Counting Graph Homomorphisms in Rust

by

Julian Mende, [redacted]

supervised by

Prof. Dr. Holger Dell

submission date: 14/06/2022

A document submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

at

Goethe University Frankfurt

Abstract

This work extends the problem of counting homomorphisms for two graphs to the problem of counting homomorphisms for a set of graphs. The set H_τ of graphs will be characterized by a single tree decomposition τ. For given τ and an arbitrary graph G the task is to compute $hom(H \to G)$ for all graphs $H \in H_\tau$. It contains a modified version of the dynamic program of Díaz, Serna and Thilikos [8]. The original and the modified version have been implemented in the Rust language and compared empirically. Experiments show that the modified version is faster than the original one for the majority of test instances.

Contents

1 Introduction

Graphs are one of the most important structures in theoretical computer science and have countless amount of applications. Their ability to model many types of networks like social relations or real-world maps is known even by non-computer-scientists. A countless number of graph algorithms can be used to solve many problems like finding the shortest path from one location to another. Even though many characteristics and parameters of graphs have been investigated intensively some of the graph problems remain hard to solve. One of those hard problems is the so-called **Counting Graph Homomorphisms Problem**. Given two graphs H and G this problem asks to compute the number of edge-preserving mappings from H to G. Are more precise definition of graph homomorphisms can be found in chapter 2.

Counting homomorphism itself seems to be a more theoretical problem but has applications in statistical physics and other graph problems. For example, the problem of counting **Independent Sets** in a given graph H can be reduced to the problem of counting all homomorphisms from H to G_{IS} where G_{IS} denotes the graph containing two connected vertices with one self-loop. Furthermore the number of isomorphic subgraphs can be computed by linear combinations of graph homomorphisms[4].

1.1 Structure

Chapter 2 explains the theoretical fundamentals in more detail with some digression to related topics. It contains all necessary definitions such as those for tree decompositions and graph homomorphisms. Additional chapter 2 provides the most known complexity results regarding the problem of counting graph homomorphisms. Chapter 3 covers the extended problem of this paper and shows approaches by simply repeating existing algorithms for graph homomorphisms. Chapter 4 provides the theoretical idea for modifying the dynamic program of Díaz, Serna and Thilikos [8] and the concrete modification itself. The implementation of these algorithms will be explained in chapter 5. It also contains explanations of all supported input and output formats. The last chapter 6 sums up the experiments and results.

2 Theoretical Basics

2.1 Basic Notation

The cardinality of any set A will be denoted by $\#A$ or $|A|$ and let $\mathcal{P}(A)$ be the power set of A.

In this work, Graphs will be denoted by capital letters like G or H. We are dealing with graphs where vertices could have self-loops but cannot have multiple edges towards the same vertex (no multigraphs). A graph G is defined as a tuple $G = (V(G), E(G))$ which consists of the finite set of vertices $V(G)$ and the set of edges $E(G) \subseteq V(G)^2$. We denote the edge $\{u, v\}$ shortly as uv.

For a Graph G and a subset of vertices $A \subseteq V(G)$ we define $G[A]$ as the subgraph of G which is induced by the vertices of A. We define $E[A] = \{uv \in E : u \in A \wedge v \in A\}$. It follows that $G[A] = (A, E[A])$.

For better distinction, we will call the vertices of trees **nodes**. The root of a tree T will be denoted by $r(T)$.

For two graphs G and H and a subset of vertices $X \subseteq V(G) \cap V(H)$ we denote by $\sim_X$ the equality of both graphs regarding the vertices in X.

definition 2.1. *For two graphs G and H and a subset X of vertices in $V(G) \cap V(H)$ we define $\sim_X$ as follows.*

$$G \sim_X H \iff G[X] = H[X] \tag{2.1}$$

2.2 Treewidth

The treewidth of a graph describes its structural resemblance to a tree. This concept was first introduced by Umberto Bertelè and Francesco Brioschi in the year 1972. In their paper [1] the treewidth was called dimension and has been defined for a sequence of so-called y-eliminations. The concept of treewidth was rediscovered and set into the context of graph minors by Neil Robertson and Paul Seymour in the year 1983 and after [16, 18, 17]. Robertson and Seymour also introduced an underlying structure for graphs, the so-called tree decomposition. Tree decompositions are the fundamental structure of this work and therefore need to be explained in more detail. The concept of treewidth strongly correlates to the idea of k-trees which are well explained in the introduction

of the book [14] by Tom Kloks. A nice introduction to treewidth and tree decompositions can be found in [5]. The following sections will describe these topics in a little bit more detail.

2.2.1 K-trees and Partial K-trees

K-trees can be imagined as trees where each tree node does not only consists of one single vertex but of a set of vertices. They are a wider abstraction of standard trees known in theoretical computer science. These bags of nodes correspond to a node of a underlying abstract tree. This tree-like structure make it easy to use algorithm design techniques already used on standard trees and adjust already known algorithms to broader sets of graphs. A k-tree can be defined recursively as following.

definition 2.2 (k-trees)**.** *K-trees are defined by construction. A Clique of size $k+1$ is a k-tree. A k-tree T' with $n+1$ vertices can be constructed from a k-tree T with n vertices by adding a new vertex u and connecting u to a k-Clique in T.*

A subgraph S of a k-tree T will be called a partial k-tree. Then we will call T a k-tree embedding of S. Figure 2.1 exemplary shows the first two steps of a 3-tree construction.

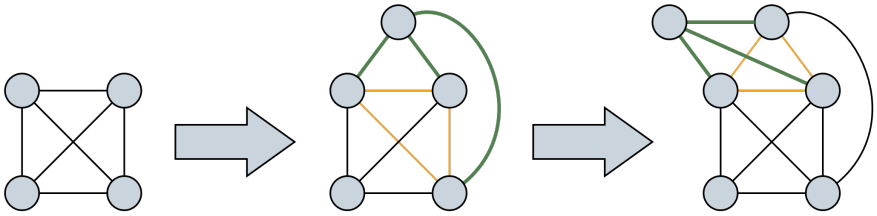

Figure 2.1: A simple construction of a 3-Tree

2.2.2 Treewidth

After defining k-trees the first definition of treewidth based on finding k-tree embeddings for a graph G can be stated. Colloquially said for a given graph G we try to find the smallest k-tree which functions as a k-tree embedding for G. And by smallest we mean the smallest k for which we can find a k-tree that fulfils the requirement of being a supergraph of G. The following definition is taken from [14].

definition 2.3 (treewidth)**.** *Let G be an arbitrary graph. The treewidth of G is the minimum k for which a k-tree embedding of G exists. The treewidth of G will be denoted by $tw(G)$*

Hence the class of partial k-trees is exactly the class of graphs with treewidth at most k.

2.2.3 Tree Decompositions

The treewidth plays an important role in parameterized complexity [5] since many problems can be characterized by this additional parameter. By restricting the input to graphs with bounded treewidth we can obtain algorithms for NP-hard problems which have a polynomial runtime in all other parameters.

But to use treewidth for parameterized problems we need another theoretical construct, the so-called tree decomposition. This structure decomposes a graph into its underlying tree structure. This directly correlates to the tree-likeliness mentioned for k-trees. More formal tree decompositions could be defined as follows. The next definitions are strongly inspired by [16, 18, 17, 8].

definition 2.4 (tree decomposition)**.** *A tree decomposition τ of a Graph G is a pair $\tau = (T, \beta(p)_{p \in V(T)})$ consisting of a tree T and a collection of subsets of $V(G)$ such that*

1. $\bigcup_{p \in V(T)} \beta(p) = V(G)$
2. *for each edge $\{u, v\} \in E(G)$ exists a $p \in V(T)$ width $u \in \beta(p)$ and $v \in \beta(p)$*
3. *for each $u \in V(G)$ the nodes in $\{p \in V(T) | u \in \beta(p)\}$ form a connected subgraph of T (They form a subtree)*

The subsets $\beta(p)$ are also called bags. We define the width of tree decomposition as $width(\tau) = \max_{p \in V(T)}\{|\beta(p)| - 1\}$.

Tree decompositions provide an alternative view of treewidth. An equivalent to definition 2.3 can now be stated by using tree decompositions.

definition 2.5 (treewidth)**.** *The treewidth $tw(G)$ of a graph G is the minimum $w \geq 0$ such that there is a tree decomposition τ' of G with $width(\tau') = w$.*

By decomposing an arbitrary graph into a tree it now becomes easier to algorithmically work on it. But especially for dynamic programming, this "basic" variant of tree decomposition is not optimal. We now define two more variants of tree decompositions. The second one will be used by the algorithm presented in this work. For the first variant, we transform the tree from definition 2.4 into a rooted tree. The following definitions were introduced by Bodlaender and Kloks [2]

definition 2.6 (rooted tree decomposition)**.** *A tree decomposition $\tau = (T, \beta(p)_{p \in V(T)})$ is called rooted if T is a rooted tree. The root of T will be denoted by $r(T)$.*

Note that in this definition the tree T becomes a rooted tree. Hence for each edge $e \in E(T)$ we can distinguish the incident nodes between parent and child.

For each node $p \in V(T)$ we define T_p as the subtree of T rooted at p that we get if we delete the edge between p and its parent node and take the connected component containing p. The set $\alpha(p)$ is then defined as the union of all bags in T_p, that is

$$\alpha(p) = \bigcup_{q \in V(T_p)} \beta(q)$$

The second variant of tree decompositions is called nice tree decomposition. It specializes rooted tree decompositions even more by adding three additional conditions. These three more conditions make it easy to handle with dynamic programming. Nice tree decompositions were also introduced by Bodlaender and Kloks [2].

definition 2.7 (nice tree decomposition)**.** *We call a rooted tree decomposition $\tau = (T, \beta(p)_{p \in V(T)})$ nice if the following conditions are satisfied:*

1. *Each node $p \in V(T)$ has a maximum of two children.*
2. *For each node $p \in V(T)$ with exactly two children q_1, q_2 it is that $\beta(p) = \beta(q_1) = \beta(q_2)$.*
3. *For each node $p \in V(T)$ with one child $q \in V(T)$ either $|\beta(p)| = |\beta(q)| + 1$ and $\beta(q) \subset \beta(p)$ or $|\beta(p)| = |\beta(q)| - 1$ and $\beta(p) \subset \beta(q)$ holds*

For a tree decomposition with these properties, it is easy to see that every node in $V(T)$ belongs to one of the following types.

