_G^{k+2})$ time.*

This result has been progressively improved and as of now, the current state-of-the-art algorithm makes use of fixed parameter tractability of the problem. Unfortunately, the algorithm holds a large drawback in terms of practical use, because the function of the parameter k grows very fast.

Theorem 2.2 (Bodlaender [3]) *There exists an FPT-algorithm that solves k -TREewidth problem for a graph G in $\mathcal{O}\left(2^{\mathcal{O}(k^3)}n_G\right)$.*

This approach to the construction of a tree decomposition seemingly contradicts the facts we mentioned in the last section. Most notably, while constructing a tree decomposition, we do not know the minimal possible width (i.e., treewidth of the input graph) beforehand. From the theoretical standpoint of view, this isn't an issue, as mentioned in [5]. That is, because we can iterate over all possible values of the parameter k and for each of its fixed value we can use the algorithm from Theorem 2.2 as a subroutine. Due to the theoretical nature of this iterative algorithm, we deem this approach to the computation of a tree decomposition as practically unusable. This conclusion has been verified in [15] and holds also for our domain of very small graphs, even if we later show a way to find a treewidth of a graph in a plausible time (Algorithm 2.5).

The main problem of this type of construction is the fact that the corresponding algorithms focus on the minimalization of the complexity involving the size of the input graph and are instead exponential in treewidth. Due to our problem domain restriction to a class of graphs with only a small number of vertices, we would like to minimize the complexity contribution for all parameters, but the number of vertices of the input graph. Because of this, we do not use any of the algorithms described so far, but instead we propose a construction based on elimination orderings, which is exponential only in the number of vertices of the input graph.

2.2.2 Construction based on an elimination ordering

To directly construct a tree decomposition of minimal treewidth, we employ a technique based on graph triangulation, as described in [5]. To fully understand this approach, we again need to define some terms.

Definition 2.4 *Graph G is chordal, if every cycle in G of length greater or equal to four has an edge connecting two non-consecutive vertices in the cycle.*

Definition 2.5 *Graph G_t is a triangulation of a graph G , if G_t is a chordal graph obtained from G by adding a possibly nonzero number of edges to G , i.e., $E(G_t) \subseteq E(G)$.*

Definition 2.6 *An elimination ordering of a graph G is a bijective mapping $e: V(G) \mapsto \{1, 2, \dots, n_G\}$.*

Definition 2.7 *Elimination ordering e of graph G is perfect, if for all $u \in V(G)$ the higher numbered vertices adjacent to u , i.e., $\{v \mid (u, v) \in E(G) \wedge e(v) > e(u)\}$, form a clique.*

Now we establish relations between chordal graphs and elimination orderings and show how do these terms relate to tree decompositions. We omit the proof of the theorem, because its results are rather well known and we would have to define many subsequent terms in order to form a rigid proof.

Theorem 2.3 (see [10]; [5]) *For a graph G , the following are equivalent:*

- (i) G is chordal;
- (ii) there exists a perfect elimination ordering of G ;
- (iii) there exists a tree decomposition (T, β) of G , where for each $x \in V(T)$ bag \mathcal{V}_x is a clique in G .

Imagine a situation in which we are given an arbitrary graph G . To make use of part (iii) of Theorem 2.3, we need to describe a way to create a chordal graph G' from G . To achieve that, we also use part (ii) of the theorem and build G' by adding edges to G accordingly to the elimination ordering π . The idea is that we construct G' from G and π in a way such that π becomes a perfect elimination ordering of G' . For the sake of clarity, we denote the resulting graph G' as G_π , to emphasize that it has been created from the elimination ordering π . By equivalence in the theorem, we then can declare G_π as a chordal graph and get information about its tree decomposition from part (iii) of the theorem. Due to the fact that we constructed G_π from G just by adding edges, G is a subgraph of G_π and by Proposition 2.1 we also have information about G .

For the described construction of graph G_π from G and π , we specify an algorithm in Algorithm 2.2 and its subroutine in Algorithm 2.1.

We have to formally prove, what we really achieve by applying Algorithm 2.2 on a particular graph.

Proposition 2.2 *For a graph G and an elimination ordering π , the result G_π of $\text{FILLGRAPH}(G, \pi)$ is a triangulation of G .*

Proof By the specification of the subroutine ADDPERFECTEDGES , we can see that π is a perfect elimination ordering of G_π , because the subroutine merely adds a minimal number of edges needed to satisfy the definition of a perfect elimination ordering. By the equivalence between statements (i) and (ii) in Theorem 2.3, graph G_π is chordal. Because the only difference between G and G_π is in edge sets, G_π is also a triangulation of G . \square

Algorithm 2.1 ADDPERFECTEDGES(G, π, u)

Input: Graph G , elimination ordering π and vertex $u \in V(G)$.**Output:** A graph G' , in which all higher numbered (in π) neighbors of u form a clique.**Procedure:**

1. Set $G' := G$.
 2. For each pair v, w of higher numbered neighbors of u , where v and w are distinct and non-adjacent, add an edge between v and w to G' . Formally, for all $v, w \in V(G)$ where $(u, v) \in E(G) \wedge (u, w) \in E(G) \wedge (v, w) \notin E(G) \wedge v \neq w \wedge \pi(v) > \pi(u) \wedge \pi(w) > \pi(u)$ do $E(G') := E(G') \cup \{(v, w)\}$.
 3. Return G' .
-

Algorithm 2.2 FILLGRAPH(G, π)

Input: Graph G and elimination ordering π .**Output:** A graph G_π for which π is a perfect elimination ordering.**Procedure:**

1. Set $G_\pi := G$.
 2. Let us denote by $\pi^{-1}(i)$ the i -th vertex in elimination ordering π . For all $i \in \{1, 2, \dots, n_G\}$ do $G_\pi := \text{ADDPERFECTEDGES}(G_\pi, \pi, \pi^{-1}(i))$.
 3. Return G_π .
-

We have shown even a little stronger claim – the result G_π of FILLGRAPH(G, π) is not only a chordal graph, but also a triangulation of G . We can use this fact in the next theorem, that describes equivalences between different characterizations of treewidth.

Theorem 2.4 (Bodlaender, Koster [5]) *For a graph G and $k \in \mathbb{N}$, the following are equivalent:*

- (i) G has treewidth at most k ;
- (ii) there exists a triangulation G_t of G , such that the maximal size of any clique in G_t is at most $k + 1$;
- (iii) there exists an elimination ordering π , such that the maximal size of any clique in G_π is at most $k + 1$;
- (iv) there exists an elimination ordering π , such that there is no vertex $u \in V(G)$ that has more than k higher numbered (in π) neighbors in G_π .

As the result, we can see that each elimination ordering π yields a tree decomposition of a certain width. It is also clear, that the optimality of such tree decomposition (in terms of its width) is directly related to the input elimination ordering. We shall address the problem of constructing the optimal elimination ordering further in this chapter. Now, we show how does the process of retrieving a tree decomposition from an elimination ordering look like. It is described in Algorithm 2.4 which uses Algorithm 2.3 as a subroutine.

Algorithm 2.3 VERTEXELIMINATION(G, π, u)

Input: Graph G , elimination ordering π of G and vertex $u \in V(G)$

Output: A graph G' , in which all higher numbered (in π) neighbors of u form a clique and in which u is not contained

Procedure:

1. Let $G' := \text{ADDPERFECTEDGES}(G, \pi, u)$.
 2. Set $V(G') := V(G) \setminus \{u\}$.
 3. Return G' .
-

Proposition 2.3 *For a graph G and an elimination ordering π the result (T, β) obtained from Algorithm 2.4 is a tree decomposition of G . Its width is by one smaller than the size of the maximal clique in G_π and it has exactly n_G nodes.*

Proof Firstly, for a tree decomposition (T, β) , let us denote by T_u a subgraph of T , which contains tree decomposition nodes with vertex u in their bags, i.e., $T_u = T[\{x \in V(T) \mid u \in \mathcal{V}_x\}]$. We prove the first part of this proposition by induction on graph size (vertex-wise) based on the recursive execution of Algorithm 2.4. In both induction cases, we are working with the “first” vertex $u = \pi^{-1}(1)$ in the elimination ordering π .

The base case of induction is for $n_G = 1$. Bag \mathcal{V}_{x_u} of tree decomposition node x_u indeed contains the only vertex in the elimination ordering – u . Also, there are no edges in G and as $n_{T_u} = 1$, T_u forms a connected subtree.

For all other cases, $n_G > 1$. We receive a tree decomposition (T', β') as a result of calling the same algorithm on a graph G' , which we obtained from Algorithm 2.3 and for which $V(G') = V(G) \setminus \{u\}$ holds. We then construct a tree decomposition (T, β) for graph G . By having $\beta(x_u) = N_G[u]$, we in fact construct a new bag \mathcal{V}_{x_u} , which contains vertex u and all of its higher numbered neighbours in π .

We now show that all three conditions for (T, β) to be a tree decomposition hold. By induction, $\bigcup_{y \in V(T')} \mathcal{V}_y = V(G')$, and thus $\bigcup_{y \in V(T)} \mathcal{V}_y = V(G)$. Also, by adding node x_u with bag \mathcal{V}_{x_u} into tree decomposition (T, β) , we have a bag containing all $\{u, w \mid (u, w) \in E(G) \setminus E(G')\}$. By induction, for all edges

Algorithm 2.4 TDFROMEO(G, π)

Input: Graph G and elimination ordering π of G **Output:** A tree decomposition (T, β) corresponding to π **Procedure:**

- If $n_G = 1$:
 1. Let $u := \pi^{-1}(1)$.
 2. Return a tree decomposition (T, β) consisting of a single node x_u which contains only u in its bag, i.e., $V(T) = \{x_u\}$, $E(T) = \emptyset$ and $\beta(x_u) = \{u\}$.
 - In other cases:
 1. Let $u := \pi^{-1}(1)$.
 2. Let $G' := \text{VERTEXELIMINATION}(G, \pi, u)$.
 3. Let $\pi' := \pi|_{V(G') \setminus \{u\}} - 1$ (coordinate-wise).
 4. Let $(T', \beta') := \text{TDFROMEO}(G', \pi')$.
 5. Let v be the lowest numbered (in π) neighbour of u in G , i.e., let $v := \pi^{-1}(i)$, where $i = \min_{(u, \pi^{-1}(j)) \in E(G)} j$.
 6. Return a tree decomposition (T, β) , where $V(T) = V(T') \cup \{x_u\}$, $E(T) = E(T') \cup \{(x_u, x_v)\}$, and where for all $x \in V(T')$ $\beta(x) = \beta'(x)$ and $\beta(x_u) = N_G[u]$.
-

$(a', b') \in E(G')$ there is $y \in V(T')$, such that $a', b' \in \mathcal{V}_y$. Therefore, for all edges $(a, b) \in E(G)$ there is $y \in V(T)$, such that $a, b \in \mathcal{V}_y$. By induction, for all $u' \in V(G')$, $T'_{u'}$ is a connected subtree. During the construction of T , we connect the newly created node x_u with node x_v . Because we apply Algorithm 2.3 on u in G to obtain G' , all neighbours $w \in N_G(u)$ are adjacent to v in G' (v is adjacent trivially) and thus T'_w contains x_v . Because x_u is adjacent to x_v in T , T_w forms a connected subtree, even though $v \in \mathcal{V}_u$.

From the process of construction in Algorithm 2.4, it is clear that we in fact constructed a tree decomposition of G_π , because we imitated the process of adding edges in Algorithm 2.2 by consecutively using Algorithm 2.3. In addition, each bag $V_{x_u}, x_u \in T$ of the tree decomposition contains a clique in G_π . Because we consecutively used Algorithm 2.3 on all vertices in π , during the construction we gradually added all cliques in G_π to bags of the tree decomposition. Width of the resultant tree decomposition of G_π is thus by one smaller than the size of the maximal clique in G_π . We can also easily see, that the resultant number of nodes in the tree decomposition is exactly n_G , as we construct a single node for each vertex in the elimination ordering.

Because G is a subgraph of G_π and $V(G) = V(G_\pi)$, any tree decomposition of G_π is a tree decomposition of G . Tree decomposition obtained from Algorithm 2.4 is thus a tree decomposition of G . \square

2.3 Elimination ordering construction

As shown in the previous section, in order to minimize the width of the tree decomposition obtained from Algorithm 2.4, we need to find an elimination ordering, from which such a construction is possible. To do so, we employ an exact algorithm from [4] that computes treewidth $\text{TW}(G)$ for a particular graph G by constructing an elimination ordering. After the computation, we can simply take the elimination ordering created in the process, and use it in Algorithm 2.4 with graph G to obtain a tree decomposition of G with width equal to $\text{TW}(G)$.

We mentioned earlier in Section 2.1 that the process of determining $\text{TW}(G)$ is NP-complete. Due to that, the chosen exact algorithm requires exponential time and space (relative to the number of vertices of the input graph G) to compute $\text{TW}(G)$. That is perfectly sufficient due to the fact, that we only need to compute treewidth for graphs with at most 20 vertices.

To explain and prove the correctness of the used algorithm, we would need to define many subsequent terms. That is, because the main idea of the algorithm is to work with an elimination ordering π , but to avoid working with the triangulation G_π of the input graph G . This approach is extensively shown and proven in [4] and thus we only define terms and propositions needed to be able to analyze and implement the algorithm correctly.

Definition 2.8 *For an elimination ordering π of G and a vertex $u \in V(G)$, we define $\pi_{<,u}$ as the set of the lower numbered vertices (in π) relatively to u . Formally, $\pi_{<,u} = \{v \in V(G) \mid \pi(v) < \pi(u)\}$.*

Definition 2.9 *For an elimination ordering π of G , a set of vertices $S \subseteq V(G)$ and a vertex $u \in V(G) \setminus S$, we define $Q_G(S, u)$ to be the following set: $Q_G(S, u) = \{v \in V \setminus S \setminus \{u\} \mid \text{there exists a path from } u \text{ to } v \text{ in } G[S \cup \{u\} \cup \{v\}]\}$.*

Proposition 2.4 *The set $Q_G(S, u)$ can be computed in $\mathcal{O}(n_G + m_G)$ time.*

Proof It suffices to try all candidate vertices $v \in V \setminus S \setminus \{u\}$ and for each one check, whether it has a neighbour in the connected component of $G[S \cup \{u\}]$, which contains u . Connected component can be retrieved by a simple depth-first search in $\mathcal{O}(n_G + m_G)$ time. Having computed the connected component, each check can be done in a constant time. We perform the checking for $\mathcal{O}(n_G)$ vertices and the maximal number of checkings performed is equal to the number of edges, which is $\mathcal{O}(m_G)$. The checking phase thus also takes $\mathcal{O}(n_G + m_G)$ time. \square

Definition 2.10 For an elimination ordering π of G and a nonempty set of vertices $S \subseteq V(G)$, we define a function $tw(S) = \min_{\pi} \max_{u \in S} |Q(\pi_{<,u}, u)|$.

Theorem 2.5 (Bodlaender et al. [4]) For a graph G , $\text{TW}(G) = tw(V(G))$.

Theorem 2.6 (Bodlaender et al. [4]) For a graph G and a nonempty set of vertices $S \subseteq V(G)$, we can compute a function $tw(S)$ as follows: $tw(S) = \min_{u \in S} \max\{tw(S \setminus \{u\}), |Q_G(S \setminus \{u\}, u)|\}$.

From the Theorem 2.6 we can directly obtain a dynamic programming algorithm that computes the treewidth for a given graph. We show how to do the computations in a top-down fashion in Algorithm 2.5.

Algorithm 2.5 TREEWIDTH(G, S)

Input: Graph G and subset of its vertices S

Output: $tw(S)$

Procedure:

- If $S = \emptyset$:
 1. Return $-\infty$.
 - In case $tw(S)$ has already been computed:
 1. Return the result directly.
 - In other cases:
 1. Return $\min_{u \in S} \max\{\text{TREEWIDTH}(G, S \setminus \{u\}), |Q(S \setminus \{u\}, u)|\}$.
-

Proposition 2.5 Algorithm 2.5, given a graph G and a subset of its vertices $S = V(G)$, computes $\text{TW}(G)$ in $\mathcal{O}(n_G(n_G + m_G)2^{n_G})$ time and $\mathcal{O}(2^{n_G})$ space.

Proof The algorithm computes $tw(V(G))$, which is by Theorem 2.5 equal to $\text{TW}(G)$. To memoize results of all possible subsets of $V(G)$, the algorithm requires $\mathcal{O}(2^{n_G})$ space. In the worst case, it needs to go through all those subsets. For a particular subset S , we need to compute $|Q(S \setminus \{u\}, u)|$ for all vertices $u \in S$. Asymptotically, there can be at most $\mathcal{O}(n_G)$ vertices in S and the computation of $|Q(S \setminus \{u\}, u)|$ takes, by Proposition 2.4, $\mathcal{O}(n_G + m_G)$ time. \square

By running Algorithm 2.5 on graph G and $V(G)$ as its subset S , we retrieve $\text{TW}(G)$. The question is, how can we use this result to obtain an elimination ordering, from which the construction of a tree decomposition of width $\text{TW}(G)$ is possible (by using Algorithm 2.4). To do so, we can reuse the information

about the results of $tw(S)$ that we obtained during the computation in Algorithm 2.5. After the computation finishes, we have access to the results of all $S \subseteq V(G)$. We can then run a new computation in the same manner as used in Algorithm 2.5, in which we don't actually compute results, but just form an elimination ordering from the already computed results. This approach is described by Algorithm 2.6.

Algorithm 2.6 GETEO(G, S, K)

Input: Graph G , subset of its vertices S and a table K , where for all subsets $S' \subseteq S$, K contains the result of $tw(S')$, that is $K(S') = tw(S')$.

Output: An elimination ordering π , from which a tree decomposition of width $tw(S)$ is constructable.

Procedure:

- If $S = \emptyset$:
 1. Return \emptyset
 - In other cases:
 1. Let $u_m := \min_{u \in S} \{K(S \setminus \{u\})\}$.
 2. Let $\pi' := \text{GETEO}(G, S \setminus \{u_m\}, K)$.
 3. Let $\pi := \pi' + 1$ and extend π by $\pi(u_m) = 1$.
 4. Return π .
-

As a result of this approach, we would like to have guaranteed that Algorithm 2.6, if given optimal results of $tw(S)$ for $S = V(G)$ and its subsets, computes an elimination ordering π , from which a construction of tree decomposition of a minimal width is possible. This behavior is in fact guaranteed, as it is implied from the proof of Theorem 2.6 in [4], which through many additional definitions explains set $Q_G(S, u)$ in a wider context. As we don't include this proof, we also omit the corresponding part of the proof of Proposition 2.6. For our purposes it suffices to explain Algorithm 2.6 as a procedure, which in a reverse direction constructs an elimination ordering π from vertices, whose consecutive elimination results in the minimal clique size in triangulation G_π . That as a result due to the previous theorems minimizes the width of a tree decomposition created from π .

Proposition 2.6 *Algorithm 2.6, given a graph G , a subset of its vertices $S = V(G)$ and results of $tw(S')$ for all subsets $S' \subseteq V(G)$, computes an elimination ordering π , for which the maximal size of any clique in G_π is $\text{TW}(G) + 1$, in $\mathcal{O}(n_G^2)$ time and $\mathcal{O}(2^{n_G})$ space.*

Proof As mentioned above, we omit the proof of the first part of the proposition. Space required by computation depends only on the size of the table with the results of $tw(V(G))$ and its subsets. The space bound is thus $\mathcal{O}(2^{n_G})$. For each subset S visited during the computation, we find the minimal value of $tw(S \setminus u)$, for all $u \in S$. However, we already have the results precomputed in table K . The result retrieval in that case takes only a constant time. Because at each call of the recursive procedure we remove a single vertex from $V(G)$, we call the procedure only n_G times. As each subset of $V(G)$ is of size $\mathcal{O}(n_G)$, the determination of the minimal value in each call takes, due to the constant time retrieval, $\mathcal{O}(n_G)$ time. Therefore, the total time spent by the construction is $\mathcal{O}(n_G^2)$. \square

2.4 Nice tree decomposition construction

Now that we presented a way how to construct a tree decomposition of a graph G with optimal width, we need to construct a so called nice tree decomposition. Such a decomposition is required for our main algorithm (and generally all algorithms that use tree decompositions), because in contrast to regular tree decompositions, their nice counterparts possess a regular structure, which can then be exploited by various approaches. In this section, we define a nice tree decomposition and show how to transform a regular tree decomposition to a nice one.

Definition 2.11 *A nice tree decomposition of a graph G is a triple (T, β, r) , where (T, β) is a tree decomposition rooted at node r and where $\deg_G(r) = 1$ and $\mathcal{V}_r = \emptyset$. Additionally, there are only four possible types of a node $x \in V(T)$ of a nice tree decomposition. For these types following conditions hold:*

- Start node – $\mathcal{V}_x = \emptyset$;
- Introduce node – x has exactly one child y and $\mathcal{V}_x = \mathcal{V}_y \cup \{u\}, u \in \mathcal{V}_y$;
- Forget node – x has exactly one child y and $\mathcal{V}_x = \mathcal{V}_y \setminus \{u\}, u \in \mathcal{V}_y$;
- Join node – x has exactly two children y, z and $\mathcal{V}_x = \mathcal{V}_y = \mathcal{V}_z$.

2.4.1 Transformation from a tree decomposition

It is well known that any tree decomposition can be transformed to a nice one without losing its particular characteristics, such as treewidth or number of nodes. We can formalize this statement in the next theorem.

Theorem 2.7 (Cygan et al. [9]) *Any tree decomposition of a graph G that consists of $\mathcal{O}(n_G)$ nodes with width at most t , can be, in $\mathcal{O}(t^2 n_G)$ time, transformed to a nice tree decomposition of G with $\mathcal{O}(t n_G)$ nodes and width bounded by t .*

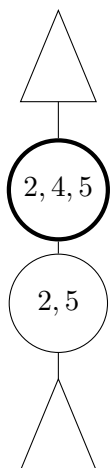


Figure 2.2: Introduce node example.

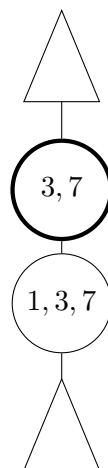


Figure 2.3: Forget node example.

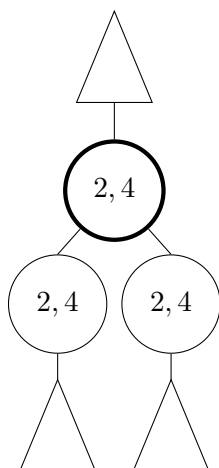


Figure 2.4: Join node example.

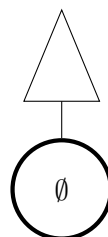


Figure 2.5: Start node example.

Proof We now describe how to handle all possible situations during the construction of a nice tree decomposition from a given tree decomposition. There are exactly three situations we need to describe. Firstly, we need to create a root of the resulting nice tree decomposition, which is a node with an empty bag. That can be easily achieved by adding forget nodes on top of the tree decomposition, as depicted in Figure 2.6. By doing so, we create at most $\mathcal{O}(t)$ new nodes, each of which has a bag of size at most $\mathcal{O}(t)$. The construction can thus be done in $\mathcal{O}(t^2)$ time.

Secondly, we need to enforce the condition which states, that all bags of neighboring nodes differ in at most a single vertex (with the exception of join nodes, which we construct later). To do so, we connect all nodes previously

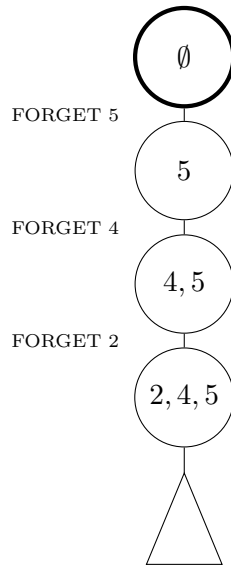


Figure 2.6: Nice tree decomposition root construction.

neighboring in the tree decomposition with a path of new introduce and forget nodes. Each of those nodes extends or reduces the content of a bag by a single vertex and the condition is thus fulfilled. An example of this approach is shown in Figure 2.7. As in the previous case, by reconnecting two nodes we create at most $\mathcal{O}(t)$ new nodes with bag sizes at most $\mathcal{O}(t)$. However this time, we need to reconnect all nodes in the tree decomposition. Because any tree decomposition is a tree, we need to apply this approach at most $\mathcal{O}(n_G)$ times. The construction can thus be done in $\mathcal{O}(t^2 n_G)$ time and adds $\mathcal{O}(t n_G)$ new nodes to the tree decomposition.

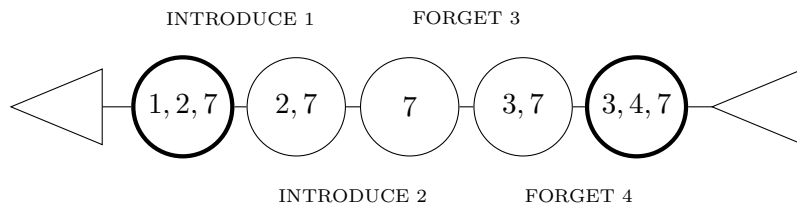


Figure 2.7: Nice tree decomposition node connecting. Marked nodes were previously adjacent and had to be reconnected.

After we perform the described two operations, we already almost have a nice tree decomposition, but there might be nodes with more than one child. We handle this situation by creating a binary tree of join nodes in the place of the original node with a large degree. Its previous neighbors are then connected to the leaves of the constructed binary tree. The construction of such a binary tree is shown in Figure 2.8. For each node with degree $k > 1$, we create a binary tree of k leaves, which as a result consists of $\mathcal{O}(k)$ new nodes. Such a binary tree can be constructed in $\mathcal{O}(tk)$ time. Because any tree decomposition is a tree, the sum of degrees of all its nodes is $\mathcal{O}(n_G)$. From this fact follows that if we apply this construction to all nodes with more than one child, we construct at most $\mathcal{O}(n_G)$ new nodes in $\mathcal{O}(tn_G)$ time.

After applying this construction to any tree decomposition, we indeed obtain another tree decomposition with the same width. In addition, we can see that the constructed tree decomposition is also a nice one. \square

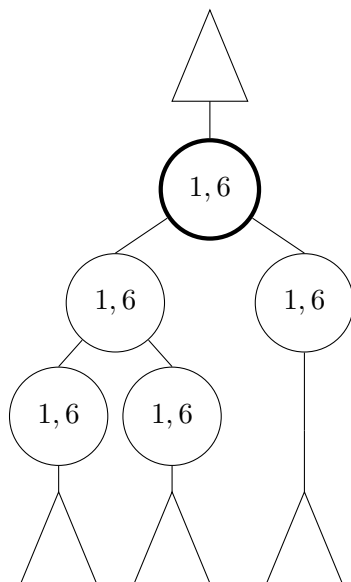


Figure 2.8: Nice tree decomposition degree reduction. The marked node had three children and so it was split into a binary tree of join nodes.

2.4.2 Specific modifications for usage in the algorithm

For the further usage of the computed nice tree decomposition in the main algorithm, we need to modify the standard process of obtaining a nice tree decomposition from a tree decomposition. As we will see throughout Chapter 3, we slightly modify the definition of a nice tree decomposition for the implementation purposes. Specifically, for the next usage of nice tree decompositions, we consider start nodes to have exactly single element in their bag.

This modification is necessary to simplify the color coding algorithm description, implementation and the performance of the final implementation.

Additional modification of the standard nice tree decomposition construction is the precomputation part. From Proposition 2.5 and a description of the time and space complexity needed for the color coding algorithm in Theorem 3.1, we can see that the resources spent on the computation of the nice tree decomposition is insignificant. Also, as we later show in the final version of the color coding algorithm in Algorithm 3.11, the nice tree decomposition is constructed only once per a problem instance. Combination of these factors practically forces us to precompute every single property we can, which is used in the color coding algorithm and which is possible to obtain before running the main algorithm. Although very implementational related, we feel obliged to include the precomputation part in the following descriptions under the name NTDPRECOMPUTE. A perfect example of what can be precomputed are introduce/forgot positions for node mappings (described in Section 3.4.2), which are used in Algorithm 3.8 and Algorithm 3.9.

We can now describe the final form of the algorithm that computes a nice tree decomposition for a given pattern graph, which is then used by the color coding algorithm. The algorithm is described in Algorithm 2.7 and its theoretical properties are described in Theorem 2.8.

Algorithm 2.7 NICETREEDECOMPOSITION(G)

Input: Graph G .

Output: A nice tree decomposition of G .

Procedure:

1. Compute $\text{TREEWIDTH}(G, V(G))$ and store the results for all subsets S of $V(G)$ (i.e., $\text{TREEWIDTH}(G, S)$) in a table K .
 2. Let $\pi := \text{GETEO}(G, V(G), K)$.
 3. Let $(T, \beta) := \text{TDFROMEO}(G, \pi)$.
 4. Construct a nice tree decomposition η from (T, β) as specified in the proof of Theorem 2.7.
 5. Let $\eta' := \text{NTDPRECOMPUTE}(\eta)$.
 6. Return η' .
-

Theorem 2.8 *There is an algorithm, which for a graph G constructs its nice tree decomposition of width $\text{TW}(G)$ in $\mathcal{O}(n_G(n_G + m_G)2^{n_G})$ time and $\mathcal{O}(2^{n_G})$ space.*

Proof Proposition 2.6 states that by running the combination of TREEWIDTH and GETEO procedures (described in Algorithm 2.5 and Algorithm 2.6), we obtain an elimination ordering π , for which the maximal size of any clique in G_π is $\text{TW}(G) + 1$. By Proposition 2.3, we obtain from procedure TD-FRAMEO a tree decomposition of G whose width is by one smaller than the maximal size of any clique in G_π . Therefore, as a result, we obtain a tree decomposition of G whose width is $\text{TW}(G)$. Theorem 2.7 shows a way to create a nice tree decomposition from a tree decomposition while maintaining its width. Procedure NTDPRECOMPUTE is specified in Algorithm 2.7 mainly for implementational purposes and therefore it doesn't change properties of the underlying nice tree decomposition.

By combining time and space bounds of all used procedures, as specified in Proposition 2.3, Proposition 2.5 and Proposition 2.6, we can see that the total time and space required to build a nice tree decomposition are exactly as proposed. \square

Color coding algorithm

In this chapter we describe color coding algorithm used in implementation which solves the SUBISO problem. The algorithm was originally described in [1] and is based on dynamic programming. Due to the algorithm's huge memory requirements (as shown below in Theorem 3.1), we propose modifications that allow us to use this algorithm even on very large input instances. Also, as this particular algorithm does not address the reconstruction of found subgraphs, we also describe a way how to retrieve all subgraphs found during the computation.

For clarity, throughout this chapter we denote by F the pattern graph, i.e. the graph that we are looking to find, and by G the graph in which we search.

3.1 Color coding technique

Color coding technique was first introduced in [1] to solve exactly the problem of subgraph isomorphism. The main idea of this approach is to randomly color the vertices of the graph in which we are looking for subgraphs, and to search only for colorful subgraphs (i.e., subgraphs with all colors assigned to them being distinct). By doing so, we lighten the amount of information needed to be remembered during the algorithm. That is because we do not have to remember all vertex assignments in a particular solution, which is what we do to prevent a multiple inclusion of a vertex to the resulting subgraph. In such a case, it suffices to only remember colors used so far.