1. **Start:** Every leaf $l \in V(T)$ is called a start node
2. **Join:** Every node $p \in V(T)$ with two children q_1, q_2 is called a join node
3. **Forget:** Every node $p \in V(T)$ with exactly one child $q \in V(T)$ such that $\beta(p) < \beta(q)$ is called a forget node
4. **Introduce:** Every node $p \in V(T)$ with exactly one child $q \in V(T)$ such that $\beta(p) > \beta(q)$ is called an introduce node

For a vertex $v \in V(G)$ and a forget node p with its child node q we say that v is forgotten at p if $\beta(p) = \beta(q) \backslash v$. For a vertex $v \in V(G)$ and a introduce node p with its child node q we say that v is introduced at p if $\beta(p) = \beta(q) \cup v$. Note that in consequence of condition 3 of definition 2.4 every vertex in $V(G)$ can be introduced multiply times but only forgotten once.

It is possible to assume for each start node l that $|\beta(l)| = 1$. Every start node l with $|\beta(l)| > 1$ can be transformed into one start node l' with $|\beta(l')| = 1$ and $|\beta(l)| - 1$ introduce nodes, which add the remaining vertices. We will also assume that $r(T)$ is a forget node with $\beta(r(T)) = \emptyset$

The following lemma is taken from [2] and provides the possibility of assuming that we always look at nice tree decompositions.

Lemma 2.1. *For a constant $k \geq 1$, given a tree decomposition $\tau = (T, \beta(p)_{p \in V(T)})$ of a graph G with $width(\tau) \leq k$ and $|V(T)| = O(n)$, where n is the number of vertices of G, one can compute a nice tree decomposition τ^* of G in time $O(n)$ with $width(\tau^*) \leq k$ and at most $O(n)$ nodes.*

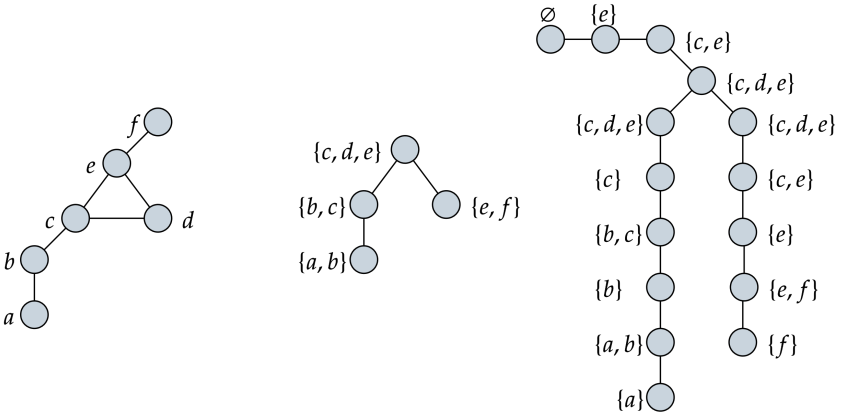

Figure 2.2: A graph(left) with one possible tree decomposition(middle) and a nice tree decomposition of it(right)

2.2.4 Traversing a Nice Tree Decomposition

The main algorithms shown in this work are dynamic programs following a special traversal of a nice tree decomposition, the so-called stingy ordering. This order reduces the amount of memory that has to be used at the same time to logarithmic in the number of nodes in the tree. The following definition is a variant of the definition from the paper [8] by Díaz, Serna and Thilikos.

definition 2.8 (stingy ordering)**.** *Let T be a rooted binary tree with $n = |V(T)|$. Let b the number of nodes in T with degree two, i.e. they have exactly two children. An ordering $u_1, \ldots, u_n$ of $V(T)$ is called stingy if the following conditions hold.*

1. *The first node of the ordering u_1 is a leaf.*
2. *The last node of the ordering u_n is the root, i.e. $u_n = r(T)$.*
3. *For every node u_i its parent appears at some position j with $j > i$.*
4. *For any j the number of nodes in the set $\{u_1, \ldots, u_j\}$ whose parents appear at a position $k > j$ is at most $\log b + 1$*

Later when calculating the number of homomorphisms the dynamic programs will follow the stingy ordering of a nice tree decomposition and calculate a list of entries for each node. This calculation will use only entries of child nodes. After the entries of a node have been calculated the dynamic program can delete the entries of its child nodes. Hence the number of nodes for which entries must be saved is at most $\log b + 1$. Calculating a stingy ordering can be done in time $O(n)$ and will be shown in section 4.1.

2.3 Graph Homomorphism and Isomorphism

2.3.1 Homomorphism

Let us move on to the main subject of this paper, graph homomorphisms. In this chapter, the basic definition of graph homomorphisms and isomorphisms will be stated and some notation will be fixed. Furthermore, some observations and claims are presented which will be used to improve the algorithms presented in the following chapter. A great introduction to the problem of counting graph homomorphisms was provided by Christian Borgs, Jennifer Chayes and Lázló Lovász in their paper[3]. For more information about graph homomorphisms themselves and their relations to other graph structures and problems, I recommend the paper[11] by Hahn and Tardif.

In general homomorphisms are structure-preserving mappings. For graphs, this means that we want to preserve edges. If two vertices in the domain graph have an edge in common the vertices they are mapped on should have too. More formally spoken, graph homomorphisms can be defined as follows.

definition 2.9 (graph homomorphism)**.** *Given two graphs H and G, a mapping $\phi : V(H) \to V(G)$ is called a graph homomorphism if it satisfies the following condition for each $\{u, v\} \in V(H)^2$.*

$$\{u, v\} \in E(H) \implies \{\phi(u), \phi(v)\} \in E(G) \tag{2.2}$$

Since this paper is about counting graph homomorphisms those will be simply called homomorphisms. A graph homomorphism from a graph G to another graph H is also called H-coloring, see for example[12]. This term is originate from the understanding of graph homomorphisms as an abstraction of n-colorings. The question of whether G has a n-Colorings is equivalent to the question of whether G has a homomorphism to the K_n where K_n denotes the complete graph with n vertices. But in the context of the above definition, the term H-Coloring will be avoided since the function of the graph identifier H and G has been swapped.

definition 2.10. *For two graphs H and G we define $Hom(H \to G)$ as the set of all homomorphisms from H to G.*

Counting homomorphisms between two graphs H and G means computing the cardinality of $|Hom(H \to G)|$. For easier notation we will denote the cardinality of $|Hom(H \to G)|$ as $hom(H \to G)$. A more precise definition of the counting problem dealt with in this paper can be found in definition 3.1. For the problem of counting graph homomorphism, it is only necessary to look at connected graphs. The following two properties of $hom(H \to G)$ taken from the paper [3] will justify this restriction.

Lemma 2.2. *Let G be an arbitrary graph and let the graph H be the disjoint union of the graphs $H_1, \ldots, H_n$ then the following equation holds.*

$$hom(H \to G) = \prod_{i=1}^{n} hom(H_i \to G) \tag{2.3}$$

Lemma 2.3. *Let H be an arbitrary graph and the graph G be the disjoint union of the graphs $G_1, \ldots, G_n$ then the following equation holds.*

$$hom(H \to G) = \sum_{i=1}^{n} hom(H \to G_i) \tag{2.4}$$

These two lemmata make it possible to first decompose a graph into its connected components, calculate the number of homomorphism for each component separately and then multiply or add them together.

2.3.2 Isomorphism

Structural similarity of two graphs could be achieved by restricting homomorphism only to the bijective ones. A bijective homomorphism is called an isomorphism. It follows the formal definition of graph isomorphism.

definition 2.11. *Given two graphs H and G. A function $\phi : V(H) \to V(G)$ is called isomorphism if ϕ is bijective and the following condition holds for every $\{u, v\} \in V(H)^2$.*

$$\{u, v\} \in E(H) \iff \{\phi(u), \phi(v)\} \in E(G) \tag{2.5}$$

Since f has to be a bijection the numbers of vertices of both graphs have to be identical. Also, the number of edges must be the same, since every edge in H will be bijectively mapped to exactly one edge in G. If there exists an isomorphism between two graphs H and G they will be called

isomorphic which will be denoted by $H \simeq G$. If two graphs H and G are isomorphic for every graph C the following equality holds.

$$|Hom(H \to C)| = |Hom(G \to C)| \tag{2.6}$$

This can be seen easily by constructing a homomorphism $\phi' : V(G) \to V(C)$ from each homomorphism in $Hom(H \to C)$ by simply using the image of an isomorphism between G and H.

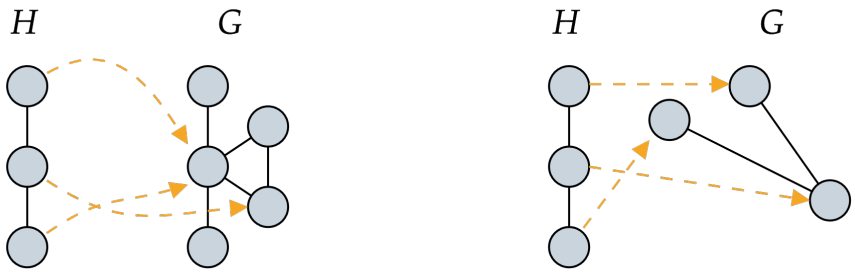

Figure 2.3: A graph homomorphism from graph H to graph G on the left and a graph isomorphism from H to G on the right

2.3.3 Complexity of Graph Homomorphism Counting

This section will be a short digression into the complexity of counting and finding graph homomorphisms. More precisely we look at the main result of Roth and Wellnitz [19]. They also provide a great introduction to the most important complexity results regarding graph homomorphism problems which we will follow along in this section. First of all, we have to distinguish between two sorts of problems regarding graph homomorphisms. The most basic sort of problem is the deciding problem, in which we are asked to decide for two given graphs if there exists a homomorphism from one graph to the other.

definition 2.12. *Let $\mathcal{H}$ and $\mathcal{G}$ be two classes of graphs. The problem $HOM(\mathcal{H} \to \mathcal{G})$ is then defined as follows. Given two graphs $H \in \mathcal{H}$ and $G \in \mathcal{G}$ decide whether there exists a homomorphism from H to G.*

Hell and Nesetril showed in their paper[12] that $HOM(\mathcal{U} \to \mathcal{G})$ where $\mathcal{U}$ denotes the class of all graphs in the following is NP-hard in generally but can be solved in polynomial time for some special classes. For the parameterized setting Grohe [10] showed that $HOM(\mathcal{H} \to \mathcal{U})$ can be

solved in polynomial time if the treewidth of the graphs in the class $\mathcal{H}$ is bounded by a constant k. On the other hand is the counting problem for graph homomorphism, the so-called Counting Graph Homomorphism Problem.

definition 2.13. *Let $\mathcal{H}$ and $\mathcal{G}$ be two classes of graphs. The problem $\#HOM(\mathcal{H} \to \mathcal{G})$ is then defined as follows. Given two graphs $H \in \mathcal{H}$ and $G \in \mathcal{G}$ count the number of homomorphism from H to G, i.e. calculate $hom(H \to G)$.*

For the counting problem exist two well-known results in complexity theory. The first one fits more into the classical complexity schema of P and NP. Dyer and Greenill [7] showed that $\#HOM(\mathcal{U} \to \mathcal{G})$ is #P-complete in general except three special classes of graphs. Also for parameterized complexity exists a characterization of the problem. The paper [6] by Dalmau and Jonsson provides the following parameterized characterization of the problem $\#HOM(\mathcal{H} \to \mathcal{U})$. The problem of counting homomorphism from a graph H to an arbitrary graph G is solvable in polynomial if H is of bounded treewidth. The paper of Roth and Wellnitz [19] also provides a significant result. They stated that every problem in $\#W[1]$, a counting equivalence to the class $W[1]$, can be translated into an equivalent graph homomorphism counting problem. More precisely given a problem $P \in \#W[1]$ there exist two classes of graphs $\mathcal{H}$ and $\mathcal{G}$ such that P is equivalent to counting homomorphism from a graph in $\mathcal{H}$ to a graph in $\mathcal{G}$. To give an example, the problem of counting all paths of given length in a graph lies in $\#W[1]$. An introduction to parameterized complexity can be found in [9].