The number of colors applied in the coloring should be chosen carefully, as we want subgraphs isomorphic to the searched pattern graph to be colored in a way, that allows us to easily and efficiently locate colorful subgraphs. The chosen way of coloring should also imply that subgraphs are colorful with a high probability.

In our case, for a graph G and a pattern graph F , we color the vertices of G with exactly n_F colors. Formally speaking, we create a random coloring

3. COLOR CODING ALGORITHM

$\zeta: V(G) \mapsto \{1, 2, \dots, n_F\}$. After the coloring, the algorithm considers as a valid result only such subgraphs G' of G , that are colorful copies of F .

Definition 3.1 *Subgraph G' of a graph G is a colorful copy of F with respect to a random coloring $\zeta: V(G) \mapsto \{1, 2, \dots, k\}$, if G' is isomorphic to F and all of its vertices are colored by distinct colors in ζ .*

To address the probability of any subgraph G' of G with $n_{G'} = n_F$ vertices being colorful, we formulate the next proposition.

Proposition 3.1 *For any given graph G and a random coloring $\zeta: V(G) \mapsto \{1, 2, \dots, k\}$, any subgraph G' of G on $n_{G'} = k$ vertices is colorful with a probability at least e^{-k} .*

Proof There are k^k possible colorings of G' . All vertices of G' are of different color in ζ in $k!$ cases. Therefore, the probability P_c of G' being colorful is:

$$P_c = \frac{k!}{k^k}$$

By Stirling's approximation, $k! \approx \sqrt{2\pi k} \left(\frac{k}{e}\right)^k$ and so the following holds:

$$P_c \approx \frac{k^k \sqrt{2\pi k}}{k^k e^k} = \frac{\sqrt{2\pi k}}{e^k} > \frac{1}{e^k} = e^{-k}$$

□

It is clear that if an algorithm is based on random coloring of the input graph, the results it produces are also heavily dependant on the chosen coloring. Even though it is possible to derandomize such algorithms, e.g., by the approach shown in [9], the described method for derandomization is impractical due to the complexity of its implementation. In conclusion, we are to use the randomized version of this algorithm. We discuss the impact of the randomness of the algorithm in the following sections, as we are yet to describe the algorithm itself.

3.2 Main algorithm idea

The algorithm is based on the dynamic programming approach. We apply this approach on the nice tree decomposition of the pattern graph F . The result of the dynamic programming part is in the form of a dynamic programming table \mathcal{D}_t , filled accordingly to a particular function \mathcal{D} . The definition of function \mathcal{D} incorporates the main idea of the algorithm, which connects the color coding approach with the gradual building of the result mapping. We formally define the function \mathcal{D} below.

The original approach differs to our modified approach in the dynamic programming part. The original approach handles the dynamic programming in a top-down fashion, in contrast to our modified algorithm, which constructs the table bottom-up. As noted previously, the way of reconstructing found results from the formed dynamic programming table has not (to our knowledge) been described, because this particular algorithm is mainly devised to solve the original decision SUBISO problem. We thus omit the description of the reconstruction for the original version of the algorithm, as we implemented only the modified variant of the algorithm.

If we break the algorithm step by step, we obtain the following high-level description:

1. Obtain a nice tree decomposition (T, β, r) of graph F by applying Algorithm 2.7;
2. create a random coloring $\zeta: V(G) \mapsto \{1, 2, \dots, n_F\}$;
3. accordingly to function \mathcal{D} construct a dynamic programming table \mathcal{D}_t , with respect to the nice tree decomposition (T, β, r) and the coloring ζ ;
4. reconstruct the results from \mathcal{D}_t .

We now formally describe the dynamic programming part of the algorithm. The target is to create a graph isomorphism $\Phi: V(F) \mapsto V(G)$. We do so by traversing the nice tree decomposition (T, β, r) of the pattern graph F and at each tree decomposition node $x \in V(T)$, we progressively construct possible partial mappings $\varphi: \mathcal{V}_x \mapsto V(G)$ with regard to required colorfulness of vertices of the result subgraph. Such a partial mapping can be viewed as an assignment of vertices contained in \mathcal{V}_x to a subset of vertices in G . If we manage to build altogether consistent partial mappings in all nodes of the tree decomposition, we can easily see that a combination of those partial mappings forms a desired result mapping. To describe a partially constructed isomorphism in the algorithm (i.e., isomorphism on the subgraph of F induced only by its vertices processed so far), we introduce a new notation \mathcal{V}_x^* .

Definition 3.2 *For a nice 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 $\mathcal{V}_x^* = \mathcal{V}_x \cup \{u \in \mathcal{V}_y \mid y \text{ is a descendant of } x \text{ in } T\}$.*

From Definition 3.2 it follows, that for the root r of T , $\mathcal{V}_r^* = V(F)$ and $F[\mathcal{V}_r^*] = F$.

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

3. COLOR CODING ALGORITHM

- (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 $\zeta(\Phi(\mathcal{V}_x^*)) = C$.

In other cases, we define $\mathcal{D}(x, \varphi, C) = 0$, which denotes that there is no such isomorphism to be found with respect to the particular configuration. In further text we denote all configurations (x, φ, C) for which $\mathcal{D}(x, \varphi, C) = 1$ as *nonzero* configurations.

3.3 Original top-down algorithm

The original version of the algorithm is based on top-down dynamic programming approach. Such an approach merely follows the recursive definition of the dynamic programming with an addition of 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. Such a table has three dimensions; each describing one parameter in a configuration $\mathcal{D}(x, \varphi, C)$. Visualization of a dynamic programming table for the top-down approach is depicted in Figure 3.1.

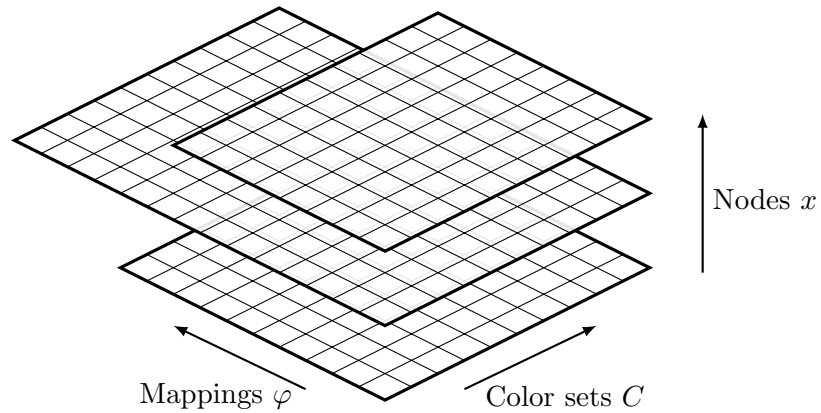


Figure 3.1: Top-down dynamic programming table. Note that the number of possible mappings varies according to the size of a bag of a nice tree decomposition node, while the number of possible color sets stays the same for all nodes.

We now describe a way to compute the result value of a particular configuration $\mathcal{D}(x, \varphi, C)$ in the dynamic programming table, for all four possible types of a nice tree decomposition node x .

For a *start* node $x \in V(T)$, there is only a single vertex u in \mathcal{V}_x^* to consider. We can thus map u to all possible vertices of G , by which we obtain

$\mathcal{D}(x, \varphi, \{\zeta(\varphi(u))\}) = 1$ for all such mappings, and $\mathcal{D}(x, \varphi, C) = 0$ for all other configurations.

For an *introduce* node $x \in V(T)$ and its child y in T , we denote by u the vertex being introduced in x , i.e., $\{u\} = \mathcal{V}_x \setminus \mathcal{V}_y$. It is clear that we set $\mathcal{D}(x, \varphi, C) = 1$ only if there is a configuration $\mathcal{D}(y, \varphi', C') = 1$, where φ' and C' are of a particular form. First of all, we require that φ is an edge consistent mapping. In this case a mapping φ is edge consistent if and only if for all edges $(v, w) \in E(F)$ between mapped vertices $v, w \in \mathcal{V}_x$, there is an edge $(\varphi(v), \varphi(w)) \in E(G)$. To form a nonzero configuration, φ must also be an extension of φ' , i.e., $\varphi = \varphi'|_{\mathcal{V}_y}$. For color sets we require that C' differs from C exactly in the color used to extend mapping φ' by vertex u , i.e., $C' = C \setminus \{\zeta(\varphi(u))\}$. In any other case we set $\mathcal{D}(x, \varphi, C) = 0$.

For a *forget* node $x \in V(T)$ and its child y in T , situation is similar to the previous case. Only this time, a vertex has been forgotten instead of introduced. We denote this vertex by u , as the vertex being forgotten in x , i.e., $\{u\} = \mathcal{V}_y \setminus \mathcal{V}_x$. Again, we set $\mathcal{D}(x, \varphi, C) = 1$ only if we find a configuration $\mathcal{D}(y, \varphi', C') = 1$ with suitable φ' and C' . In this case, for an edge consistent mapping φ , we require that φ' is an extension of φ , i.e., $\varphi = \varphi'|_{\mathcal{V}_x}$. No new color is added to color sets a forget node, and we thus look only at $C' = C$. In any other case we set $\mathcal{D}(x, \varphi, C) = 0$.

For a *join* node $x \in V(T)$, we denote by y and w its children in T . We set $\mathcal{D}(x, \varphi, C) = 1$ only if we find configurations $\mathcal{D}(y, \varphi', C') = 1$ and $\mathcal{D}(w, \varphi'', C'') = 1$ with suitable φ', φ'' and C', C'' . Since $\mathcal{V}_x = \mathcal{V}_y \cup \mathcal{V}_w$, it must hold that $\varphi' = \varphi'' = \varphi$. The only concern in this case are the colors in C' and C'' that were used to build particular isomorphic subgraphs. It is clear, that the union of sets C' and C'' must result in C and also the intersection of both those sets must contain exactly the colors to color the vertices in \mathcal{V}_x . Formally, it must hold that $C' \cup C'' = C$ and $C' \cap C'' = \{\zeta(\varphi(\mathcal{V}_x))\}$. The former condition is implied by the construction, while a failure to comply with the latter condition would mean that either one color would be used twice in the resulting subgraph (in the case when the intersection contains more colors), or there would be not enough colors to color the vertices in \mathcal{V}_x (in the other case). If any of these conditions is not fulfilled, we set $\mathcal{D}(x, \varphi, C) = 0$.

From the description we can see, that the steps taken in all four types of nodes correspond to the specification of a nonzero configuration $\mathcal{D}(x, \varphi, C)$. Therefore, it remains to analyze the time and space required to run this algorithm.

Theorem 3.1 *There is an algorithm that solves the SUBISO problem for an input graph G and a pattern graph F with probability at least $(1 - \frac{1}{e})$ in $\mathcal{O}\left(n_G^{\text{TW}(F)+1} 2^{\mathcal{O}(n_F)}\right)$ time and $\mathcal{O}\left(n_G^{\text{TW}(F)+1} \text{TW}(F) n_F 2^{n_F}\right)$ space.*

Proof At each node of the tree decomposition, there are $n_G^{\text{TW}(F)+1}$ possible

3. COLOR CODING ALGORITHM

Table 3.1: Memory required by the top-down algorithm for $\text{TW}(F) = 1$.

| n_G | n_F | Required memory [B] |
|--------|-------|---------------------|
| 100 | 5 | $1.6 \cdot 10^6$ |
| 100 | 20 | $2.1 \cdot 10^{11}$ |
| 10000 | 5 | $1.6 \cdot 10^{10}$ |
| 100000 | 5 | $1.6 \cdot 10^{12}$ |
| 100000 | 20 | $2.1 \cdot 10^{17}$ |

mappings and 2^{n_F} possible color sets. Because of the Theorem 2.7 and Proposition 2.3, there are $\mathcal{O}(\text{TW}(F)n_F)$ vertices in a nice tree decomposition of F that we have. From these facts we can see that the bound for used space holds.

By Proposition 3.1, the probability of the algorithm answering correctly is e^{-n_F} . The probability of an incorrent answer is thus $(1 - e^{-n_F})$. If we repeat the algorithm e^{n_F} times, the probability of an incorrent answer is $(1 - e^{-n_F})^{e^{n_F}}$. Because the inequality $1 + x \leq e^x$ holds, we can write the probability of an incorrect answer as:

$$(1 - e^{-n_F})^{e^{n_F}} \leq \left(e^{-e^{-n_F}}\right)^{e^{n_F}} = e^{-1} = \frac{1}{e}$$

From the Theorem 2.8 we can see, that the time and space required to create a nice tree decomposition is, although exponential, irrelevant the in comparison to the time required to run the color coding algorithm. In addition, we construct the nice tree decomposition only once, while we repeat the algorithm e^{n_F} times. \square

We have shown, that to solve the SUBISO problem by color coding in a top-down fashion, we require a large amount of memory which can be by no means reduced. That is, because the dynamic programming table needs to contain the results of all configurations (whether zero, nonzero, or unset). To demonstrate how large the memory requirements are, lets assume the result of a configuration $\mathcal{D}(x, \varphi, C)$ requires 1 bit of memory in the table \mathcal{D}_t . Tables 3.1 and 3.2 show the required amount of memory for several combinations of sizes of input graphs G , F and $\text{TW}(F)$.

Lets suppose we have a computer with an above-standard memory available with the value of 100 GB. That is equivalent to 10^{11} B of memory. It is clear from the above tables, that for a slightly larger input and pattern graphs, we do not have enough memory to handle this way of computation. We might have enough memory for some larger input sizes for pattern graphs of treewidth equivalent to 1, but such pattern graphs only cover the family of trees.

Table 3.2: Memory required by the top-down algorithm for $\text{TW}(F) = 2$.

| n_G | n_F | Required memory [B] |
|--------|-------|---------------------|
| 100 | 5 | $3.2 \cdot 10^8$ |
| 100 | 20 | $4.2 \cdot 10^{13}$ |
| 10000 | 5 | $3.2 \cdot 10^{14}$ |
| 100000 | 5 | $3.2 \cdot 10^{17}$ |
| 100000 | 20 | $4.2 \cdot 10^{22}$ |

Another big issue is that the fast growth of the complexity affects not only the space complexity, but also the time required for the computation. We perceive time complexity less critical than the space complexity, because with large time requirements, the algorithm can still be run. On the other hand, the space complexity and the parameters of the computer can restrict, whether we are even able to run the algorithm.

As a conclusion, the first priority is to modify the original algorithm in a way, that it uses less space for the computation. However, the time complexity of the modified algorithm should not be neglected either. We describe the modifications to the original algorithm in the next section.

3.4 Modified bottom-up algorithm

We can divide proposed modifications of the original algorithm into several parts. Firstly, we introduce the key change of the approach as we describe how to handle the dynamic programming computation in a bottom-up fashion. Secondly, we focus on the precise minimization of the memory and time required by the algorithm, by carefully altering and optimizing some of the performed operations. As a last part, we describe how to reconstruct found subgraphs from the dynamic programming table used to solve the decision variant of the problem.

It should be stated that neither of these modifications improves the complexities in the worst case situation, but as we later show in Chapter 5, the proposed modifications greatly reduce time and space required to run the modified version of the algorithm.

3.4.1 Dynamic programming approach modifications

The main motivation to use a bottom-up dynamic programming approach is derived from the large memory requirements of the top-down approach, that cannot be mitigated in any way. It follows from the fact that in the dynamic programming table, while using top-down approach, we need to store results for all configurations, whether nonzero (and thus potential to form an

3. COLOR CODING ALGORITHM

isomorphic subgraph), or not. By using the bottom-up approach, we fill the table in the reverse direction.