In the end of this chapter, I want to clarify the differences between some notations. In this section, we looked at the problems of deciding and counting graph homomorphisms. These problems are always written in capital letters, so they can be distinguished from the set of homomorphisms from one graph to another and its cardinality. There are three different notations.

1. $HOM(\mathcal{H} \to \mathcal{G})$ describes the problem of deciding whether a homomorphism from $\mathcal{H}$ to $\mathcal{G}$ exists where $\mathcal{H}$ and $\mathcal{G}$ are classes of graphs.

2. $Hom(H \to G)$ denotes the set of all homomorphisms from the graph H to the graph G.

3. $hom(H \to G)$ is the number of homomorphisms from H to G, i.e. $hom(H \to G) = |Hom(H \to G)|$.

3 Counting Homorphisms

The following chapter is about the main problem this paper deals with. In this problem, we do not only want to compute the number of homomorphisms for a single graph but for a whole set of graphs. These graphs will be characterized by a given tree decomposition. This chapter starts by giving a formal definition of the problem and continues by describing some properties of the set H_τ. After looking at this formal aspect of the problem, we will look at the running time of existing algorithms which can be simply repeated for all graphs in the set.

Given a tree decomposition $\tau = (T, \beta(p)_{p \in V(T)})$ we define H_τ to be the set of all graphs that may have τ as their tree decomposition. The problem this paper deals with is then defined as.

definition 3.1 (problem)**.** *Given a tree decomposition $\tau = (T, \beta(p)_{p \in V(T)})$, the set H_τ and a graph G, compute $hom(H \to G)$ for all $H \in H_\tau$.*

3.1 Characteristics of H_τ

To gain a better understanding we will observe the set H_τ in the following section. We assume that τ is a correct tree decomposition. Remember that the set H_τ is defined as the set of all graphs that may have τ as their tree decomposition. Hence we can observe some restrictions on the graphs of H_τ. Let us look at an arbitrary graph H of H_τ. Which properties does this graph have?

The most obvious property of H regards the set of vertices of H. Recalling the definition 2.4 of tree decompositions, especially the first condition, we can easily see that $V(H)$ must equal the union of all bags of τ. Since all graphs in H_τ have the exact same set of vertices, we will denote this set by V_τ. Formally stated

$$V(H) = V_\tau := \bigcup_{p \in V(T)} \beta(p) \tag{3.1}$$

The second condition of definition 2.4 restricts the set of possible edges the graph H may have. This condition requires that the following implication holds $[\{u, v\} \in E(H)] \implies [\exists p \in V(T) : u \in \beta(p) \wedge v \in \beta(p)]$. Put into words this means that an edge $\{u, v\}$ is only allowed if there exists a node $p \in V(T)$ with its bag $\beta(p)$ containing both u and v. Since the set of possible edges is equal for every graph $H \in H_\tau$ and only depends on τ we will denote this set by E_τ.

$$E_\tau := \{\{u, v\} \in V_\tau^2 : \exists p \in V(T) : u \in \beta(p) \wedge v \in \beta(p)\} \tag{3.2}$$

Note that condition 3 of definition 2.4 does not restrict the set of H_τ because it only effects the structure of the Tree T. Using the definition of V_τ and E_τ we can now define the set H_τ more formally.

$$H_\tau := \{V_\tau\} \times \mathcal{P}(E_\tau) \tag{3.3}$$

Since V_τ is fixed for each graph in H_τ, they differ only by their set of edges. Hence for a fixed τ we can index each graph $H \in H_\tau$ by its set of edges $E(H) \in \mathcal{P}(E_\tau)$. Let $\sigma_\tau : \mathcal{P}(E_\tau) \to H_\tau$ where the following holds

$$\sigma_\tau(E) \mapsto H = (V_\tau, E) \tag{3.4}$$

The index function always depends on the tree decomposition τ and gives as a boundary for the number of graphs in H_τ.

$$|H_\tau| = |\mathcal{P}(E_\tau)| = 2^{|E_\tau|} \leq 2^{|V_\tau|^2} \tag{3.5}$$

The index function provide a beautiful way of storing graphs from H_τ efficiently when E_τ is known. Since we are dealing with a potential set, we can represent every set $E \in \mathcal{P}(E_\tau)$ and therefore every $H \in H_\tau$ as a bit vector $\vec{H}$ of length $|E_\tau|$. Every edge in E_τ gets an unique index in $\{1, 2, \ldots, |E_\tau|\}$. We will denote the index of edge $e \in E_\tau$ with $i(e)$. The function σ_τ should then be defined as a function mapping from $\{0, 1\}^{|E_\tau|}$ to H_τ.

$$\sigma_\tau(\vec{H}) \mapsto H = (V_\tau, E) \iff \left[\bigwedge_{e \in E_\tau} (\vec{H}_{i(e)} = 1 \iff e \in E)\right] \tag{3.6}$$

We then call $\vec{H}$ the characteristic vector of the graph H. Note that σ_τ is obviously injective and $2^{|E_\tau|} = |\mathcal{P}(E_\tau)| = |H_\tau|$, therefore the function σ_τ is bijective. Hence each graph in H_τ can be described by a bit vector of length $|E_\tau|$ when the indexing of E_τ is known. It may confuse the reader why this representation of $H \in H_\tau$ will be described so early. The reason for this lies in the simplification of some proofs presented in the following sections. Since we have a bijection between H_τ and $\{0, 1\}^{|E_\tau|}$ we can argue about the cardinality of subsets of H_τ by using simple arguments for the corresponding bit vectors.

3.2 EXISTING METHODS

Existing methods are limited to counting homomorphism from one graph to another. To compute $hom(H \to G)$ for each $H \in H_\tau$ one can simply execute those algorithms for all graphs. Assuming the running time of an arbitrary algorithm for calculating $hom(H \to G)$ would be $O(T)$ then we could solve our problem in time $O(T \cdot |H_\tau|) = O(T \cdot 2^{|E_\tau|})$. Each of the following two sections cover an algorithm for computing $hom(H \to G)$

3.2.1 Brute Force Algorithm

Given an arbitrary tree decomposition τ, a graph $H \in H_\tau$ and an arbitrary graph G. The brute force algorithm will try out every mapping f from H to G and checks if f is a homomorphism by looping over each edge uv in $E(H)$ controlling whether $f(u)f(v) \in E(G)$ holds or not. The number of all mappings from H to G is $|V(G)|^{|V(H)|}$ since every vertex in H can be mapped to every vertex in G. Every mapping then has to be checked to be edge-preserving. Therefore the image of every edge in $E(H)$ can be controlled by checking the corresponding entry in the adjacent matrix of G. Assuming this entry look-up can be done in constant time, we will get an runtime of $O(|V(G)|^{|V(H)|} \cdot |E(H)|)$ for calculating $hom(H \to G)$. The overall runtime would be $O(2^{|E_\tau|} \cdot |V(G)|^{|V(H)|} \cdot |E(H)|)$ for solving the problem from definition 3.1.

3.2.2 DP of Díaz, Serna and Thilikos

In this section, we are going to discover the dynamic programming algorithm[8] of Díaz, Serna and Thilikos. It will be described in more detail since the algorithm presented in this work is just a variation of it and therefore it is important to understand the details. The following theorem is one of the main results in the paper[8].

Theorem 3.2.1. *Given a graph H with $h = |V(H)|$, a nice tree decomposition $\tau = (T, \{\beta(p)\}_{p \in V(T)})$ of H with width k and a stingy ordering $u_1, \ldots, u_m$ of the nodes in $V(T)$ then, there is an algorithm that computes $hom(H \to G)$ in $O(hn^{k+1} \cdot \min\{k, n\})$ steps using $O(n^{k+1} \cdot \log h)$ additional space, where $n = |V(G)|$.*

We assume that the graph H with its tree decomposition $\tau = (T, \{\beta(p)\}_{p \in V(T)})$ and a stingy ordering $u_1, \ldots, u_m$ is given and we want to compute $hom(H \to G)$ for an arbitrary graph G. By following a stingy ordering the algorithm successively fills out the table $I_p(\phi)$ which saves the number of extending homomorphisms for each node $p \in V(T)$ and each mapping $\phi \in F_p$. The set F_p contains all mappings from $\beta(p)$ to $V(G)$, i.e. $F_p := \{\phi : \beta(p) \to V(G)\}$.

definition 3.2. *Given a node $p \in V(T)$ and a mapping $\phi \in F_p$ the table $I_p(\phi)$ is defined as*

$$I_p(\phi) = |\{\theta \in Hom(H[\alpha(p)] \to G) : \theta|_{\beta(p)} = \phi\}| \tag{3.7}$$

where $\theta|_S$ describes the restriction of the function θ to S, i.e.

$$\theta|_S = \{(v, a) \in \theta : v \in S\} \tag{3.8}$$

The entries of $I_p(\phi)$ are computed step by step beginning at the leaves of the tree decomposition τ. Each type of node will be handled differently. We will now observe each type and discuss the behaviour encountering a node of this type beginning with leaves. After the entries of a node have been computed the data corresponding to all of its child nodes will be deleted, since they are only needed once. Pseudocode is also provided later.

Let $p \in V(T)$ be a leaf of the tree decomposition τ and v the unique vertex of $\beta(p)$. For setting the entries we have to distinguish between two possible cases. If the edge $\{v, v\}$ is not contained in $E(H)$, we can simply set $I_p((v, a)) = 1$ for each $a \in V(G)$. The only extending homomorphism of (v, a) is the mapping itself. Hence we can set the entry without further checking. In the second case the edge $\{v, v\}$ is an edge of H. Here we have to check if the vertex a, the image of v, also has a self loop. Hence we set $I_p((v, a)) = 1$ if and only if $\{a, a\} \in E(G)$ for each $a \in V(G)$.

Let p be an introduce node, q its child node and v be the unique vertex been introduced at p. The set S_q is defined as $S_q = \{u \in \beta(q) : \{u, v\} \in E(H[\alpha(p)])\}$. Which is the set of all vertices in $\beta(p)$ adjacent to v. The introduced vertex v has to be mapped on a vertex $a \in V(G)$ such that all vertices in S_q are mapped to adjacent vertices of a. Therefore the algorithm needs to check if $\{a, \phi(u)\} \in E(G)$ holds for ever $u \in S_q$. Calculating the entry itself after checking this condition is easy. The algorihtm just has to set the new entry to the old one, i.e $I_p(\phi \cup \{(v, a)\}) = I_q(\phi)$. The algorithm ϕ will be extended by the introduced vertex. This does not change the amount of extending homomorphisms in the rest of $\alpha(p)$.

Let p be a forget node, q its child node and v be the unique vertex been forgotten at p. By removing one vertex the number of mappings in F_p will be smaller than those in F_q. Therefore some mappings $\phi : \beta(q) \to V(G)$ are resulting in the same mapping when v has been removed. The entry $I_p(\phi)$ will then be set to $I_p(\phi) = \sum_{a \in V(G)} I_q(\phi \cup \{(v, a)\})$.

Let p be a join node with its children q_1 and q_2. Since $\beta(p) = \beta(q_1) = \beta(q_2)$ holds, we also know that $F_p = F_{q_1} = F_{q_2}$. The graph $H[\alpha(p)]$ is the union of $H\alpha(q_1)$ and $H[\alpha(q_2)]$. Since all edges these three (sub-)graphs have in common are contained $\beta(p) = \beta(q_1) = \beta(q_2)$ every mapping in F_p can be described as a combination of a mapping from F_{q_1} and a mapping from F_{q_2}. Hence we have to compute $I_p(\phi) = I_{q_1}(\phi) \cdot I_{q_2}(\phi)$ for every $\phi \in F_p$.

The complexity result can be obtained by the following short analysis. The number of entries that must be computed at each node p is $|F_p|$. The cardinality of F_p is given by $|V(G)|^{|\beta(p)|}$. Since the tree decomposition is of width k we can bound this term by $|V(G)|^{k+1}$.

The computation of one entry for leaf, forget or join node takes time $O(n)$. For introduce nodes the computation time depends on the representation of $N_p(v)$. Therefore the running time is $O(\min\{n,k\})$.

Hence the complexity follows. Given a nice tree decomposition τ The numbers of homomorphisms $hom(H \to G)$ for each graph in H_τ can then be computed in time $O(2^{|E_\tau|} \cdot hn^{k+1} \cdot \min\{k,n\})$. In comparison to the brute force algorithm, this means an enormous speed-up since bounding the width of τ will lead to polynomial time needed for each graph.

For each node $I_p(\phi)$ could have at most $n^{|\beta(p)|}$ entries, which equals the number of possible mappings from $\beta(p)$ to $V(G)$. Since $|\beta(p)|$ is bounded by $k+1$, the number of entries per node is bounded by n^{k+1}. By definition 2.8 of the stingy ordering we know, that the data of at most $\log h + 1$ nodes have to be available at the same time. Hence the additional space needed is $O(n^{k+1} \cdot \log h)$.

Algorithm 3.1: Counting graph homomorphisms

```
1   input: Graphs H and G,
2   a nice tree decomposition τ = (T, {β(p)}_{p∈V(T)}) of H
3   and a stingy ordering U = u_1, ..., u_m of V(T)
4   output: hom(H → G)
5
6   for i = 1 to n
7       set p = u_i
8
9       if p is a leaf with β(p) = {v}
10          for all a ∈ V(G) set I_p((v,a)) = 1 if [{v,v} ∈ V(H) ⟹ {a,a} ∈ V(G)]
11
12      if p is an introduce node
13          let q be its unique child and {v} ∈ β(p)\β(q)
14          set N_p(v) = {u ∈ β(p) | {u,v} ∈ E}
15
16          for all φ ∈ F_q and a ∈ V(H)
17              if ∀_{u∈N_p(v)} {φ(u), a} ∈ E(G)
18                  set I_p(φ ∪ {(v,a)}) = I_q(φ)
19              else
20                  set I_p(φ ∪ {(v,a)}) = 0
21
22          erase information on node q
23
24      if p is a forget node
25          let q be its unique child and {v} ∈ β(q)\β(p)
26          for all φ ∈ F_p set I_p(φ) = Σ_{a∈V(H)} I_q(φ ∪ {(v,a)})
27
28          erase information on node q
```

```
29
30              if p is a join node
31                  let q_1 and q_2 be its children
32                  for all φ ∈ F_p set I_p(φ) = I_{q_1}(φ) · I_{q_2}(φ)
33
34                  erase information on nodes q_1 and q_2
35
36          return I_{r(T)}(∅)
```

3.3 Generating possible edges

Before using one of the algorithms mentioned above, the graphs have to be generated. Since the set of vertices is fixed, only the set of possible edges E_τ has to be generated. In the following, a simple algorithm for calculating all possible edges will be described.

Algorithm 3.2: Computing possible edges

```
1       input: A nice tree decomposition τ = (T, {β(p)}_{p∈V(T)})
2       and a stingy ordering U = u_1, ..., u_m of V(T)
3       output: E_τ
4
5       set E_τ = ∅
6
7       for i = 1 to m
8           set p = u_i
9           set E'_τ = E_τ ∪ β(p)^2
10          set E_τ = E'_τ
11
12      return E_τ
```

This algorithm simply unions the Cartesian product of all bags. Therefore the running time is bounded by $O(|V(T)| \cdot |V_\tau|^2)$ since checking if an edge already has been added or not only needs time $O(|V_\tau|)$ by using an adjacency matrix.

To prove correctness, assume that there exists an edge $uv \in E_\tau$ which has not been found by the algorithm. Then u and v cannot be contained in one bag at once. Otherwise, algorithm 3.2 would have found it. But that is a contradiction to the definition of E_τ since there must be a bag containing both u and v.

4 Equivalent Entries

This chapter contains a modification of the dynamic program of Díaz, Serna and Thilikos presented in the previous chapter. The chapter begins with a simple algorithm for calculating a stingy ordering. It continues by presenting the theoretical base for reusing entries of the dynamic program. After that, the computational steps for each type of node will be described in more detail. The chapter ends with a short analysis of the time and space complexity.

4.1 Computing a Stingy Ordering

This section provides a recursive algorithm proofing lemma 4.1. We define $b(p)$ for $p \in V(T)$ as the number of nodes with degree exactly two in the subtree T_p. The concatenation of two orderings will be denoted by $\circ$.

Lemma 4.1 (Computing a stingy ordering)**.** *Given a binary tree T with n nodes. A stingy ordering of T can be computed in time $O(n)$ using algorithm 4.1.*

Proof. The time complexity of this algorithm can be analysed quickly by noting that each node in T will only be once the root of a subtree and hence observed only once. The number of operations needed per node is constant. Hence the time complexity follows. The correctness of the algorithm will be proven by induction. Therefore we show that stingy_ordering(p) returns a correct stingy ordering of T_p for all $p \in V(T)$ if it already returned a correct stingy ordering of its child nodes.

base case: Assuming that p is a leaf with degree zero. The algorithm will just return p as a stingy ordering. Conditions 1,2,3 of definition 2.8 are easy to see. Condition 4 also holds since $b(p) = 0$ and $\log(b(p)) + 1 = 1$.

induction step: Assuming that stingy_ordering correctly computes a stingy ordering for all children of p. Condition 1 follows the observation that stingy_ordering first returns an entry for the stingy ordering when reaching a leaf. Hence the first node of the ordering must be a leaf.

Since the node p will always be put to the back of the stingy ordering, p will always be behind its children and condition 3 is satisfied.

At last we have to proof that condition 4 holds. For a given order $U = u_1, \ldots, u_m$ of tree nodes we define $cw(U) = \max_j |\{u \in U | \text{p(u)} > \text{j}\}|$, where $p(u)$ denotes the position of the parent

of u in the ordering U. Note that cw stands for cut-width because the definition of $cw()$ equals the more general definition of cut-width for graphs, when imagine the order as a line graph with edges between adjacent nodes. Here cw was defined to better describe condition 4. Condition 4 holds as long $cw(p) \leq \log(b) + 1$ is given. For the proof we distinguish between the following cases.

case 1 : Let p be a leaf then the condition holds as shown in the base case.

case 2 : Let p be a node with degree exactly one and let $u_1, \ldots, u_m$ the order returned by stingy_ordering(p). Let q be the unique child of p. Since stingy_ordering(q) already returned a correct stingy ordering we know that condition 4 holds. When attaching p to the end of stingy_ordering(q) the cut-width stays the same since q is directly before p and there cannot be any node in stingy_ordering(q) for which holds that its parent comes after p.

case 3 : Let p be a node with degree exactly two. Assume $b(q_1) \geq b(q_2)$. The cut-width of $stingy\_ordering(q_1)$ will not be increased by the new order but the cut-width of $stingy\_ordering(q_2)$ will be increased by the edge between q_1 and p. This argument is visually shown in figure 4.1.

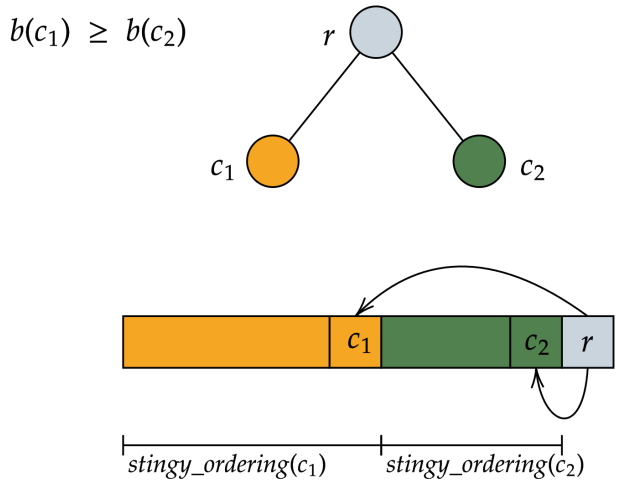

Figure 4.1: A visualization of the proof of case 3

Therefore holds the following equation.

$$cw(p) \leq \max\{cw(q_1), cw(q_2) + 1\} \tag{4.1}$$

Now we have to distinguish between two more cases. **case 3.1 :** $cw(q_1) >= cw(q_2)$. Then it follows that.

$$cw(p) = cw(q) \leq \log\left(b(q_1)\right) + 1 \leq \log\left(b(p)\right) + 1 \tag{4.2}$$

case 3.2 : $cw(q_1) < cw(q_2)$. Note that by assumption $b(q_2) < \frac{1}{2} \cdot b(p)$ since $b(q_1) \leq b(q_2)$ and $b(p) = b(q_1) + b(q_2) + 1$. It follows

$$cw(p) = cw(q_2) + 1 \leq \log\left(b(q_2)\right) + 2 = \log\left(\frac{1}{2}b(p)\right) \leq \log\left(b(p)\right) + 1 \tag{4.3}$$

Which results in condition 3 that states that $cw(p) \leq \log(b(p)) + 1$.