As described in Section 3.3, for a configuration $\mathcal{D}(x, \varphi, C)$ in a nice tree decomposition node x , we previously (in general) tried all possible partial mappings φ' and color subsets C' and checked, whether for a children node y there is a nonzero configuration $\mathcal{D}(y, \varphi', C') = 1$. In this approach, we aim to construct all nonzero configurations of a parent node just from the list of nonzero configurations in its child/children.

For that purpose, we formally divide the dynamic programming table \mathcal{D}_t into lists of nonzero configurations, where each nice tree decomposition node has a list of its own. We basically split the three-dimensional table in the dimension of nodes and instead of storing a two-dimensional table of results of all combinations of partial mappings and color sets for the particular node, we store only the nonzero configurations in lists. Formally, for every node $x \in V(T)$, let us denote by $\mathcal{D}_{t,x}$ a list of all mappings φ with a list of their corresponding color sets C , for which $\mathcal{D}(x, \varphi, C) = 1$. Such a list for a particular node is depicted in Figure 3.2. We can easily see, that the information contained in $\mathcal{D}_{t,x}$ for all $x \in V(T)$ is, in terms of contained information, equivalent to maintaining the whole table \mathcal{D}_t , as all configurations not present in the lists can be considered as configurations with a result equal to zero.

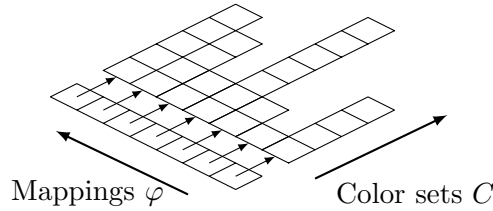


Figure 3.2: Bottom-up dynamic programming list.

Again, as in the previous approach, we describe how to compute the result of a particular configuration $\mathcal{D}(x, \varphi, C)$ for all types of a nice tree decomposition node. In difference to the previous case, this time we describe the computation of a list $\mathcal{D}_{t,x}$ from the knowledge of list/lists of its child/children.

For a *start* 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 so we obtain a list with n_G partial mappings φ , in which the color list for each mapping contains a single color set $\{\zeta(\varphi(u))\}$, where u is the mapped vertex.

For an *introduce* node $x \in T$ its child y in T , we denote by u the vertex being introduced in x , i.e., $\{u\} = \mathcal{V}_x \setminus \mathcal{V}_y$. For all nonzero combinations of a partial mapping and a color set in $\mathcal{D}_{t,y}$, i.e., for all $\mathcal{D}_{t,y}(\varphi', C')$ in the list, 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 two conditions are fulfilled. Firstly, we must guarantee φ

Algorithm 3.1 SUBISOBOTTOMUP(G, F, ζ, η)

Input: Graph G , pattern graph F , random coloring ζ , and a nice tree decomposition η of F .

Output: A bottom-up dynamic programming list of the nonzero configurations in the root of η .

Procedure:

Execute the following recursive subroutine SUBISOREC(G, F, ζ, η, r) on the root r of η and return its output. The subroutine shares the input and output with this procedure. The only additional input parameter is $x \in \eta$, which corresponds to the currently processed node of η .

SUBISOREC(G, F, ζ, η, x):

- If x is a leaf node:
 1. Return SUBISOLEAF(G, F, ζ, η, x), as described in Algorithm 3.6.
 - If x is an introduce node:
 1. Let y be the child of x in η .
 2. Let $\mathcal{D}_{t,y} := \text{SUBISOREC}(G, F, \zeta, \eta, y)$.
 3. Return SUBISOINTRODUCE($G, F, \zeta, \eta, x, \mathcal{D}_{t,y}$), as described in Algorithm 3.9.
 - If x is a forget node:
 1. Let y be the child of x in η .
 2. Let $\mathcal{D}_{t,y} := \text{SUBISOREC}(G, F, \zeta, \eta, y)$.
 3. Return SUBISOFORGET($G, F, \zeta, \eta, x, \mathcal{D}_{t,y}$), as described in Algorithm 3.8.
 - Else x is a join node:
 1. Let y, w be the children of x in η .
 2. Let $\mathcal{D}_{t,y} := \text{SUBISOREC}(G, F, \zeta, \eta, y)$.
 3. Let $\mathcal{D}_{t,w} := \text{SUBISOREC}(G, F, \zeta, \eta, w)$.
 4. Return SUBISOJOIN($G, F, \zeta, \eta, x, \mathcal{D}_{t,y}, \mathcal{D}_{t,w}$), as described in Algorithm 3.7.
-

3. COLOR CODING ALGORITHM

to be edge consistent. This condition is fulfilled if and only if 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 $w \in N_{F[\mathcal{V}_x]}(u)$. The second condition is that the new mapping $\varphi(u)$ of the vertex u must extend the previous colorset C' . That is, $C = C' \cup \{\zeta(\varphi(u))\} \neq C'$. After checking those two conditions, we can add (φ, C) to $\mathcal{D}_{t,x}$.

For a *forget* node $x \in V(T)$ and its child y in T , there is not much work to do, as in the bottom-up construction, we directly obtain the parent list of nonzero configurations. We denote by u , the vertex being forgotten in x , i.e., $\{u\} = \mathcal{V}_y \setminus \mathcal{V}_x$. In this case, for all partial mappings φ' in the list $\mathcal{D}_{t,y}$, we create a new mapping φ that excludes the mapping $\varphi'(u)$ for vertex u , i.e., $\varphi = \varphi'|_{\mathcal{V}_x}$. Particular color sets corresponding to φ' are not changed, as they represent colors already used in the construction. After this step, we might need to merge color lists of previously different mappings, as after the removal of the mapping $\varphi'(u)$, they might have become the same mappings.

For a *join* node $x \in V(T)$, we denote by y and w its children in T . We traverse the children lists $\mathcal{D}_{t,y}$ and $\mathcal{D}_{t,w}$ and look for partial mappings φ' and φ'' , for which $\varphi' = \varphi''$ holds. Such mappings are the only ones to potentially form a new nonzero configuration in the parent list, as due to the fact that $\mathcal{V}_x = \mathcal{V}_y = \mathcal{V}_w$, we construct the new partial mapping φ as $\varphi = \varphi' = \varphi''$. However, for each such mapping φ , we must also construct the new list of color sets, which would afterwards be corresponding to the mapping in the parent list. We do that by travelling color lists corresponding to mappings φ' and φ'' in $\mathcal{D}_{t,y}$ and $\mathcal{D}_{t,w}$, respectively, and for particular sets C' and C'' from the color lists of φ' and φ'' construct a new color set $C = C' \cup C''$. Similarly to the the top-down approach, we also check, whether the intersection of C' and C'' contains exactly the colors to color the vertices in \mathcal{V}_x . That is, for mapping φ , we add to $\mathcal{D}_{t,x}$ a color set C , if $C' \cap C'' = \{\zeta(\varphi(\mathcal{V}_x))\}$.

As a last part of the description of the bottom-up approach, we need to describe the way to travel the nice tree decomposition, during which the above described processing of its nodes happens. 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. To formally define the bottom-up approach, the traversal is described in Algorithm 3.1. The algorithm is mainly composed of sub-routines that handle the situations for each of the nice tree decomposition nodes' type. These sub-routines are described in the

next section, as we first need to explain underwent optimizations that are used in them.

From this high-level description of the bottom-up approach, we can see that several efficiency challenges arise. To point some out, we need to somehow merge color lists in forget nodes, or traverse mapping lists in join nodes to check for mappings that are equal. All such operations, if performed inefficiently, could negatively influence the time complexity of the algorithm. Because of that, we address these smaller implementation problems in the next section, where we in detail describe the algorithm.

3.4.2 Complexity optimization modifications

As a first step, we need to establish a way to represent mappings and color sets in our version of the algorithm. It is important to do so, because the subsequent optimizations minimizing the memory requirements are derived from the used representations.

Representation of mappings

For a mapping representation, we suppose the content of all bags of the used nice tree decomposition stay in the same order during the whole algorithm procedure. Such condition can be fulfilled easily and naturally, but the proposed representation relies on it. The reason is that to minimize the memory needed, in a mapping representation we only store the vertices to which the vertices inside a particular bag are mapped to. Because the order of the vertices in a bag doesn't change, 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.

A mapping $\varphi: \mathcal{V}_x \mapsto V(G)$ in a nice tree decomposition node x is then represented as an ordered tuple of $|\mathcal{V}_x|$ vertices from G . An example of a representation of a particular mapping φ can be seen in Figure 3.3.

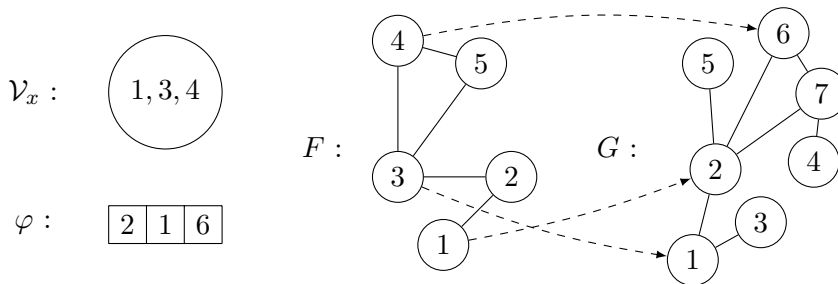


Figure 3.3: Representation of partial mappings.

3. COLOR CODING ALGORITHM

When mappings change size of their tuple during the processing of introduce or forget nodes, for convenience in the description of the algorithm, we allow an operation which concatenates two tuples (or a tuple with a vertex) representing mappings. We denote this operation as `CONCATENATE`. For example, the result of performing `CONCATENATE((1, 7, 2), 6, (4, 5))` is a tuple $(1, 7, 2, 6, 4, 5)$.

Representation of color sets

Color sets can be represented as bitmasks, where an i -th bit states, whether a color i is contained in the set or not. For optimization purposes, we represent bitmasks with an integer number. To retrieve a bitmask from a number, we simply express the number in a binary numeral system. This representation proves as very useful and effective, mostly because in the algorithm we use n_F colors (as described in Section 3.2) and we aim at pattern graphs with at most 20 vertices (see 1.5). In the implementation we can then represent a color set with a 32-bit number.

Compression optimizations

We describe a way to reduce the memory requirements needed to store all of the dynamic programming lists. From the description of the operations in section 3.4.1, we can see that we are never processing information from more than one partial mapping contained in a particular input dynamic programming list. Although we will need to store information about more than one partial mapping (e.g., during the merging of color sets of multiple mappings in forget nodes), we truly process the input list (or lists in join nodes) one mapping at a time. That allows us to store the dynamic programming lists in a compressed way and to decompress it only on a mapping retrieval basis.

For a compression, we would like to serialize dynamic programming lists into a simple buffer of bytes. To do so, we need to design a way how to store and retrieve serialized lists. The way of doing this is straightforward, as we can store a list as a continuous group of records, each of which represents one partial mapping and its corresponding list of color sets. Each record then needs to contain:

- A mapping in the form of ordered tuple of vertices,
- color sets corresponding to the mapping in the form of non-negative integer numbers,
- the number of color sets included.

To be able to deserialize a record, we of course need to store those items in a different order. Particularly, for color sets we first need to know the number of color sets in order to retrieve the correct number of stored information.

The items stored for a single mapping record are thus in order: mapping, the number of color sets, color sets. We do not need to store the number of vertices in a mapping, because we will be accessing dynamic programming lists during the processing of nice tree decomposition nodes. The number of vertices in a mapping can in that case be easily determined by the size of a bag of a particular node. An example of how two mappings from a dynamic programming list are serialized to a buffer is depicted in Figure 3.4.

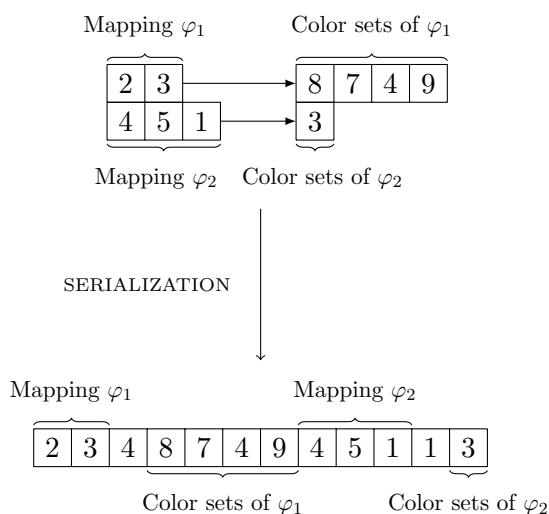


Figure 3.4: Serialization of mappings from a dynamic programming list.

We can now see that a serialized buffer and a dynamic programming list are the two equivalent objects in terms of the contained information. In the following text, we thus don't distinguish between those two terms.

The largest part of the serialized buffer is, in the worst case, formed by color sets corresponding to mappings. In particular, a serialized record of a single mapping consists of two numbers representing lengths, $\text{TW}(F)$ numbers representing a mapping, and at most 2^{n_F} numbers representing particular color sets. However, in difference to the mapping, there isn't any requirement on the order of the color sets. We can thus keep these sets sorted in the increasing order of their number representation, which allows us to use delta compression on them.

The usage of delta compression is in this case very beneficial, because before storing the numbers in the buffer, we aim to encode them with a variable length code. It follows from the fact that the standard integer data types in programming languages consist of 32 bits, which is way too much for our purpose. By additionally using the delta compression on the numbers, we are guaranteed to distinctively decrease the memory needed to store serialized lists of sorted color sets, which is, as argued above, the largest part of the

3. COLOR CODING ALGORITHM

Table 3.3: Bits needed to encode integer numbers by the LibUCW variable length code.

| Range of encoded numbers | Required memory [B] |
|--|---------------------|
| $\langle 0; 128 \rangle$ | 1 |
| $\langle 128; 16512 \rangle$ | 2 |
| $\langle 16512; 2113664 \rangle$ | 3 |
| $\langle 2113664; 270549120 \rangle$ | 4 |
| $\langle 270549120; 34630287488 \rangle$ | 5 |

dynamic programming lists. To make the implementation easier, we use delta compression on all numbers in the record, even though they might not be in a sorted order. This fact doesn't hurt us, because in these cases when we store a large or negative difference (these two are equal in 2's complement), we use the same memory as if we stored the numbers without the variable length code. However, the unsorted part of the record is neglectible in comparison to the possible number of sorted color sets. We measure and discuss how much memory does this approach save in the corresponding Section 5.5.1.

The variable length code we use is a part of the LibUCW library, which was used to implement the algorithm. The library and its functionalities we use are described in the implementation chapter in Section 4.3. The properties of the variable length code from the library are shown in Table 3.3.

We have now described the way of storing and accessing dynamic programming lists. To formally introduce these procedures, we first show how to decode and encode a single number in Algorithm 3.2 and Algorithm 3.3. In these procedures, we don't describe a way how to append or retrieve numbers to the resulting serialized buffer, as this part of the algorithm is purely implementational. We however mention the type of buffer we use in Chapter 4. In Algorithm 3.4 and Algorithm 3.5, we show how to work with a single mapping record in context of dynamic programming lists. To make the description clearer, all of these procedures expect the underlying buffers to persistently remember some information about the history of encoding/decoding.

Mapping order optimization

We derive the next optimization directly from the operations performed in join nodes. For two dynamic programming lists from a join node's children, we need to locate equal partial mappings. Such operation, if performed naively, could lead to a quadratic complexity in terms of the number of mappings present in the lists. On the other hand, have the lists been sorted, it is fairly easy to go through those lists in a two-pointer fashion and locate equal mappings in a time linear in the number of mappings in the lists. Due to this, we are going to process nice tree decomposition nodes in a way that all

Algorithm 3.2 ENCODENUMBER(B, x)

Input: A data buffer B , which stores the last number e_B encoded into it, and an integer number x .