□

Algorithm 4.1: Computing a stingy ordering

```
1  stingy_ordering (T)
2  input: a rooted binary tree T with root r
3  output: a stingy ordering of u_1,...,u_n of V(T)
4      let c = deg(r)
5      if c == 0
6          return r
7      if c == 1
8          let q be the unique child of r
9          return stingy_ordering(T_q) ∘ r
10     if c == 2
11         let q_1 and q_2 be the children of r assuming that b(q_1) ≥ b(q_2)
12         return stingy_ordering(q_1) ∘ stingy_ordering(q_2) ∘ r
```

4.2 Equivalent Entries

The following section lays the theoretical foundation for adjusting the algorithm 3.1 to the problem stated in definition 3.1. This modification of the dynamic program is based on the following observation. Given two graphs H' and H'' from the set H_τ, an arbitrary graph G and an arbitrary node $p \in V(T)$. If H' and H'' are equal regarding the vertices in $\alpha(p)$ the entries $I_p(\phi)$ are equal for both graphs and all mappings ϕ. Hence those entries have to be computed only once and may speed up the computation. First of all the table of algorithm 3.1 has to be modified. The following definition describes a first, simple approach for extending the table. Therefore recall the definitions of section 3.2.2.

definition 4.1. *For each graph $H \in H_\tau$, node $p \in V(T)$ and mapping $\phi \in F_p$ the entries of the table are defined by the following equation.*

$$I_{H,p}(\phi) = |\{\theta \in Hom(H[\alpha(p)] \to G) : \theta|_{\beta(p)} = \phi\}| \tag{4.4}$$

Note that the only difference between the table of definition 3.2 and definition 4.4 is that we extend it by one dimension, namely the graph $H \in H_\tau$. In other words $I_{H,p}(\phi)$ is the number of extending homomorphisms of ϕ in $H[\alpha(p)]$.

The number of entries in $I_{H,p}(\phi)$ increases to $|H_\tau| \cdot |V(G)|^{k+1} \cdot |V(T)|$ which is equal to $2^{|E_\tau|} \cdot |V(G)|^{k+1} \cdot |V(T)|$.

The following lemma is the foundation of the algorithm.

Lemma 4.2. *Let H' and H'' be graphs in H_τ and let G be an arbitrary graph. Then holds $I_{H',p}(\phi) = I_{H'',p}(\phi)$ if $H' \sim_{\alpha(p)} H''$ for $p \in V(T)$ and $\phi \in F_p$.*

Proof. Assuming that $H' \sim_{\alpha(p)} H''$, the set of homomorphisms from H' to G is equal to the set of homomorphisms from H'' to G. It is $Hom(H'[\alpha(p)] \to G) = Hom(H''[\alpha(p)] \to G)$. Hence the following equation holds.

$$I_{H',p}(\phi) = |\{\theta \in Hom(H'[\alpha(p)] \to G) : \theta|_{\beta(p)} = \phi\}| \tag{4.5}$$

$$= |\{\theta \in Hom(H''[\alpha(p)] \to G) : \theta|_{\beta(p)} = \phi\}| = I_{H'',p}(\phi) \tag{4.6}$$

□

This lemma gives us a tool for reducing the number of computations. The following corollary results immediately.

Corollary 4.2.0.1. *For a graph $H \in H_\tau$ and a node $p \in V(T)$ holds the following. Every graph $H' \in [H]_{\alpha(p)}$ has the following property.*

$$I_{H',p}(\phi) = I_{H,p}(\phi) \tag{4.7}$$

Here $[H]_{\alpha(p)}$ denotes the class of all graphs $H' \in H_\tau$ for which $H \sim_{\alpha(p)} H'$ holds. For better readability the notation does not contain τ, but it should be clear by context which tree decomposition τ is meant.

Corollary 4.2.0.1 leads to many identical entries which will be computed unnecessarily. For this reason, we will reduce the size of the table to only those entries which are needed.

4.3 Reduced Table

If $H' \sim_{\alpha(p)} H''$ holds for two graphs H' and H'' both graphs have the same subgraph induced by the vertices of $\alpha(p)$. Hence the equivalence class of a graph H regarding a node p can be represented by a single (sub-)graph. For a node $p \in V(T)$ and a subset of possible edges $E \subseteq E_\tau[\alpha(p)]$ we will define the graph $S_E^{[p]}$ as follows.

$$S_E^{[p]} = (\alpha(p), E) \tag{4.8}$$

The graph $S_E^{[p]}$ is then the representative of the equivalence class $A_E^{[p]}$. That is

$$A_E^{[p]} = [S_E^{[p]}]_{\alpha(p)} \tag{4.9}$$

For a graph $H \in H_\tau$ and a node $p \in V(T)$ we can easily determine which equivalence class contains it by looking at which subgraph $S_E^{[p]}$ is equal to H when it is restricted to the vertices in $\alpha(p)$. But since every graph in $A_E^{[p]}$ has the same entry we should only concentrate on calculating the values for the representatives $S_E^{[p]}$.

For a node $p \in V(T)$ we will denote the set of all equivalence classes by $A^{[p]}$ and the set of all representatives by $S^{[p]}$.

We will now modify the table $I_{H,p}(\phi)$ again to reduce the number of entries. We see that every equivalence class can be characterized by a single graph that is restricted to the vertices in $\alpha(p)$. Hence we just need one entry for each $A_E^{[p]}$ instead of one entry for each $H \in H_\tau$. This reduces the size of entries per node $p \in V(T)$ from $O(2^{|E_\tau|} \cdot |V(G)|^{k+1})$ to $O(2^{|E_\tau[\alpha(p)]|} \cdot |V(G)|^{k+1})$. We can now define a new table that considers graphs of $S^{[p]}$.

$$I_{S_E^{[p]},p}(\phi) = |\{\theta \in Hom(S_E^{[p]} \to G) : \theta|_{\beta(p)} = \phi\}| \tag{4.10}$$

Since the vertex set is fixed we can describe $S_E^{[p]}$ clearly by its set of Edges E. This simplifies the table to the following definition.

$$I_{E,p}(\phi) = |\{\theta \in Hom(S_E^{[p]} \to G) : \theta|_{\beta(p)} = \phi\}| \tag{4.11}$$

with $p \in V(T)$ and $E \subseteq E_\tau[\alpha(p)]$. The difficulty in counting the entries now lies in finding the correct entries in the tables of the child nodes. Note that the equivalence classes also provide a partition of H_τ which can get finer at each node. So it makes sense to understand the task of finding the correct entries as a partition refinement task.

Another point of view may be, that we are building the set H_τ bottom-up step by step while moving the tree upward. Hence we could also build the set E_τ step by step while walking up the tree. In the beginning - in a leaf - we have the loosest partition of H_τ which only consist of two class; the graph with one vertex and no edge and the graph with one vertex and a self-loop. These classes becomes finer until we have for every graph in H_τ a single class in the root node.

4.4 Computational Steps

This section is about the details of modifying algorithm 3.1. The computational frame stays the same as the specific calculations for the entries. One main difficulty lies in addressing the right entries of the dynamic table by choosing the correct edge set. The right choice of table entries and the computation itself will be described in the following for each node type separately.

4.4.1 Leaf

Let p be a leaf node and let v be the unique vertex in $\beta(p)$. The only possible edge contained in $E_\tau[\alpha(p)]$ could be the self loop of v. Therefore we have to set two entries for each mapping $\phi : \beta(p) \to V(G)$. One for graphs containing the edge $\{v, v\}$ and one for graphs without edges. The second one is really easy since for every mapping the entry could be set to one, i. e. we set for all $\phi : \beta(p) \to V(G)$

$$I_{\emptyset,p}(\phi) = 1 \tag{4.12}$$

The first case is harder because we have to check if the image of v also contains a self-loop. This means for all $\phi : \beta(p) \to V(G)$ we set

$$I_{\{vv\},p}(\phi) = 1 \tag{4.13}$$

if $\{\phi(v), \phi(v)\}$ is contained in G. Otherwise the entry is set to zero.

4.4.2 Introduce Node

Let p be a introduce node with its unique child q and the introduced vertex v. We define the set of edges E_p as the subset of possible edges in $\alpha(p)$ which are incident to v.

$$E_p = \{e \in E_\tau[\alpha(p)] | v \in e\} = \{e \in E_\tau[\beta(p)] | v \in e\} \tag{4.14}$$

This edges will be called introduced edges at p. By adding the introducing the edges E_p every set $E \in E_\tau[\alpha(q)]$ could be extended by $2^{|E_p|}$ possible sets of new edges. In other words the set H_τ will be split into $2^{|E_p|}$ times more classes regarding the vertices in $\alpha(p)$.

Given $E \subseteq E_\tau[\alpha(p)]$ finding the corresponding set of edges E' in $E_\tau[\alpha(p)]$ can be done easily by restricting E to the the edges of $E_\tau[\alpha(p)]$, i.e. $E' = E \cup E_\tau[\alpha(p)]$. This is possible because the graph $S^{[q]}_{E \cap E_\tau[\alpha(q)]}$ is a subgraph of $S^{[p]}_E$. Note that for $E \in E_\tau[\alpha(p)]$ holds $E \backslash E_p = E \cap E_\tau[\alpha(q)]$.

Then we can compute $I_{E,p}(\phi \cup \{(v,a)\})$ for all $a \in V(G)$ the following way. If $\forall_{u \in N_p(v)} \{\phi(u), a\} \in E(G)$ holds, we set

$$I_{E,p}(\phi \cup \{(v,a)\}) = I_{E \cap E_\tau[q],p}(\phi) \tag{4.15}$$

otherwise $I_{E,p}(\phi \cup \{(v,a)\})$ will be set to zero.