Output: Data buffer B with x in the compressed form appended to it.

Procedure:

1. Let $e_B := 0$ if B is empty.
 2. Set $d := x - e_B$.
 3. Let d_v be d encoded in a variable length encoding, i.e., in our case $d_v := \text{LIBUCWVARINTECODE}(d)$.
 4. Append d_v to B .
 5. Set $e_B := x$.
 6. Return B .
-

Algorithm 3.3 DECODENUMBER(B)

Input: A data buffer B , which stores the last number r_B read from it and which is set to read p -th number encoded in it.

Output: A pair (x, B') , where x is the p -th number decoded from B and B' is B set to read $(p + 1)$ -th number encoded in it..

Procedure:

1. Set $r_B := 0$ if B has not been read yet, i.e., $p = 1$.
 2. Let d be the first number decoded from B in terms of variable length encoding, i.e., in our case $d := \text{LIBUCWVARINTDECODE}(B)$.
 3. Set B to point to the next number stored after x .
 4. Set $x := r_B + d$.
 5. Set $r_B := x$.
 6. Return (x, B) .
-

3. COLOR CODING ALGORITHM

Algorithm 3.4 STORERECORD(B, φ, C_{list})

Input: A data buffer B , a mapping φ and a list of color sets C_{list} .

Output: Data buffer B with the record about φ appended in serialized form to B .

Procedure:

1. For all numbers φ_x in the tuple representing φ , do $B := \text{ENCODENUMBER}(B, \varphi_x)$.
 2. Do $B := \text{ENCODENUMBER}(B, |C_{list}|)$.
 3. For all color sets $C \in C_{list}$, do $B := \text{ENCODENUMBER}(B, C)$.
 4. Return B .
-

Algorithm 3.5 GETRECORD(B, S)

Input: A data buffer B set to read p -th record encoded in it, and a size S of a bag of corresponding nice tree decomposition node.

Output: A tuple $(\varphi', C'_{list}, B')$, where φ' is a mapping and C'_{list} is a list of color sets, both decoded from p -th record in B , and B' is B set to read $(p+1)$ -th record encoded in it.

Procedure:

1. For the count of numbers equal to S , do $(x, B) := \text{DECODENUMBER}(B)$ and store x to the corresponding position in φ' .
 2. Do $(C_{length}, B) := \text{DECODENUMBER}(B)$.
 3. For C_{length} numbers do $(C, B) := \text{DECODENUMBER}(B)$ and append C to C'_{list} .
 4. Return (φ', C'_{list}, B) .
-

dynamic programming lists used throughout the computation remain sorted in the lexicographical order of the contained mappings. As a result, all operations performed in nice tree decomposition nodes can be based on the fact the input children lists are sorted accordingly. The requirement on the order however makes the handling of introduce and forget nodes a bit more complex.

Having formally defined the procedures that work with the dynamic programming lists and the way of ordering of the mapping records, we can now describe the subroutines used in Algorithm 3.1. In particular, we show in Algorithm 3.6 how to process start nodes, and in Algorithm 3.7 how to process join nodes. For introduce and forget nodes, we are yet to describe additional steps that are needed to be taken in their processing, as mentioned above.

Algorithm 3.6 SUBISOLEAF(G, F, ζ, η, x)

Input: Graph G , pattern graph F , random coloring ζ , a nice tree decomposition η of F and a start node $x \in \eta$.

Output: A dynamic programming list $D_{t,x}$ with nonzero configurations after processed in a start node x .

Procedure:

1. Initialize $D_{t,x}$ as an empty dynamic programming list.
2. For all vertices $u \in V(G)$ in increasing order, do $D_{t,x} := \text{STORERECORD}(D_{t,x}, (u), \{\zeta(u)\})$.
3. Return $D_{t,x}$.

The problem in introduce and forget nodes arises from the fact that even though the input dynamic programming list is sorted in a lexicographical order of mappings, by consecutively performing the corresponding operation on mappings in both introduce and forget nodes, we can destroy the original order. For introduce nodes, one would say it suffices to extend the mappings by vertices in order of their number in graph G . That is however untrue, as shown in a counterexample in Figure 3.5.

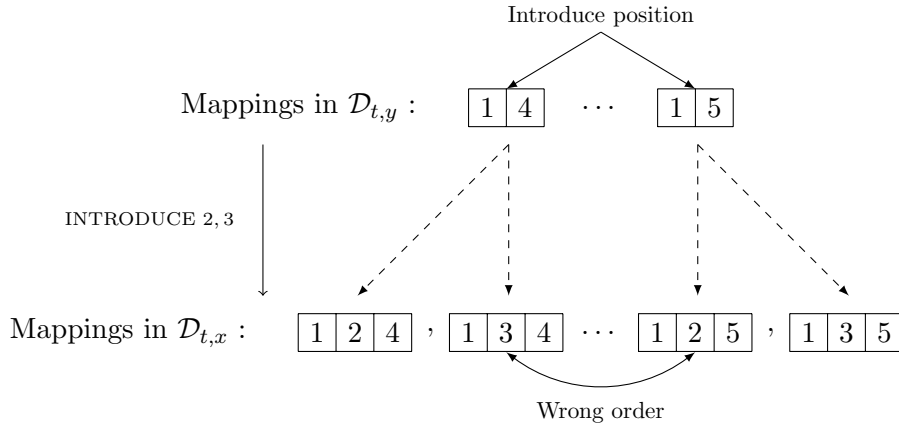


Figure 3.5: Disruption in the order of mappings in an introduce node.

For forget nodes it can be seen fairly easy, that by the removal of a number from an arbitrary position in a mapping tuple, we can destroy the original order. Again, an example proving this can be seen in Figure 3.6.

To address this problem, while working with mappings, we split them into a prefix containing elements of the mapping tuple located before the position to which a new mapping of a vertex is introduced (or the position from which

Algorithm 3.7 SUBISOJOIN($G, F, \zeta, \eta, x, D_{t,y}, D_{t,w}$)

Input: Graph G , pattern graph F , random coloring ζ , a nice tree decomposition η of F , a join node $x \in \eta$, and dynamic programming lists $D_{t,y}$ and $D_{t,w}$.

Output: A dynamic programming list $D_{t,x}$ with nonzero configurations after processed in a join node x .

Procedure:

1. Initialize $D_{t,x}$ as an empty dynamic programming list.
 2. Do $(\varphi', C'_{list}, D_{t,y}) := \text{GETRECORD}(D_{t,y}, |\mathcal{V}_x|)$ and $(\varphi'', C''_{list}, D_{t,w}) := \text{GETRECORD}(D_{t,w}, |\mathcal{V}_x|)$.
 3. If there was an attempt to use GETRECORD on $D_{t,y}$ or $D_{t,w}$ while any of these lists was already completely read, go to step 5.
 4. Distinguish three cases:
 - If φ' is lexicographically smaller than φ'' , then $(\varphi', C'_{list}, D_{t,y}) := \text{GETRECORD}(D_{t,y}, |\mathcal{V}_x|)$ and go to step 3.
 - If φ' is lexicographically greater than φ'' , then $(\varphi'', C''_{list}, D_{t,w}) := \text{GETRECORD}(D_{t,w}, |\mathcal{V}_x|)$ and go to step 3.
 - Else $\varphi' = \varphi''$:
 - (1) Initialize C_{list} as an empty list.
 - (2) For all $C' \in C'_{list}$ and for all $C'' \in C''_{list}$, if $C' \cap C'' = \zeta(\varphi'(\mathcal{V}_x))$, then $C_{list} := C_{list} \cup \{C' \cup C''\}$.
 - (3) Order C_{list} in order of increasing numbers.
 - (4) Do $D_{t,x} := \text{STORERECORD}(D_{t,x}, \varphi', C_{list})$.
 - (5) Go to step 2.
 5. Return $D_{t,x}$.
-

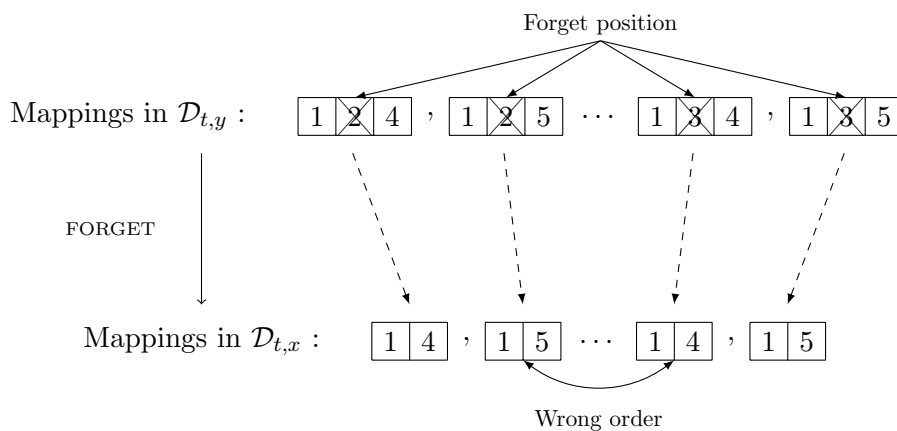


Figure 3.6: Disruption in the order of mappings in a forget node.

a mapping of a vertex is forgotten) and into a suffix of the rest of the elements. Such a division is useful, because it can be seen that possible disruptions in order occur only between mappings with the same prefixes. It follows from the fact that mappings are transformed exactly at the position between their prefix and their suffix, and so the order of mappings' prefixes is unharmed.

Because the mappings we process are ordered, their prefixes are also ordered. To maintain the mappings in the correct order during and after we process them, it then suffices to correctly order suffixes for the currently processed prefix. After we reach a mapping with a different prefix, we can safely append already processed mappings (and their corresponding records) with the same prefix to the resulting dynamic programming list.

Such an approach is good for two reasons. Firstly, for the cases of mappings with a non-empty prefix, we don't have to sort all the mappings contained in a particular dynamic programming list at once, but just the mappings with the same prefix at a time. Secondly, this approach allows us to easily implement the operation needed in forget nodes, in which we need to merge color sets of mappings that have become equal after the processing. Because only mappings with the same prefix can become equal, we just append the color sets of mappings with equal suffixes.

To efficiently sort and access suffixes, for each prefix we use a binary search tree, in which we store a pair consisting of (*suffix*, *color sets list*). After encountering a different prefix, we obtain the correct order of all suffixes corresponding to a previous prefix by an in-order traversal of the particular binary search tree. We describe the sub-routine of Algorithm 3.1, which is used to process forget nodes formally in Algorithm 3.8. To describe the sub-routine for introduce nodes, we still need to explain additional optimizations that are used in it.

Algorithm 3.8 SUBISOFORGET($G, F, \zeta, \eta, x, D_{t,y}$)

Input: Graph G , pattern graph F , random coloring ζ , a nice tree decomposition η of F , a forget node $x \in \eta$, and a dynamic programming list $D_{t,y}$.

Output: A dynamic programming list $D_{t,x}$ with nonzero configurations after processed in a forget node x .

Procedure:

1. Initialize $D_{t,x}$ as an empty dynamic programming list and initialize t as an empty binary search tree of pairs (*suffix, color sets list*), with *suffix* as a key. Tree t is ordered lexicographically with respect to keys.
 2. For each record in $D_{t,y}$, set $(\varphi', C'_{list}, D_{t,y}) := \text{GETRECORD}(D_{t,y}, |\mathcal{V}_x|+1)$ and do the following:
 - (1) Let c be the position in φ' at which the process of forgetting occurs.
 - (2) Let $p(\varphi')$ be the prefix of φ' consisting of elements in tuple φ' before position c and let $s(\varphi')$ be the suffix of φ' consisting of elements in tuple φ' after position c .
 - (3) If $p(\varphi')$ is not the first prefix processed in $D_{t,x}$ and $p(\varphi') \neq p_t$, do:
 - (3.1) Perform an in-order traversal of t and for each stored record (s_t, C_{list}) order C_{list} in order of increasing numbers and do $D_{t,x} := \text{STORERECORD}(D_{t,x}, \text{CONCATENATE}(p_t, s_t), C_{list})$.
 - (3.2) Set t to be empty.
 - (4) Add a record $(s(\varphi'), C'_{list})$ into t . That is, if t contains a record $(s(\varphi'), X)$, set $(s(\varphi'), X) := (s(\varphi'), X \cup C'_{list})$, else insert in t a new record $(s(\varphi'), C'_{list})$.
 - (5) Set $p_t := p(\varphi')$.
 3. For the last unprocessed prefix, do the same as in (4.1).
 4. Return $D_{t,x}$.
-

Mapping expansion optimization #1

The main “brute-force” work of the algorithm is performed in two types of nodes – leaf and introduce. This follows from the algorithm description in 3.4.1, which states that 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 every vertex from G . However, there is a way to reduce the vertices needed to be tried in the case when we are processing an introduce node.

The idea is based on the fact, that we always need to check whether the new mapping of an introduced vertex is edge consistent with the mapping of the remaining vertices for the corresponding bag (as the rest of the bag is always already mapped). To comply with the edge consistency condition, for every edge between vertices in the bag, there must be a corresponding edge in the mapping of the vertices in the bag. Because it suffices to check only the edges between the newly mapped vertex and its neighbors, we can easily determine which vertices of G would be edge consistent without using any brute-force approach. To obtain such candidates for a vertex u that is being introduced to a nice tree decomposition node x , we look at the neighbors of u in $F[\mathcal{V}_x]$. If there are any (i.e., $\deg_{F[\mathcal{V}_x]}(u) > 0$), we look at the vertices they are mapped to. Formally, we look at vertices of G in the set $\varphi(N_{F[\mathcal{V}_x]}(u))$. From the edge consistency condition, it is clear that the only possible candidate vertices in G , on which the vertex u can be mapped, must be adjacent to all vertices in $\varphi(N_{F[\mathcal{V}_x]}(u))$. Note that after performing this selection of mapping candidates, we do not have to check the edge consistency condition for the remaining vertices anymore. The idea is illustrated in Figure 3.7.

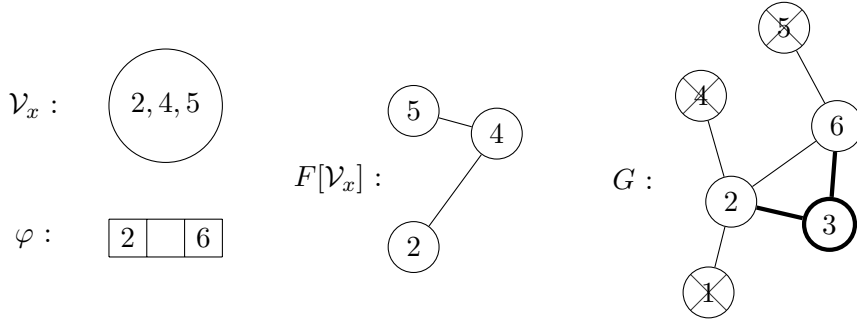


Figure 3.7: Mapping expansion modification based on edge consistency. The vertex labeled 4 in $F[\mathcal{V}_x]$ is being introduced. The only possible vertex to extend the mapping φ is the vertex 3 in G .

In the case $\deg_{F[\mathcal{V}_x]}(u) = 0$ we have to use the regular approach, in which we try all possible nodes of G . Because such case is the most computationally demanding part of the algorithm, in the next section we further optimize it.

Mapping expansion optimization #2

In the case the vertex u being introduced to an introduce node x has no neighbors in $F[\mathcal{V}_x]$, in the worst case we need to try to map u onto all vertices of G . That is unfavorable for us, because as discussed in Section 1.5, we aim to use the algorithm for input graphs G of a possibly large size. On the other hand, the pattern graphs F tend to be smaller than the input graphs G by several orders of magnitude, which is the fact we exploit in this optimization.