4.4.3 Forget Node

Let p be a forget node with its unique child q. Let $v \in V(G)$ be the vertex that has been forgotten at p. Since only one vertex will be forgotten at p we know that $\alpha(p)$ equals $\alpha(q)$. Which means that also the equivalence classes are identical, because no new possible edge has been added. Hence $E_\tau[\alpha(p)] = E_\tau[\alpha(q)]$ and we can access the equivalence classes with same subset of edges as its index. For $\phi \in F_p$ and $E \subseteq E_\tau[\alpha(p)]$ we set

$$I_{E,p}(\phi) = \sum_{a \in V(G)} I_{E,q}(\phi \cup \{(v,a)\}) \tag{4.16}$$

4.4.4 Join node

Let p be a join node with its children q_1 and q_2. To compute $I_{E,p}(\phi)$ we need to know which graph in $S^{[q_1]}$ and which in $S^{[q_2]}$ can be combined to $S_E^{[p]}$. Since both graph must be identical regarding the edges in $E_\tau[\alpha(q_1)] \cap E_\tau[\alpha(q_2)]$ we can compute the entry the following way.

$$I_{E,p}(\phi) = I_{E \cap E_\tau[\alpha(q_1)],q_1}(\phi) \cdot I_{E \cap E_\tau[\alpha(q_2)],q_2}(\phi) \tag{4.17}$$

In other words, we are combining each extending homomorphism of ϕ in $\alpha(q_1)$ with each extending homomorphism of ϕ in $\alpha(q_2)$. We access this numbers through the subgraphs of $S_E^{[p]}$ in $\alpha(q_1)$ and $\alpha(q_2)$. This works because the following two equations hold.

- $S_E^{[p]}[\alpha(q_1)] = S^{[p]}_{E \cap E_\tau[\alpha(q_1)]}$
- $S_E^{[p]}[\alpha(q_2)] = S^{[p]}_{E \cap E_\tau[\alpha(q_2)]}$

4.5 Algorithm and Analysis

It follows the pseudo-code of the modified dynamic program.

Algorithm 4.2: modified dynamic program

```
1  input: nice tree decomposition τ and a graph G
2  and a stingy ordering U = u_1, ..., u_m of V(T)
```

```
3   output: hom(H → G) for all H ∈ H_τ.
4
5   for i = 1 to n
6       set p = u_i
7
8       if p is a leaf with β(p) = {v}
9           for all a ∈ V(G)
10              set I_{∅,p}((v, a)) = 1
11              set I_{{v,v},p}((v, a)) = 1 if {a, a} ∈ V(G) and 0 otherwise
12
13      if p is an introduce node
14          let q be its unique child and {v} ∈ β(p)\β(q)
15
16          for all E ⊆ E_τ[α(p)]
17              set N_p(v) = {u ∈ β(p)|{u, v} ∈ E}
18
19              for all φ ∈ F_q and a ∈ V(H)
20                  if ∀_{u∈N_p(v)}{φ(u), a} ∈ E(G)
21                      set I_{E,p}(φ ∪ {(v, a)}) = I_{E∩E_τ[q],p}(φ)
22                  else
23                      set I_{E,p}(φ ∪ {(v, a)}) = 0
24
25          erase information on node q
26
27      if p is a forget node
28          let q be its unique child and {v} ∈ β(q)\β(p)
29          for all E ⊆ E_τ[α(p)]
30              for all φ ∈ F_p set I_{E,p}(φ) = Σ_{a∈V(H)} I_{E,q}(φ ∪ {(v, a)})
31
32          erase information on node q
33
34      if p is a join node
35          let q_1 and q_2 be its children
36          for all E ⊆ E_τ[α(p)]
37              for all φ ∈ F_p
38                  set I_{E,p}(φ) = I_{E∩E_τ[α(q_1)],q_1} · I_{E∩E_τ[α(q_2)],q_2}
39
40          erase information on nodes q_1 and q_2
41
42  return I_{E,p}(∅) for all E ⊆ E_τ
```

The number of homomorphisms from a graph $H \in H_\tau$ to a graph G can be found in entry $I_{E(H),p}(\emptyset)$.

4.5.1 Correctness

The correctness follows mainly by the proof of correctness provided for the algorithm 3.1. Since the calculations themselves are similar to those of section3.2.2 its correctness can be assumed. Beyond this, it is important to verify that always the right entries will be accessed by the indexation

over the edge sets. This property has already been shown in the previous section where all steps have been described in more detail. Hence the correctness of the algorithm is shown.

4.5.2 Time

Lemma 4.3. *The algorithm 4.2 runs in time*

$$O\left(|V(G)| \cdot |V(G)|^{width(\tau)+1} \cdot \sum_{p \in v(T)} 2^{|E[\alpha(p)]|}\right) \tag{4.18}$$

Proof. Since the algorithm is a dynamic program the running time can be bounded by the number of entries multiplied with the time need to compute one entry. The time needed for computing a single entry can be bounded by $O(|V(G)|)$. For each node $p \in V(T)$ there are exactly $2^{|E[\alpha(p)]|}$ subsets of possible edges regarding the vertices in $\alpha(p)$. Furthermore there are $|V(G)|^{|\beta(p)|} \leq |V(G)|^{width(\tau)+1}$ mappings from $\beta(p)$ to $V(G)$. For each combination of edges and mapping we have to compute one entry. Resulting in a running time of $O(|V(G)| \cdot |V(G)|^{width(\tau)+1} \cdot 2^{|E[\alpha(p)]|})$ per node. Summing up this time over all nodes results in the following overall running time.

$$O\left(\sum_{p \in v(T)} |V(G)| \cdot |V(G)|^{width(\tau)+1} \cdot 2^{|E[\alpha(p)]|}\right) \tag{4.19}$$

At this point, it is also possible to extract the term $2^{|E[\alpha(p)]|}$ out of the sum and bound it by $2^{|E_\tau|}$. However, this would result in the same running time as in the analysis of the algorithm 3.1. But the hypothetical speed up comes from the fact that $2^{|E[\alpha(p)]|}$ is strictly smaller than $2^{|E_\tau|}$ in a significant amount of nodes in T. Therefore we remain with this more fine-grained analysis of the running time and conclude the stated running time. □

4.5.3 Space

Since the algorithm computes the homomorphism number for all graphs in parallel the amount of additional space increases rapidly relative to the number of edges in E_τ. The following lemma states this more formally.

Lemma 4.4. *The algorithm 4.2 uses additional space*

$$O\left(2^{|E_\tau|} \cdot n^{k+1} \cdot \log h\right) \tag{4.20}$$

where h equals $|V_\tau|$.

Proof. Looking at some node $p \in V(T)$ we need one entry for each combination of mapping $\phi : \beta(p) \to V(G)$ and edge set $E \subseteq E_\tau[\alpha(p)]$. Hence the number of entries per node is bounded by $2^{|E_\tau|} \cdot n^{k+1}$ because the number of subsets of $E_\tau[\alpha(p)]$ is $2^{|E_\tau[\alpha(p)]|}$ which is obviously at most $2^{|E_\tau|}$. And by condition 4 of definition 2.8 we know that the entries of at most $\log h + 1$ nodes has to be accessible at the same time. Therefore the space complexity follows.

□

5 Implementation

The implementation of the algorithms presented in the previous chapters was written in Rust and can be found here. Rust is a multi-paradigm programming language focusing on memory safety and clean code. The Rust compiler forces the programmer to write safe code by enforcing simple ownership rules of data by its borrow checker. The syntax of Rust is strongly orientated by C++. Further information and a nice introduction can be found in the Rust book and the comprehensive documentation.

The project mainly consists of four components. First of all, there is a file handler which can import nice tree decomposition and graphs. The format of these files will be described in the following section. The second component is a graph generation module. This single module contains methods for generating E_τ and H_τ for a given τ. These methods are used by the third and main component, the implementation of homomorphism counting algorithms mentioned in chapter 4. The last component consists of several experiments tracing the running time of the algorithms and comparing them. The following diagram gives an overview of the modules and their dependencies.

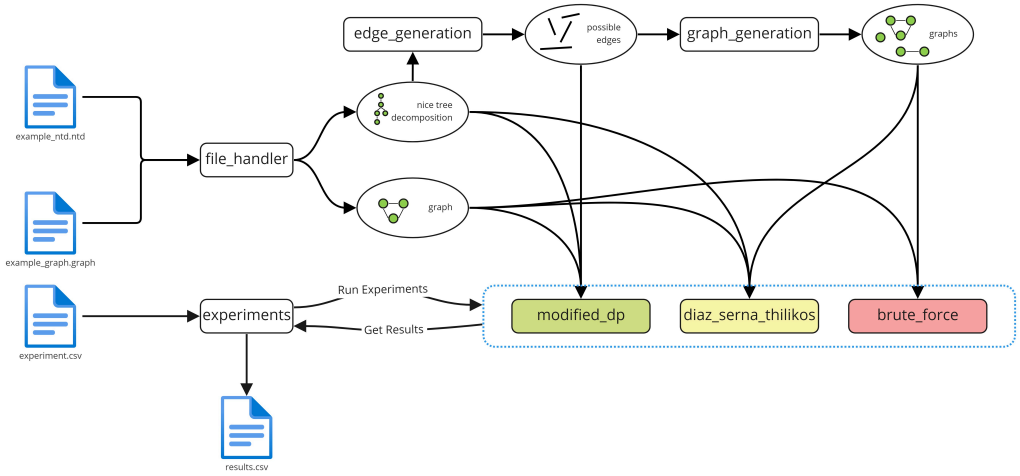

Figure 5.1: An overview of the project structure

5.1 FILE FORMATS

The file handler supports three different file formats. One format for nice tree decompositions and two formats for graphs. The format representing nice tree decomposition is a modification of the tree decomposition format for the PACE challenge. The first graph format with the file extension **.graph** is a simplification of the METIS [13] graph format. The other file format for graphs has the **.gr** extension and is the graph format originally used by the DIMACS challenge. The following three subsections shortly explain those formats in more detail.

5.1.1 METIS

The first non-comment line of METIS files contains the number of vertices of the graph and the number of edges. Entries in this format are separated by the space character and lines by the character **\n**. Comment lines start with the symbol **%**. The following lines represent the adjacency of the vertices by listing the neighbours of the i-th vertex in the i-th line. Note that graphs which have been observed in this paper are undirected and therefore each edge is represented twice.

Format 5.1: METIS example

```
1 % A graph with 5 vertices and 3 edges
2 5 3
3 % Here begins the list of neighbours of each vertex
4 4 5
5
6
7 % The following line expresses the neighbours (1 and 5) of vertex 4
8 1 5
9 1 4
```

The example above represents the graph in figure 5.2.

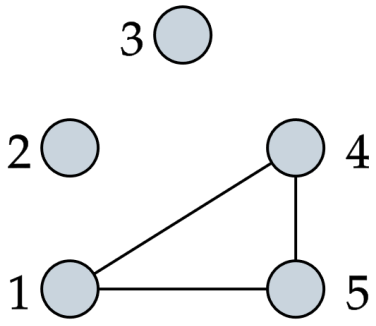

Figure 5.2: Graph represented by both formats

5.1.2 DIMACS

The DIMACS format starts with a description line where the problem descriptor, the number of vertices and the number of edges are listed after a foregoing **p**. Each argument is separated by a space and each line by **\n**. Comment lines start with a **c**. Each next line represents a single edge by containing both incident vertices. Here we have to list each edge only once. The following example represents the graph in figure 5.2. Note that **hc** stands only for homomorphism counting which is an exemplary description of the problem.