For such an introduce node, the partial mappings we work with already contain the mapping of all vertices in \mathcal{V}_x but the introduced vertex u . A partial mapping processed in an introduce node already anchors the possible resulting subgraph to a certain position in G . During the construction of possible mapping of u , it is then pointless to try mapping u to vertices in G that could by no means form a resulting subgraph isomorphic to F . A decision when the mapping is pointless or not can be made due to a possible sheer difference between the position of a candidate vertex on which u is to be mapped and the position of the vertex on which another vertex from \mathcal{V}_x is already mapped. Such a decision is possible because we assume input graphs consist of a single connected component.

We obtain the maximal possible distance to be considered in G by the eccentricity in F of the vertices from \mathcal{V}_x , that are already mapped to vertices of G . If we happen to need to try all vertices of G to obtain a mapping of u (in order to extend a mapping φ), we determine the already mapped vertex w with minimal eccentricity in F , i.e., we determine $\min_{v \in \mathcal{V}_x \setminus \{u\}} \text{exc}_F(v)$. It is then clear, that to construct all possible mappings of u , it suffices to try vertices from G that are located at most $\text{exc}(w)$ units of distance far from $\varphi(w)$. We could in fact restrict the set of candidate vertices for mapping even further, by determining which vertices are in suitable distance from all of the vertices already mapped in φ . That would however be purposeless, as this optimization aims to drastically reduce the number of candidate vertices mainly in the cases when of a graph G with a large number of vertices and a large diameter. The reduction of an already reduced list of vertex candidates is thus insignificant to the total achieved result. The principle of this optimization is illustrated in Figure 3.8.

We can now describe the last sub-routine of Algorithm 3.1, which is used to process introduce nodes. It is described in Algorithm 3.9. By analyzing Algorithm 3.1, we arrive at the following conclusion.

Theorem 3.2 *There is an algorithm that solves the SUBISO problem for an input graph G and a pattern graph F with probability at least $(1 - \frac{1}{e})$ in $\mathcal{O}\left(n_G^{\text{TW}(F)+1} 2^{\mathcal{O}(n_F)} \log R\right)$ time and $\mathcal{O}(R)$ space, where R is the maximal number of nonzero dynamic programming configurations encountered over all runs of its internal dynamic programming procedure. Dynamic programming configurations are part of a state space of size $\mathcal{O}\left(n_G^{\text{TW}(F)+1} \text{TW}(F) n_F 2^{n_F}\right)$.*

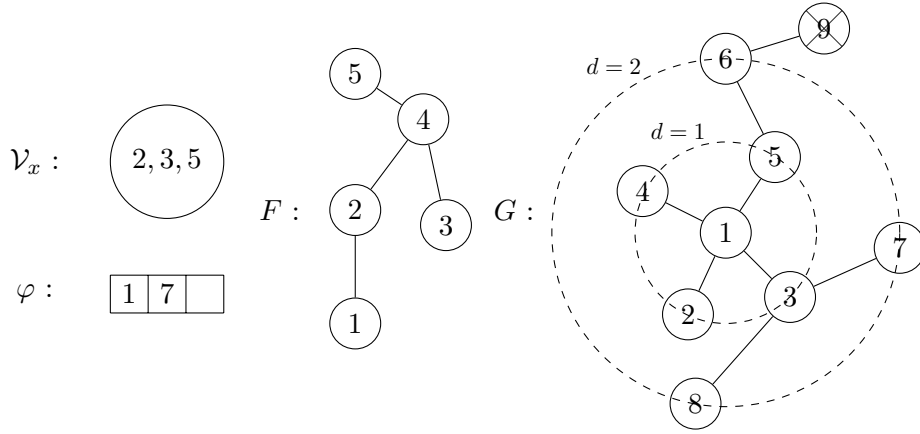


Figure 3.8: Mapping expansion modification based on eccentricity. In the depicted case, we cannot use the optimization based on the edge consistency, as the introduced vertex 5 has no neighbors in $F[\mathcal{V}_x]$. The vertex with the minimal eccentricity in F is the vertex labeled with 2. It can be easily seen, that the introduced vertex can not be mapped to the vertex labeled 9 in G , because its distance d to $\varphi(2)$ (vertex 1 in G) would be inconsistent with the eccentricity of the vertex 2 in F .

Proof In Section 3.4.1 we argued the equivalence between a top down and bottom up approach of the color coding algorithm. In the worst case, the space needed by the modified bottom up algorithm is the same as in the unmodified version. Both versions thus share the same space bound described in Theorem 3.1. However, as we described above, in the bottom up version of the algorithm we only store the nonzero configurations during the dynamic programming part and so the space used is in fact bounded by R .

In the described bottom-up version of the algorithm, at each node of the tree decomposition we in addition store all actual mappings in a binary search tree. Querying of such tree results in a $\mathcal{O}(\log R)$ slowdown in comparison to the original algorithm. All other operations do not modify the bounds proven in Theorem 3.1. \square

3.4.3 Reconstruction of the result

We reconstruct the result in a reverse direction to the bottom-up approach by using the dynamic programming lists computed during the first phase of the computation in Algorithm 3.1. That is, we consecutively, from root to leaves, take nonzero records in dynamic programming lists and as follows from the main idea in Section 3.2, we gradually construct a result mapping Φ from the encountered partial mappings φ .

3. COLOR CODING ALGORITHM

Algorithm 3.9 SUBISOINTRODUCE($G, F, \zeta, \eta, x, D_{t,y}$)

Input: Graph G , pattern graph F , random coloring ζ , a nice tree decomposition η of F , an introduce node $x \in \eta$, and a dynamic programming list $D_{t,y}$.

Output: A dynamic programming list $D_{t,x}$ with viable (nonzero) configurations after processed in an introduce node x .

Procedure:

1. Initialize $D_{t,x}$ as an empty dynamic programming list and initialize t as an empty binary search tree of pairs (*suffix, color sets list*), with *suffix* as a key. Tree t is ordered lexicographically with respect to keys.
 2. For each record in $D_{t,y}$, set $(\varphi', C'_{list}, D_{t,y}) := \text{GETRECORD}(D_{t,y}, |\mathcal{V}_x| - 1)$ and do the following:
 - (1) Let c be the position in φ' at which the process of introducing occurs.
 - (2) Let $p(\varphi')$ be the prefix of φ' consisting of elements in tuple φ' upto position c and let $s(\varphi')$ be the suffix of φ' consisting of elements in tuple φ' after position c .
 - (3) If $p(\varphi')$ is not the first prefix processed in $D_{t,x}$ and $p(\varphi') \neq p_t$, do:
 - (3.1) Perform an in-order traversal of t and for each stored record (s_t, C_{list}) order C_{list} in order of increasing numbers and do $D_{t,x} := \text{STORERECORD}(D_{t,x}, \text{CONCATENATE}(p_t, s_t), C_{list})$.
 - (3.2) Set t to be empty.
 - (4) Let $u \in V(F)$ be the vertex introduced in x .
 - (5) Consider following two cases:
 - If $\deg_{F[\mathcal{V}_x]}(u) > 0$:
 - (5.1) Let $S := \{v \in V(G) \mid v \text{ is adjacent to all } \varphi'(N_{F[\mathcal{V}_x]}(u))\}$.
 - In the other case:
 - (5.1) Let $w := \min_{v \in \mathcal{V}_x \setminus \{u\}} \text{exc}_F(v)$.
 - (5.2) Let $S := \{v \in V(G) \mid d_G(\varphi'(w), v) \leq \text{exc}_F(w)\}$.
 - (6) For all $v \in S$ do:
 - (6.1) Let $C_{list} := \{C \in C'_{list} \mid C \cup \{\zeta(v)\} \neq C\}$.
 - (6.2) If $C_{list} \neq \emptyset$, let $s_I := \text{CONCATENATE}(v, s(\varphi'))$ and add a record (s_I, C_{list}) into t . That is, if t contains a record (s_I, X) , set $(s_I, X) := (s_I, X \cup C_{list})$, else insert in t a new record (s_I, C_{list}) .
 - (7) Set $p_t := p(\varphi')$.
 3. For the last unprocessed prefix, do the same as in (4.1).
 4. Return $D_{t,x}$.
-

To represent result mappings $\Phi: V(F) \mapsto V(G)$, we can use the same tuple structure as in the representation of partial mappings in the record in dynamic programming lists. Only this time, tuples representing result mappings contain n_F elements.

For the reversed traversal, the roles of introduce and forget nodes swap. In particular, as we now traverse from the root of the nice tree decomposition to its leaves, extensions and restrictions of partial mappings occur between introduce and forget nodes in the reverse order. Because in the reconstruction we are only going to extend result mappings step by step, the work will be done in forget nodes, as in the reverse direction a forget node implies an addition of a vertex to partial mappings. In any other type of node we do not have to compute anything, as the result is indeed built from mapping extensions only.

During the reconstruction, the procedure in a node x of a nice tree decomposition η will work with a list \mathcal{R} of result mappings, where any mapping $\Phi \in \mathcal{R}$ contains a partial solution built from the mapping extensions in forget nodes encountered so far on the way from the root of the nice tree decomposition. For any node that is not a forget node, we merely pass \mathcal{R} to all children (if the node is not a start node). For a forget node x , suppose y is a child of x in η . We need to check for all mappings $\varphi \in \mathcal{D}_{t,y}$ and for all $\Phi \in \mathcal{R}$, whether φ was used as a part of the solution encoded in Φ . To do so, it suffices to check some properties of the forgotten node u , $\{u\} = \mathcal{V}_y \setminus \mathcal{V}_x$ for all possible combinations of φ and Φ . Namely we require that the color of $\varphi(u)$ extends the color set of all mapped vertices in Φ . That is, we require $\zeta(\Phi(V(F))) \cup \zeta(\varphi(u)) \neq \zeta(\Phi(V(F)))$. Even though we use $V(F)$ in the condition, it is important to realize that Φ is in this phase defined only for some subset of $V(F)$ and the condition thus can be fulfilled. Also, it is needed for Φ to be an extension of φ in all vertices of \mathcal{V}_y but u (because u is not yet included in Φ). As such a set in fact equals \mathcal{V}_x , it needs to hold $\varphi|_{\mathcal{V}_x} = \Phi|_{\mathcal{V}_x}$. For each valid pair of φ and Φ , we add a new mapping consisting of Φ extended by mapping of u in φ to the output list of solutions for this node.

We formally describe the reconstruction algorithm in Algorithm 3.10 and discuss its properties in Theorem 3.3.

Theorem 3.3 *For a filled dynamic programming table \mathcal{D}_t of the color coding algorithm described in Algorithm 3.1, which solves the SUBISO problem for a graph G and a graph F , there exists an algorithm which reconstruct subgraphs found during the run of the color coding algorithm, i.e., solves the SUBISOENUM problem. If \mathcal{D}_t contains R nonzero records and there are S solutions to the SUBISOENUM problem, then the reconstructing algorithm runs in $\mathcal{O}(SR \cdot \text{TW}(F))$ time and uses $\mathcal{O}(R + S \cdot n_F)$ space.*

Proof Time and space bounds follow directly from the combination of the description of the color coding algorithm in Algorithm 3.1 and the description of algorithm used for reconstruction in Algorithm 3.10. For each of S possible

Algorithm 3.10 RECONSTRUCTRESULTS(G, F, ζ, η)

Input: Graph G , pattern graph F , random coloring ζ , and a nice tree decomposition η of F .

Output: A list of mappings of subgraphs which correspond to nonzero configurations in $\mathcal{D}_{t,x}$.

Procedure:

Execute the following recursive subroutine with parameters RECONSTRUCTREC($G, F, \zeta, \eta, r, \mathcal{R}_0$), where r is the root of η and a list \mathcal{R}_0 contains a single empty result mapping. The subroutine shares the input and output with this procedure. The two additional input parameters are $x \in \eta$, which corresponds to the currently processed node of η and \mathcal{R} which represents the current set of solutions.

RECONSTRUCTREC($G, F, \zeta, \eta, x, \mathcal{R}$):

- If x is a leaf node:
 1. Return \mathcal{R} .
 - If x is an introduce node:
 1. Let y be the child of x in η .
 2. Return RECONSTRUCTREC($G, F, \zeta, \eta, y, \mathcal{R}$).
 - If x is a forget node:
 1. Let y be the child of x in η .
 2. Initialize \mathcal{R}' as an empty list of mappings $V(F) \mapsto V(G)$.
 3. Let $\{u\} := \mathcal{V}_y \setminus \mathcal{V}_x$ (u is the vertex forgotten in x).
 4. For each record in the child dynamic programming list $D_{t,y}$, set $(\varphi, C_{list}, D_{t,y}) := \text{GETRECORD}(D_{t,y}, |\mathcal{V}_y|)$ and do the following:
 - (1) For all $\Phi' \in \mathcal{R}$:
 - If $\zeta(\Phi'(V(F))) \cup \zeta(\varphi(u)) \neq \zeta(\Phi'(V(F)))$ and at the same time $\varphi|_{\mathcal{V}_x} = \Phi'|_{\mathcal{V}_x}$:
 - (1.1) Let $\Phi := \Phi'$ with the extension of $\Phi(u) = \varphi(u)$.
 - (1.2) Set $\mathcal{R}' := \mathcal{R}' \cup \Phi$.
 5. Return RECONSTRUCTREC($G, F, \zeta, \eta, y, \mathcal{R}'$).
 - Else x is a join node:
 1. Let y and w be the children of x in η .
 2. Let $\mathcal{R}' := \text{RECONSTRUCTREC}(G, F, \zeta, \eta, y, \mathcal{R})$.
 3. Return RECONSTRUCTREC($G, F, \zeta, \eta, w, \mathcal{R}'$).
-

solutions, we iterate through R nonzero records and compare the mappings in $\mathcal{O}(\text{TW}(F))$ time. Nonzero records take $\mathcal{O}(R)$ space, and to represent all solutions we require $\mathcal{O}(S \cdot n_F)$ space. \square

3.5 Final form of the algorithm and its properties

Theorem 3.2 states that by running Algorithm 3.1 e^{n_F} times, we can solve the SUBISO problem with the probability at least $\frac{1}{e}$. But what if we would like the algorithm to have a different probability of successfully solving the problem? Formally, we are interested in the number of repetitions needed, so that the probability of the algorithm not founding a subgraphs which should have been found is ε .

Proposition 3.2 *By running the color coding algorithm (described in Algorithm 3.1) $e^{n_F \log \frac{1}{\varepsilon}}$ times each time with a random coloring $\zeta: V(G) \mapsto \{1, 2, \dots, n_F\}$, the probability of the algorithm not founding a proper solution to the SUBISO problem is at most ε .*

Proof The algorithm only detects colorful subgraphs. By Proposition 3.1, the probability of a subgraph being colorful in the case of random coloring with n_F colors is e^{-n_F} . Therefore, the probability of the algorithm not detecting a particular subgraph is $1 - e^{-n_F}$. After repeating the algorithm $e^{n_F \log \frac{1}{\varepsilon}}$ times, the probability (with the usage of inequality $(1 + x) \leq e^x$) is:

$$(1 - e^{-n_F})^{e^{n_F \log \frac{1}{\varepsilon}}} = \left(\left(1 - \frac{1}{e^{n_F}}\right)^{e^{n_F}} \right)^{\log \frac{1}{\varepsilon}} < \left(\frac{1}{e}\right)^{\log \frac{1}{\varepsilon}} = e^{\log \varepsilon} = \varepsilon$$

\square

In Table 3.4, Table 3.5 and Table 3.6 we show how many repetitions of the main algorithm would be needed in the case of the desired error rate $\varepsilon = 0.5$, $\varepsilon = \frac{1}{e}$ and $\varepsilon = 0.01$, respectively.

Table 3.4: Number of the algorithm repetitions needed to achieve $\varepsilon = 0.5$.

| n_F | Number of repetitions |
|-------|-----------------------|
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 10 | 21 |
| 15 | 90 |
| 20 | 403 |

3. COLOR CODING ALGORITHM

Table 3.5: Number of the algorithm repetitions needed to achieve $\varepsilon = \frac{1}{e}$.

| n_F | Number of repetitions |
|-------|-----------------------|
| 2 | 8 |
| 3 | 21 |
| 4 | 55 |
| 5 | 149 |
| 10 | 22027 |
| 15 | 3269017 |
| 20 | 485165196 |

Table 3.6: Number of the algorithm repetitions needed to achieve $\varepsilon = 0.01$.

| n_F | Number of repetitions |
|-------|-----------------------|
| 2 | 55 |
| 3 | 404 |
| 4 | 2981 |
| 5 | 22027 |
| 10 | 485165196 |
| 15 | $1.06 \cdot 10^{13}$ |
| 20 | $2.35 \cdot 10^{17}$ |

The number of repetitions needed to achieve a promisingly low error rate is in theory rather high. However, the nature of the algorithm at least partially solves this problem. Because we are repeating the algorithm which on itself produces continuous results, it is possible to obtain already found subgraphs progressively, even though the whole procedure has not in theory finished yet. By extending the running time, we gradually lower the error rate of the algorithm and we can thus run the algorithm as long as we are not satisfied with the result. This is a very good property of the algorithm, as we are not required to wait for it to become finished, which could possible take (as seen in Table 3.6) many repetitions.

We can now formulate the final and complete version of the algorithm, which solves the SUBISOENUM problem, in which we also require the algorithm to locate and output the isomorphic subgraphs (as described in Section 1.3). The algorithm is described in Algorithm 3.11 and its properties are formally proved in Theorem 3.4.

Theorem 3.4 *There is an algorithm that solves the SUBISOENUM problem for an input graph G and a pattern graph F with probability at least $(1 - \epsilon)$ in $\mathcal{O}\left(n_G^{\text{TW}(F)+1} 2^{\mathcal{O}(n_F \log \frac{1}{\epsilon})} \log R + e^{n_F \log \frac{1}{\epsilon}} S R \cdot \text{TW}(F)\right)$ time and $\mathcal{O}(R + S \cdot n_F)$ space, where S is the number of solutions to the problem and R is the maximal*

Algorithm 3.11 SUBISOALGORITHM(G, F, ε)

Input: Graph G , graph F and an error rate ε .

Output: A set of subgraphs of G which are isomorphic to F with error ε .

Procedure:

1. Let $\mathcal{R} := \emptyset$.
 2. Let $\eta := \text{NICETREEDecomposition}(F)$, as described in Algorithm 2.7.
 3. Repeat $e^{n_F \log \frac{1}{\varepsilon}}$ times:
 - (1) Let ζ be a random coloring $\zeta: V(G) \mapsto \{1, 2, \dots, n_F\}$.
 - (2) Perform the dynamic programming part of the algorithm, that is $\mathcal{D}_{t,r} := \text{SUBISOBOTTOMUP}(G, F, \zeta, \eta)$, as described in Algorithm 3.1.
 - (3) Perform the reconstruction from $\mathcal{D}_{t,r}$ into \mathcal{R}_ζ , that is $\mathcal{R}_\zeta := \text{RECONSTRUCTRESULTS}(\mathcal{D}_t)$, as described in Algorithm 3.10.
 - (4) Set $\mathcal{R} := \mathcal{R} \cup \mathcal{R}_\zeta$.
 4. Return \mathcal{R} .
-

number of nonzero dynamic programming configurations encountered over all runs of its internal dynamic programming procedure. Dynamic programming configurations are part of a state space of size $\mathcal{O}\left(n_G^{\text{TW}(F)+1} \text{TW}(F) n_F 2^{n_F}\right)$.

Proof The corresponding Algorithm 3.11 is a combination of three algorithms: Algorithm 2.7, Algorithm 3.1 and Algorithm 3.10. As we discussed in the proof of Theorem 3.1, time and space requirements of Algorithm 2.7 are insignificant, if compared with the requirements of Algorithm 3.1. The time and space requirements of the reconstruction algorithm (described in Algorithm 3.10) for a single run of the dynamic programming algorithm are by Theorem 3.3 dependent on the number of solution to the problem S and on the number of nonzero configurations encountered R . Time required for a single reconstruction of results is $\mathcal{O}(SR \cdot \text{TW}(F))$ and space required is $\mathcal{O}(R + S \cdot n_F)$.

To achieve an error rate of ε , by Proposition 3.2 it suffices to repeat the core procedure and the reconstruction of results $e^{n_F \log \frac{1}{\varepsilon}}$ times, which gives us the time complexity required in total. \square

Implementation

This chapter contains a description of a module, which implements the color coding algorithm presented in Algorithm 3.11 and all of its affiliated procedures. So far, in numerous parts of the thesis, we have established that to effectively solve the extended SUBISO problem, we need to optimize our approach down to the slightest detail. In the previous chapter, we primarily concentrated on the optimizations of the color coding algorithm itself. The necessity of optimization however holds also for the implementational part of the algorithm, which is what influenced the solution of several design questions.

4.1 Chosen technologies

To be able to make the resulting module as efficient as possible, we opted to create the implementation in C. To address the performance matter even further, the module is based on a low-level C library LibUCW. The library contains many tools needed to achieve an efficient implementation and we closely describe it in Section 4.3.

4.2 Licensing and availability

The implemented module is available under the LGPL license, with compliance to the license of this thesis. Its source codes can be found on the enclosed CD, or in a public Github repository located at <http://github.com/josik/subiso/>.

4.3 LibUCW library

The used LibUCW library is a C library freely available under the LGPL license. Its home page is located at <http://www.ucw.cz/libucw/>.

The library offers a set of efficient tools in order to maximize the performance of C programs. The performance of the library code is achieved by a highly optimized code, most particularly in a way directed to reduce the number of function calls. Many parts of the code are thus generated by C preprocessor directly for the intended usage. To make the implementation of the module efficient, we used the following parts of the LibUCW library:

- *Growing arrays* – Arrays that automatically handle their size and perform possibly needed reallocations. They allow to store items of an arbitrary type and are represented by a simple pointer. We use these arrays instead of regular C arrays in all cases when an array without a fixed size is needed.
- *Growing fastbuffers* – Data buffers that are optimized for speed and dynamically adapt their size to the size of their content. It is not allowed to interleave requests to read and write to the buffers, as only a single mode (read/write) is allowed at a time. We use these buffers to store serialized and compressed data of dynamic programming lists.
- *Hash tables* – Hash tables that are generated specifically for the given parameters, such as key type, or key comparison function. We use hash tables to represent adjacency list for graphs.
- *Red black trees* – Red black trees are similarly to hash tables generated for the given parameters. We mainly use red black trees during the processing of dynamic programming lists.
- *Sorter* – Sorting routine which is a combination of quick sort and insert sort and is optimized by limiting function calls by inlining.
- *Variable length encoding* – Variable length encoding of integer numbers, whose properties are described in Table 3.3. We use the encoding to reduce the memory needed for dynamic programming lists.

4.4 Structure of the module

Because the final form of the implemented algorithm consists of several sub-routines, each of which covers a specific part of the computation, the module can be logically divided into parts. The division is as follows:

- Part covering the creation of a tree decomposition for a given pattern graph. This part is implemented in `tree_dec.c`.
- Part covering the transformation of a tree decomposition to a nice one. This part is implemented in `nice_tree_dec.c`.

- Part covering the searching for isomorphic subgraphs based on a nice tree decomposition. This part is implemented in `subiso.c`.
- Part covering the processing of found results and their interpretation. This part is implemented in `graph_result.c`.

In addition to the parts of the implemented algorithm, the module also contains a part covering the testing of the structures created during the run of the algorithm, which is implemented in `tests.c`. Namely, it contains procedures to test the properties either of created nice tree decompositions or found result subgraphs. A detailed description of the implemented tests can be found in Section 5.3.

4.5 Usage of the module

The module is accessible through command-line interface and the input is loaded through specifiable external files. The output from the module is directed to the standard output. After a successful installation (which is described in Appendix C), the module can be run via binary file `grs` created during the installation.

4.5.1 Run parameters

To run the module, there are mandatory parameters which have to be specified. The full interface of the module is as follows:

```
./grs <source graph file> <pattern graph file>  
      [computation seed] [number of repetitions]
```

The meaning of each particular parameter is:

- Mandatory parameters:
 - `<source graph file>` – file with graph G in the format specified below.
 - `<pattern graph file>` – file with graph F in the format specified below.
- Optional parameters:
 - `[computation seed]` – number for random number generator initialization (in case it is not specified, the seed is determined randomly).
 - `[number of repetitions]` – number of repetitions of the main algorithm (in case it is not specified, the algorithm is repeated until a theoretical rate of error $\varepsilon = \frac{1}{e}$ is achieved).

4.5.2 Graph file format

Graph files which are presented to the module must be in accordance to a specific format, which is in the following form.

On the first line of the graph file, there is a number of vertices n of the graph. On the i -th from the next n lines, there is information about the neighbors of the $(i - 1)$ -th vertex (vertices are thus to be numbered from 0). Specifically, there is a number s_{i-1} of neighbors of vertex $i - 1$. After that, there are s_{i-1} numbers, each of which represents a neighbor of vertex $i - 1$. An example of this input format is shown in Figure 4.1.

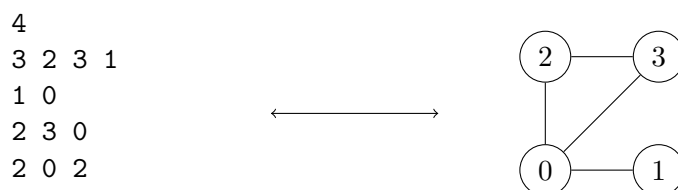


Figure 4.1: Graph representation in files.

4.5.3 Produced output

As described in Section 5.1, the algorithm continuously produces result subgraphs, because its main part is run repeatedly. To allow users of the module to extract results in cases when the specified number of runs has not yet been finished, it suffices to output found subgraphs after each run. In order to make the output human readable (especially for this thesis), the enclosed implementation outputs vertex unique subgraphs after all runs of the algorithm are completed. The output format is in the form so it is easily readable. In addition, current version of the module shows information about the current computation (e.g., about the number of subgraphs found so far).

Results and performance

In this chapter we perform series of qualitative and quantitative tests of the implemented module. To demonstrate its usability, we chose, with some exceptions, to test its performance on real-world graphs. We also measure how efficient were in fact some of the optimizations we proposed and implemented. In the end we discuss the results in the comparison to other related work.

5.1 Testing environment

The testing was performed on a 64-bit linux system with Intel Core i5-2400 CPU @ 3.10GHz. The module was compiled with gcc compiler (version 4.7.2) with `-O3` optimizations enabled. All quantitative results, if not said otherwise, are an average of 5 independent measurements.

5.2 Testing data

As an input graph G , we used three different graphs of various properties. We shall denote these graph instances as ECOLI, TRANS and SLASH.

The first of these instances, ECOLI, models protein-protein interactions of bacterium *Escherichia coli*. It is a rather small and sparse network, which consists of 673 vertices and 865 unoriented edges.

The second instance, TRANS, is an artificially made graph, which simulates transfers on bank accounts. It is a very sparse network, which consists of 45733 vertices and 44727 unoriented edges.

Third instance, SLASH, models Slashdot social network interactions from February 2009. More details about this instance can be found in [12]. It consists of 82168 vertices and 543382 unoriented edges.

As we can see, all three instances are quite sparse. However, for the problem SUBISOENUM, even such networks contain large amount of result subgraphs. We will confirm that in quantitative tests, where we will have to

restrict the number of enumerated solutions in order to measure the running time.

For the pattern graphs, we use 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, cycles and stars of several sizes, as well as grids and complete graphs. We shall denote a path on n vertices \mathcal{P}_n , a cycle on n vertices \mathcal{C}_n , a star on n vertices \mathcal{S}_n , a complete graph on n vertices \mathcal{K}_n and a $n \times m$ grid $\mathcal{G}_{n,m}$. Treewidths of these pattern graphs are shown in Table 5.1.

Table 5.1: Treewidth of pattern graphs.

| Pattern F | $\text{TW}(F)$ |
|---------------------|----------------|
| \mathcal{P}_n | 1 |
| \mathcal{S}_n | 1 |
| \mathcal{C}_n | 2 |
| \mathcal{K}_n | $n - 1$ |
| $\mathcal{G}_{n,m}$ | $\min\{n, m\}$ |

5.3 Qualitative tests

To ensure the results produced by the module are correct, as a part of module we implemented a set of tests in `tests.c`. With these tests we can check three important structures created during the computation. Firstly, we can check properties of the created tree decomposition and of the nice tree decomposition constructed from it. That is important because the tree decomposition properties need to be ensured for the main algorithm to work correctly. Secondly, we can check if all found subgraphs are isomorphic to the given pattern graph.

For all runs of the algorithm described and measured in this chapter, we additionally used the implemented tests to verify the correctness of created tree decompositions and most importantly of computed results. There was no run which would fail any of the described tests.

5.4 Quantitative tests

In this section we measure performance of the module, i.e., its real time and memory requirements. First we measure the time and space needed to build a nice tree decomposition. Then we show the requirements of the dynamic programming part of the whole algorithm.

5.4.1 Performance tests of nice tree decomposition construction

We have already discussed that in theory, it suffices for us to compute a tree decomposition in time exponential to the size of the pattern. We also claimed that due to the incomparability of the requirements of the nice tree decomposition construction procedure and of the main part of the algorithm, during the construction we can precompute many useful things with no negative consequences on the total running time.

We confirm the theoretical results by measuring the time and space needed to build a nice tree decomposition for various patterns of a given size and treewidth. The results of this test are shown in Table 5.2.

Table 5.2: Performance of nice tree decomposition construction.

| n_F | $\text{TW}(F)$ | Time [s] | Memory [MB] |
|-------|----------------|----------|-------------|
| 10 | 1 | 0.01 | 0.93 |
| 15 | 1 | 0.26 | 0.94 |
| 20 | 1 | 12.36 | 4.80 |
| 10 | 2 | 0.01 | 0.93 |
| 15 | 2 | 0.25 | 0.94 |
| 20 | 2 | 12.58 | 4.84 |
| 10 | 3 | 0.01 | 0.93 |
| 15 | 3 | 0.32 | 0.94 |
| 20 | 3 | 16.08 | 4.80 |

By observing the measured results, we can see that the algorithm used to construct a tree decomposition is indeed exponential only in the size of the input pattern graph. The slightly higher running time for $\text{TW}(F) = 3$ can be caused by a different structure of the tree decomposition, which as a result makes the procedure of construction of a nice tree decomposition do more work.

As a main result from this test, we see that compared to running times mentioned in the next section, the construction of a nice tree decomposition doesn't induce a slowdown and the procedure which we are using was thus chosen correctly.

5.4.2 Performance tests of the algorithm

Due to the randomized properties of the main algorithm, in order to achieve a certain 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, as discussed in Section 5.1. It is then only logical to measure performance of the algorithm

in the context of a single run of the algorithm only. Results from such a testing, however, should be still taken as an rough average, because the running time of a single run of the algorithm depends on many factors.

In the light of these facts, we have approached the performance tests of the algorithm in a way, where we average the results of many single runs of the algorithm. 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 n times the time required for a single run, if there are n runs in total.