Format 5.2: dimacs example

```
1 % A graph with 5 vertices and 3 edges
2 p hc 5 3
3 % Here begins the listing of all edges
4 1 4
5 1 5
6 4 5
```

5.1.3 NTD

Nice tree decomposition files have the file extension **.ntd** and are structured the following way. The file starts with a description line containing the number of nodes, the maximum bag size and the number of vertices of the original graph. The description line is marked with an **s** at the beginning of the line. Each argument is separated by a space and lines are separated by **\n**. Comment lines begin with the character **#**. The next lines contain information about single nodes. These node description lines start with an **n** followed by the node number and a character describing the node type. The following characters are used to describe node types.

- **l** : leaf node
- **i** : introduce node
- **f** : forget node
- **j** : join node

The node type character is followed by the vertices represented by integers contained in the bag of the node.

After listing the nodes, the adjacency relation of the nodes has to be described. These adjacency lines are of simple form. They start with a foregoing **a** and continue with two nodes whereas the first node is the parent of the second node. The following example describes the nice tree decomposition in figure 5.3.

Format 5.3: ntd example

```
# a simple nice tree decomposition with 4 nodes and of width 1
s 4 2 2
# The following lines describe the nodes
n 1 l 1
n 2 i 1 2
n 3 f 2
n 4 f
# The following lines describe the adjacency of nodes
a 2 1
a 3 2
a 4 3
```

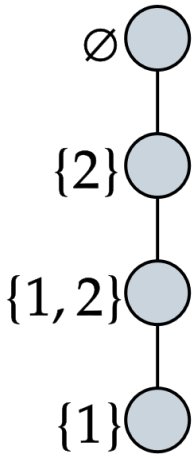

Figure 5.3: Nice Tree decomposition described in example 5.3

5.2 Internal Representation

This section describes the internal representation of graphs and nice tree decompositions. This knowledge is maybe useful for further practical improvements.

5.2.1 Graph

Graphs are internally represented by adjacency matrices provided by the external crate of petgraph. This Rust library provides some basic graph data structures. The implementation uses the graph

type MatrixGraph to ensure edge checking in constant time. Since graphs are static after creation the running time for vertex and edge manipulations is asymptotically uninteresting.

5.2.2 Nice Tree Decomposition

The internal representation of nice tree decompositions has been written from scratch and can be found in the **tree_decomposition.rs** file. This representation mainly consists of a tree structure realized with adjacency lists and a hashmap that maps each node to its containing data such as the bag and the node type. The bags of nodes are represented by hash sets of unsigned integers. The structure **NiceTreeDecomposition** also stores a stingy ordering, which will be computed during the Construction of the object.

5.3 Díaz, Serna and Thilikos

Algorithm 3.1 has been implemented in the file **diaz_serna_thilikos.rs** which also contains **DP-Data**. This structure stores and organizes all necessary data for the dynamic program. The entries of its table are stored inside two nested hash maps.

The implementation of the algorithm can be found in the **diaz_serna_thilikos.rs** file. The algorithm itself gets three arguments

1. a graph H
2. a nice tree decomposition τ of H
3. a graph G

and returns $hom(H \to G)$. There also exists another version of the algorithm which takes only two arguments: A nice tree decomposition τ and a graph G. This version is used for testing and includes the graph generation step. Therefore it returns $hom(H \to G)$ for all $H \in H_\tau$.

5.3.1 Integer Representation

An interesting part of the implementation is the usage of integer functions also used in a C++ implementation by Nielsen, Clausen and Reeve of algorithm 3.1 and have been described in their paper[15]. Integer functions are a compact way of representing a mappings $f : A \to B$ as $|A|$-digit numbers in base $|B|$. This representation is used to describe mappings from a bag $\beta(p)$ to the vertices $V(G)$ of graph G. There are three methods defined for integer functions: **apply**, **extend** and **reduce**. The implementation of these methods can be found in the **integer_functions.rs** file.

Let $p \in V(T)$ be an arbitrary node of the tree decomposition τ. define $k := |\beta(p)|$ and $n := V(G)$. Having an order of $\beta(p)$ gives each vertex in the bag a unique significance between 0 and $k-1$. Let f be a mapping from $\beta(p)$ to $V(G)$ and let $int(f)$ its integer representation. Then the i-th digit of $int(f)$ contains the value of the image of the vertex v in $\beta(p)$ with significance i. This requires also a partial order of vertices in $V(G)$. As an ordering the natural order of the vertex indices may be available. Lets explain the three methods in more detail by the following example.

Let $k = 3$ with $\beta(p) = 4, 2, 9$. Using the natural ordering of the vertices we conclude that the vertex 2 has significance 0, the vertex 4 has significance 1 and the vertex 9 has significance 2. Furthermore let $n = 4$ with $V(G) = 0, 1, 2, 3$. The mapping $f : \beta(p) \to V(G)$ with $f(2) = 3, f(4) = 1, f(9) = 0$ can then be represented as the following integer

$$int(f) = 3 \cdot 3^0 + 1 \cdot 3^1 + 0 \cdot 3^2 = 3 + 9 = 12 \tag{5.1}$$

For better readability, the integer representation of a function will simply be denoted by the function itself. Given the basis n, the integer representation f of the mapping f and the significance s the **apply(n,f,s)** function returns the digit d with significance s of f. The digit d can be computed by the following equation which basically shifts all digits smaller than s to the right and then takes the last digit.

$$d = \lfloor f/n^s \rfloor \mod n \tag{5.2}$$

For additional given value v the **extend(n,f,s,v)** function inserts the value v to digit s and shifts all digit with significance higher than s one to the left. This is realised by the following calculation. First separate the digits r with significance smaller than s by $r = f \mod n^s$. These digits remain in their position. The digits l with the significance of at least s can then be obtained by $f - r$. All digits in l have to be shifted one to the left by multiplying them with the basis n. Adding $l \cdot n$ and r together would result in a number where the digit with significance s would have value 0. By multiplying v with n^s the value v will be put to the s-th digit. Summing up the new mapping f^* will be computed by the following formula.

$$f^* = n \cdot l + v \cdot n^s \cdot r \tag{5.3}$$

Given the basis n, a mapping f and the significance s, the method **reduce(n,f,s)** removes the digit with significance s and shifts all digits with greater significance one to the right. The digits with significance smaller than s stay in place. Therefore we can compute $r = f \mod n^s$ and later add them to the new number. To get all digits with higher significance we can simply subtract

those digits of f which has significance at most s, i.e. $l = f - (f \mod n^{s+1})$. These digits have to be shifted one to the right by dividing them by n. For the new mapping f^* we get

$$f^* = l/n + r \tag{5.4}$$

5.4 Modified Dynamic Program

The implementation of the modified dynamic program is nearly identical to the implementation of algorithm 3.1. Mappings are also saved as integer functions and the computational steps themselves follow the same program logic. The implementation of the modified dynamic program can be found in the file **modified_dp.rs**. The function executing the modified dynamic program is called **modified_dp**, which takes two arguments: a tree decomposition τ and a graph G.

The main difference between both algorithms lies in the table that is used by the dynamic program. Since the table has been extended by edge sets, those have to be represented in some way. The following subsection describes the representation of edge sets.

5.4.1 Edge Representation

As mentioned in section 3.1 subsets of the possible edges can be represented as a bit vector. Therefore the program fixes an ordering of possible edges by giving each edge an index. Then each subset of edges can be represented as a vector of bits, where a bit is set to one if and only if the corresponding edge is contained in the set. The intersection and the union of two edge sets can then be realized by simple bitwise operations. The union of two sets can be realized by a bitwise **OR** and the intersection by a bitwise **AND**.

5.5 Test implementation

There are in general two types of tests that have been implemented in this project. First of all, we have unit tests checking the correctness of the implemented methods and structures. Those can be found in the file **unit_tests.rs** and have been implemented using Rust's unit test environment. This allows simple execution of the unit test by using the command **"cargo test"**. Then we have the performance experiments implemented in the file **experiments.rs**. The aim of the experiments is on one hand to compare the actual running time with the theoretical analysis. On the other hand, they should check if the modified dynamic program may be faster than the original one. These tests have been written by scratch and can be executed with the **run_experiment** function which takes a file as an argument. This file contains a binary matrix stating which combinations of nice tree decompositions and graphs should be tested. The results will then be written

into a CSV file containing the following columns: The names of the nice tree decomposition and the graph file, important parameters of both, 5 measurements of the original DP and 5 measurements of the modified DP with its means. Example 5.4 tests the following combinations. the file *ntd_bench_2.ntd* will be tested together with *randgraph_4_5.graph* but not with *randgraph_4_6.graph*, but *ntd_bench_2.ntd* will be tested with both.

Format 5.4: Experiment Matrix Example

```
1 ,randgraph_4_5.graph,randgraph_4_6.graph
2 ntd_bench_2.ntd,1,0
3 ntd_bench_3.ntd,1,1
```

6 Experiments and Summary

6.1 Experiments

6.1.1 Instances

Some test instances were created by hand but the majority of instances were created by generators. This section shortly explains the types of instances that were created by generators. The generators are all written in python and can be found in the **python** directory of the project.

The first generator can be found in the file **random_graphs.py**. This generator simply generates graphs with $n = 2^i$ vertices for some $i \in \mathbb{N}$ and randomly selects edges. Each graph is then written into a graph file represented in the METIS format. The file name contains the number of vertices and the number of edges.

Additional there are three generators for nice tree decompositions. The first generator constructs a nice tree decomposition τ with width $k-1$ and $2k$ nodes for a given parameter $k \in \mathbb{N}$ of the following form. Simply introduce $k-1$ vertices after the first vertex has been introduced in a leaf. Then forget the vertices one after another. The set H_τ therefore is the set of all subgraphs of the complete graph with k vertices and self-loops and therefore $|E_\tau| = \binom{n+1}{2} = \frac{n(n+1)}{2}$. These nice tree decompositions are called **inflated nice tree decompositions**. This generator can be found in the **inflated_ntd.py** file.