For testing, we used the mentioned pattern graphs in various sizes, most commonly in sizes 5, 10 and 15. We divided the time and space required into two parts. One part covers the dynamic programming subroutine, which solves the SUBISO problem. We denote this part as a computational part, shortly Comp. The second part we measure is the part of reconstruction of the results. We denote this part as a reconstruction part, shortly Recon. The reason behind this division is to measure how effectively we improved the solution for the SUBISO problem on its own – this corresponds to the computational part. The reconstruction part then covers the retrieval of the solution to the SUBISOENUM problem from the dynamic tables solving the SUBISO problem.

Also, because of the time bounds of the algorithm shown in Theorem 3.4, the complexity of solution reconstruction depends directly on the number of solutions. In order to be able to measure the computation for larger networks with many result subgraphs, in reconstruction part we measure only the time and space required to retrieve first 100000 solutions. We also restrict ourselves with a 1000 MB memory limit for the computational part, as the dynamic programming lists use the largest part of the memory for the computational part, and the time required to reconstruct results also directly depends on their size.

We show the results of the performance testing in Table 5.3 for ECOLI, in Table 5.4 for TRANS, and in Table 5.5 for SLASH.

Table 5.3: Performance of a single run of the algorithm on ECOLI dataset.

| Pattern | Comp. time [s] | Comp. memory [MB] | Recon. time [s] | Recon. memory [MB] | Results |
|---------------------|----------------|-------------------|-----------------|--------------------|---------|
| \mathcal{P}_5 | 0.01 | 1.46 | 0.03 | 1.91 | 836 |
| \mathcal{P}_{10} | 0.01 | 1.81 | 0.71 | 15.43 | 3014 |
| \mathcal{P}_{15} | 0.05 | 4.48 | 6.24 | 97.70 | 1956 |
| \mathcal{S}_5 | 0.01 | 1.67 | 0.03 | 3.49 | 640 |
| \mathcal{S}_{10} | 0.01 | 1.92 | 0.15 | 23.44 | 62 |
| \mathcal{S}_{15} | 0.02 | 3.7 | 0.31 | 31.18 | 1 |
| \mathcal{C}_5 | 0.01 | 1.81 | 0.01 | 1.88 | 6 |
| \mathcal{C}_{10} | 0.11 | 6.45 | 0.09 | 6.82 | 2 |
| \mathcal{C}_{15} | 0.74 | 19.51 | 2.73 | 19.57 | 1 |
| $\mathcal{G}_{3,4}$ | 0.42 | 13.13 | – | – | 0 |
| \mathcal{K}_4 | 0.01 | 1.40 | – | – | 0 |

Table 5.4: Performance of a single run of the algorithm on TRANS dataset.

| Pattern | Comp. time [s] | Comp. memory [MB] | Recon. time [s] | Recon. memory [MB] | Results |
|---------------------|----------------|-------------------|-----------------|--------------------|---------|
| \mathcal{P}_5 | 0.37 | 32.45 | 14.32 | 35.94 | 34572 |
| \mathcal{P}_{10} | 0.80 | 34.06 | 0.32 | 35.02 | 331 |
| \mathcal{P}_{15} | 1.07 | 34.83 | 0.55 | 35.49 | 5 |
| \mathcal{S}_5 | 0.86 | 37.60 | 1.24 | 96.86 | 96615 |
| \mathcal{S}_{10} | 2.28 | 49.90 | 1.67 | 119.12 | 41277 |
| \mathcal{S}_{15} | 5.80 | 63.25 | 3.11 | 154.86 | 24274 |
| \mathcal{C}_5 | 3.24 | 157.66 | 0.13 | 159.14 | 2 |
| \mathcal{C}_{10} | 7.14 | 227.46 | – | – | 0 |
| \mathcal{C}_{15} | 9.73 | 253.30 | – | – | 0 |
| $\mathcal{G}_{3,4}$ | – | > 1000 | – | – | – |
| \mathcal{K}_4 | 0.28 | 31.73 | – | – | 0 |

Table 5.5: Performance of a single run of the algorithm on SLASH dataset.

| Pattern | Comp. time [s] | Comp. memory [MB] | Recon. time [s] | Recon. memory [MB] | Results |
|---------------------|----------------|-------------------|-----------------|--------------------|----------|
| \mathcal{P}_5 | 13.3 | 231.67 | 230.95 | 299.67 | > 100000 |
| \mathcal{P}_{10} | 49.13 | 825.60 | 274.63 | 944.52 | > 100000 |
| \mathcal{P}_{15} | – | > 1000 | – | – | – |
| \mathcal{S}_5 | 12.07 | 236.55 | 42.89 | 306.47 | > 100000 |
| \mathcal{S}_{10} | 30.97 | 331.58 | 101.22 | 488.01 | > 100000 |
| \mathcal{S}_{15} | 196.01 | 463.60 | 207.46 | 793.25 | > 100000 |
| \mathcal{C}_5 | – | > 1000 | – | – | – |
| \mathcal{C}_{10} | – | > 1000 | – | – | – |
| \mathcal{C}_{15} | – | > 1000 | – | – | – |
| $\mathcal{G}_{3,4}$ | – | > 1000 | – | – | – |
| \mathcal{K}_4 | – | > 1000 | – | – | – |

5.5 Modification tests

In this section we test, whether the optimizations proposed in Section 3.4.2 have positive influence on the algorithm or not. In particular, we test whether the compressed serialization of mapping records is of any use and we also test the impact of optimizations regarding mapping expansion.

5.5.1 Tests of buffer compression

We tested the difference between buffers storing plain numbers and buffers storing delta compressed numbers additionally encoded with variable length code. To do so, we compared the time and space required for both implementations in a single run of the computational part of the algorithm on several problem instances. The results are shown in Table 5.6 and Table 5.7.

Table 5.6: Time and space needed with uncompressed buffers.

| Problem instance | Time [s] | Memory [MB] |
|---------------------------|----------|-------------|
| ECOLI, \mathcal{C}_{15} | 0.70 | 22.09 |
| TRANS, \mathcal{S}_{15} | 5.51 | 73.99 |
| SLASH, \mathcal{P}_{10} | 56.48 | 1298.17 |
| SLASH, \mathcal{S}_{10} | 29.91 | 381.92 |

Table 5.7: Time and space needed with compressed buffers.

| Problem instance | Time [s] | Memory [MB] |
|---------------------------|----------|-------------|
| ECOLI, \mathcal{C}_{15} | 0.74 | 19.51 |
| TRANS, \mathcal{S}_{15} | 5.80 | 63.25 |
| SLASH, \mathcal{P}_{10} | 49.13 | 825.60 |
| SLASH, \mathcal{S}_{10} | 30.97 | 331.58 |

As a result, we can see that the compressed implementation of buffers really helps to keep the memory requirements low. Table 5.8 shows, that the time spent with the compression is irrelevant in contrast with the memory saved. In one case the implementation with compression is even faster, as there is much less content to be read from the buffer than in the uncompressed case.

5.5.2 Tests of mapping optimizations

Optimizations regarding mapping extensions were also tested. In this test, we measured how many assignments to vertices from G were for some problem instance candidates to be added to a certain mapping, both in cases with and

Table 5.8: Compressed buffer space storage efficiency.

| Problem instance | Slowdown ratio | Memory saved [MB] |
|---------------------------|----------------|-------------------|
| ECOLI, \mathcal{C}_{15} | 1.06 | 2.58 |
| TRANS, \mathcal{S}_{15} | 1.05 | 10.74 |
| SLASH, \mathcal{P}_{10} | 0.87 | 472.57 |
| SLASH, \mathcal{S}_{10} | 1.04 | 50.34 |

without the mapping expansion optimization #1 and #2. The result of the test is described in Table 5.9.

Table 5.9: Number of vertex candidates with/without mapping optimizations.

| Problem instance | Without opt. | With opt. | Ratio |
|---------------------------|--------------|-----------|---------------------|
| ECOLI, \mathcal{C}_{15} | 263442144 | 1368072 | $5.1 \cdot 10^{-3}$ |
| TRANS, \mathcal{S}_{15} | 29281742308 | 1892618 | $6.4 \cdot 10^{-5}$ |
| SLASH, \mathcal{P}_{10} | 25225493832 | 4383805 | $1.7 \cdot 10^{-4}$ |
| SLASH, \mathcal{S}_{10} | 60764961528 | 10520379 | $1.7 \cdot 10^{-4}$ |

From the results it is clear that proposed mapping expansion optimizations have huge effect on the computational part of the algorithm.

5.6 Discussion of results and comparison to related work

Tests imply that our implementation of the color coding algorithm properly solves the SUBISOENUM problem even for large instances of the input graph. In particular, it has been shown that, in the computational part, we are able to effectively solve the SUBISO problem. From that, in the reconstruction part, we are able to obtain the results of the SUBISOENUM problem.

The only cases when we are not able to obtain all of the results happen when the input graph contains so many solutions, that we are not able to store them due to the lack of space. Even though we devised a very efficient method of compression of the used mapping structure, we can't store an amount of solutions, which is several magnitudes higher than the size of the memory we possess.

5.6.1 Comparison to related work

The implementation of new VF2 algorithm presented in [8] is, compared to our module, very fast, but only solves the induced variant of the SUBISO problem.

5. RESULTS AND PERFORMANCE

The implementation presented in [19] incorporates similar approach of color coding, but is in addition to our module implemented in parallel.

The solution in [16], as many of other subgraph isomorphism papers (e.g., [18] or [11]), solves the problem only for paths or trees . In contrast to this, our solution allows to query for any pattern graph up to 20 vertices.

Conclusion

We surveyed known results regarding the subgraph isomorphism problem and we described the choice of the color coding approach. For the construction of a nice tree decomposition, we discussed the preference of an algorithm exponential in the size of the input graph. Within the same matter, we described a non-trivial conjunction of the theory about chordal graphs, elimination orderings and tree decompositions in order to be able to obtain tree decompositions of optimal width. We modified and optimized the original color coding algorithm in a way which allows its usage even on instances that would otherwise not be admissible because of memory and/or time requirements.

We implemented an efficient C module which contains all parts of the wide-ranging solution. Namely, we implemented the construction of a tree decomposition, the transformation of a tree decomposition to a nice one, and the modified bottom-up dynamic programming algorithm based on color coding. To maximize the performance of the module, we used various tools from a low-level C library LibUCW.

We extensively tested the implemented module both from qualitative and quantitative points of view. In particular, not only did we successfully verify the validity of outputs created by the module, but we also showed its ability to cope with instances of large size.

Future work

For the future research of the issue, there are two main directions to be considered. One of the them would be based on designing further optimizations of the module, mostly in an implementational way. Because the main computational part of the algorithm lies in the genuine testing of assignments of many vertices, there exists a possibility to enhance this very algorithm to run in parallel. There are also many modifications that could be implemented into the module, e.g., support of oriented graphs or support of vertex/edge labels,

CONCLUSION

which allow us to more precisely specify the area of the input graph to be searched.

The second direction would be to incorporate a theoretical approach, as there are many interesting properties of the problem that could be exploited to do some of the operations more effectively. In particular, it would be interesting to more closely aggregate the computational and reconstruction phase in the presented algorithm.

Bibliography

- [1] Alon, N., Yuster, R., and Zwick, U. Color-coding. *J. ACM* 42, 4 (1995), 844–856.
- [2] Arnborg, S., Corneil, D. G., and Proskurowski, A. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods* 8, 2 (1987), 277–284.
- [3] Bodlaender, H. L. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.* 25, 6 (1996), 1305–1317.
- [4] Bodlaender, H. L., Fomin, F. V., Koster, A. M. C. A., Kratsch, D., and Thilikos, D. M. On exact algorithms for treewidth. *ACM Trans. Algorithms* 9, 1 (2012), 12:1–12:23.
- [5] Bodlaender, H. L., and Koster, A. M. C. A. Treewidth computations I. upper bounds. *Inf. Comput.* 208, 3 (2010), 259–275.
- [6] Bonnici, V., Giugno, R., Pulvirenti, A., Shasha, D., and Ferro, A. A subgraph isomorphism algorithm and its application to biochemical data. *BMC bioinformatics* 14, Suppl 7 (2013), S13.
- [7] Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. Performance evaluation of the VF graph matching algorithm. In *Proceedings of the 10th International Conference on Image Analysis and Processing* (1999), ICIAP '99, IEEE Computer Society.
- [8] Cordella, L. P., Foggia, P., Sansone, C., and Vento, M. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 26, 10 (2004), 1367–1372.
- [9] Cygan, M., Fomin, F. V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., and Saurabh, S. *Parameterized Algorithms*. Springer, 2015.

- [10] Golumbic, M. C. *Algorithmic Graph Theory and Perfect Graphs*, 2nd ed. Elsevier, 2004.
- [11] Hüffner, F., Wernicke, S., and Zichner, T. Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica* 52, 2 (2008), 114–132.
- [12] Leskovec, J., Huttenlocher, D., and Kleinberg, J. Predicting positive and negative links in online social networks. In *Proceedings of the 19th International Conference on World Wide Web* (2010), ACM, pp. 641–650.
- [13] Marx, D. Can you beat treewidth? *Theory of Computing* 6, 1 (2010), 85–112.
- [14] McKay, B. D., and Piperno, A. Practical graph isomorphism, II. *J. Symb. Comput.* 60 (2014), 94–112.
- [15] Röhrig, H. Tree decomposition: A feasibility study. Master’s thesis, Universität des Saarlandes, 1998.
- [16] Romijn, L., Nualláin, B. Ó., and Torenvliet, L. Discovering motifs in real-world social networks. In *SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science. Proceedings* (2015), pp. 463–474.
- [17] Ullmann, J. R. An algorithm for subgraph isomorphism. *J. ACM* 23, 1 (1976), 31–42.
- [18] Zhao, Z. Subgraph querying in relational networks: A mapreduce approach. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum* (2012), IPDPSW ’12, IEEE Computer Society, pp. 2502–2505.
- [19] Zhao, Z., Khan, M., Kumar, V. S. A., and Marathe, M. V. Subgraph enumeration in large social contact networks using parallel color coding and streaming. In *Proceedings of the 2010 39th International Conference on Parallel Processing* (2010), ICPP ’10, IEEE Computer Society, pp. 594–603.

Acronyms

DFS Depth-first search

ETH Exponential time hypothesis

LGPL GNU Lesser General Public License

Contents of enclosed CD

| | |
|----------------------|---|
| readme.txt | the file with CD contents description |
| src | the directory with source codes |
| ├── subiso | sources of the implemented module |
| ├── instances | graph files of instances used in testing |
| ├── thesis | the directory of \LaTeX source codes of the thesis |
| text | the thesis text directory |
| ├── thesis.pdf | the thesis text in PDF format |

Module installation guide

C.1 Prerequisites

Module consists of C implementation files, C header files and an installation file `Makefile`. It is intended to be run on a unix-like system. For the installation, the system is required to contain the following prerequisites:

- GNU tools:
 - `gcc` version 4.0 or higher
 - `bash` version 2.0 or higher
- LibUCW (for installation guide see <http://www.ucw.cz/libucw/>)
- Additional LibUCW prerequisites:
 - `perl`
 - `pkg-config`

C.2 Installation

The installation is to be performed from the folder containing installation file `Makefile`. There are in total three modes in which the module can be installed:

- Installation by command `make` – This is the default mode of installation. It produces minimal information about the run and yields the maximal performance. It is suitable for the regular usage of the module, or for the performance testing of the module.
- Installation by command `make tests` – This is the testing mode of installation. It produces minimal information about the run, but in addition tests the validity of structures created during the run. It is suitable for the qualitative testing of the module.

C. MODULE INSTALLATION GUIDE

- Installation by command `make debug` – This is the debug mode of installation. It produces detailed information about the run and the structures used in the run. It is suitable for the debugging purposes.

After a successful installation, a binary executable file `grs` is produced. To uninstall the module or to change the installation type of the module, it is required to execute command `make clean`. After its execution the module folder is reverted to its original state.