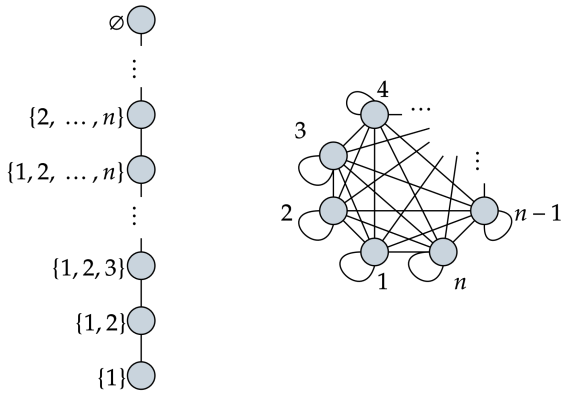

Figure 6.1: Inflated Nice Tree Decomposition on the left yielding the possible edges on the right

The second generator produces **path nice tree decompositons** and can be found in the **path_ntd.py** file. These nice tree decompositions consist of $2k$ nodes for a given $k \in \mathbb{N}$ and have width 1. They are made of alternating introduce and forget nodes while all bags except the root have at least one vertex contained. Therefore $|E_\tau| = 2n - 1$. The pattern simply introduces a new vertex and then forgets the vertex that has been introduced before.

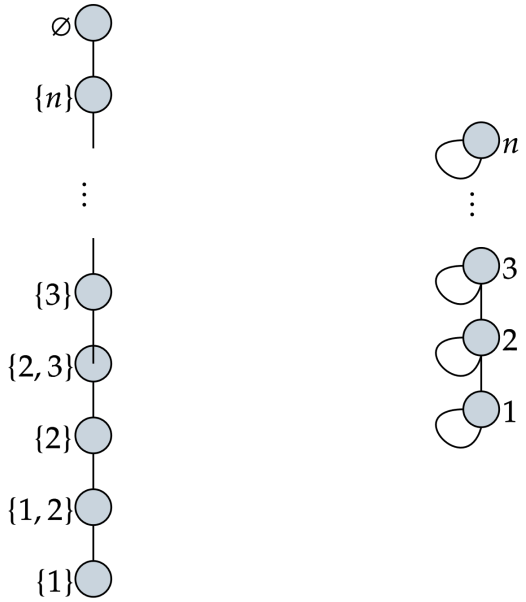

Figure 6.2: A path-like Nice Tree Decomposition on the left yielding the possible edges on the right

The third generator is located in the file **fixed_length_path_ntd.py** and produces instances similar to those of the previous generator. Given two arguments $n, j \in \mathbb{N}$ with $j < n$ the generator constructs a nice tree decomposition of width 2 and $|E_\tau| = n + j$ while fixing the number of nodes to $2k$ and the number of vertices to n. The pattern of this generator works as follows. For $j = 0$ each introduced node and the leaf is followed by a forget node with an empty bag. Therefore E_τ consists only of self-loops. For $j = 1$ the set E_τ will additional have the edge $\{1, 2\}$. Therefore the first(on the leaf to root path) forget node will be changed to an introduce node with the bag containing 1 and 2, which adds the edge to E_τ. And the following introduce node becomes a forget node with the bag containing 2. This generator allows to scale E_τ between n and $2n - 1$ while fixing all other parameters of the nice tree decomposition.

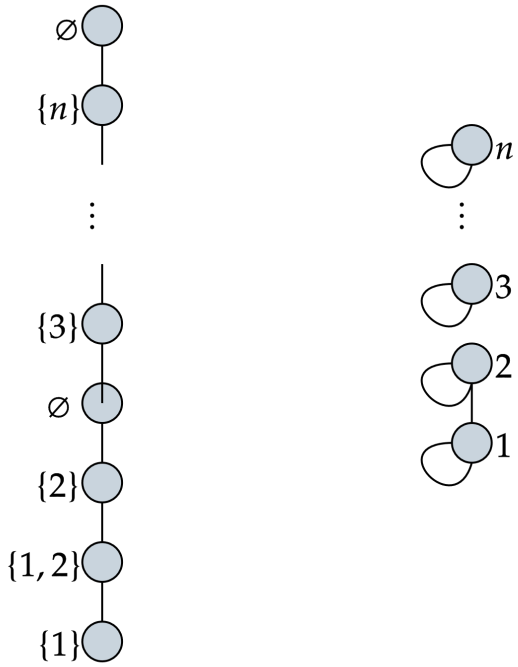

Figure 6.3: A path-like Nice Tree Decomposition of fixed length on the left yielding the possible edges on the right. In this example j equals 1.

6.1.2 Running Time of the Algorithms

This section independently considers experimental results regarding the running time of the algorithm and compares them to the theoretical running times. Let us resume the theoretical running time of the three algorithms. First of all, we have the running time for the brute force algorithm, which iterates through all mappings and check if they are homomorphisms or not. The running time of the brute force algorithm is $O(2^{|E_\tau|} \cdot |V(G)|^{|V_\tau|} \cdot |E_\tau|)$. The running time of the dynamic program of Díaz, Serna and Thilikos has the running time $O(2^{|E_\tau|} \cdot |V_\tau||V(G)|^{k+1} \cdot \min\{k, |V(G)|\})$ where k denotes the width of τ. The modified dynamic program from chapter 4 has a theoretical running time of $O\left(|V(G)| \cdot |V(G)|^{k+1} \cdot \sum_{p \in v(T)} 2^{|E[\alpha(p)]|}\right)$. The running time of the algorithms will be stated in microseconds.

Figures 6.4 and 6.5 show the running time of the brute force algorithm in relationship to the parameters $|E_\tau|$ and $|V(G)|$ in logarithmic representation. In Figure 6.4 a exponential growth of the running time in relation to $|E_\tau|$ can be observed when looking at the tendency of the running time. Also an exponential rise in relation to V_τ is noticeable in figure 6.4. Figure 6.5 on the other hands shows how the brute force algorithm behaves when fixing τ and increasing $V(G)$.

Figures 6.6 shows the running time of the dynamic program of Díaz, Serna and Thilikos in relation to the number of possible edges $|E_\tau|$. The strictly increasing exponential curve of the running time matches the theoretical assumptions. Figure 6.7 shows its running time plotted against $|V(G)|$. The linear behaviour in the logarithmic representation correlates to the polynomial $|V(G)|^{|k+1|}$. Hence the theoretical analysis fits the experimental results.

Nearly the same results can be seen in figures 6.8 and 6.9 for the modified dynamic program. The measured data in figure 6.8 underlie a little bit greater scattering in comparison to 6.6. The graph in figure 6.9 shows a linear growth in its logarithmic representation which correctly relates to the term $|V(G)|^{|k+1|}$.

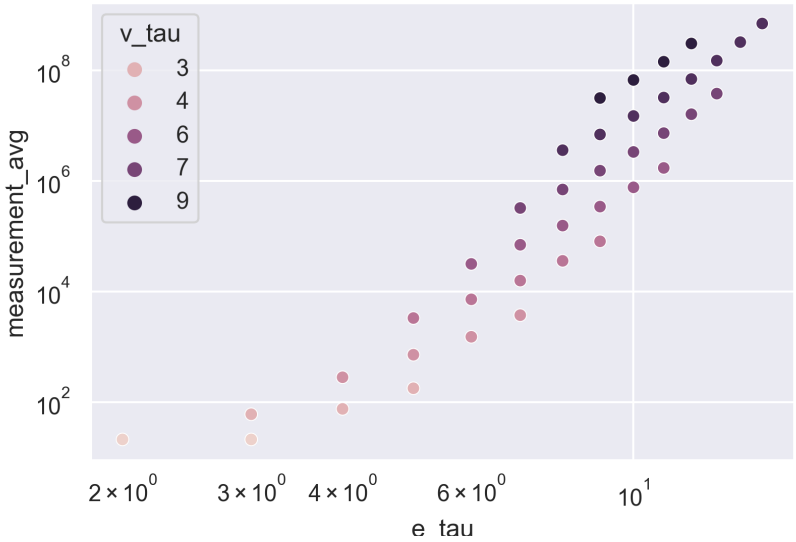

Figure 6.4: running time of the brute force algorithm in relationship to $|E_\tau|$

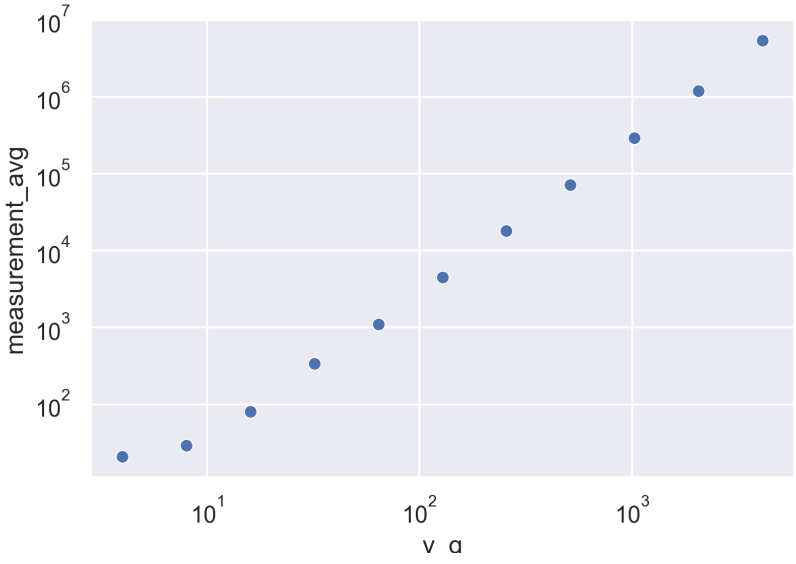

Figure 6.5: running time of the brute force algorithm in relationship to $|V(G)|$

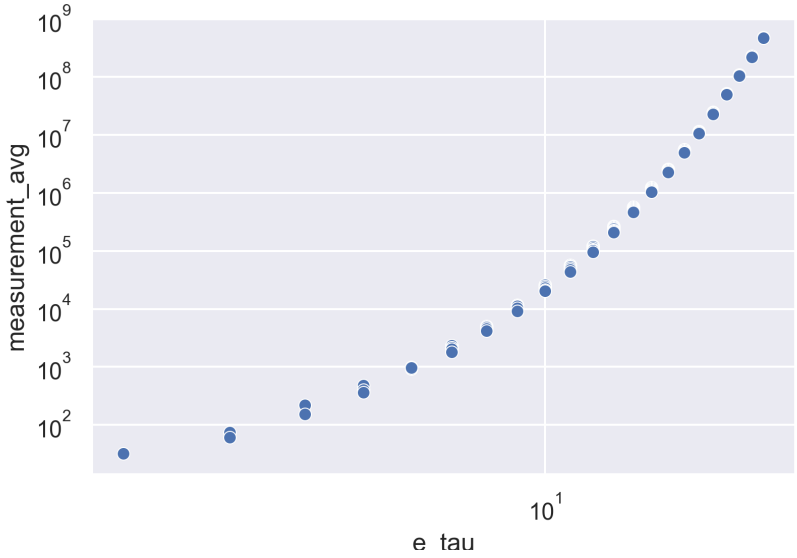

Figure 6.6: running time of the DP of Diaz,Serna & Thilikos in relationship to $|E_\tau|$

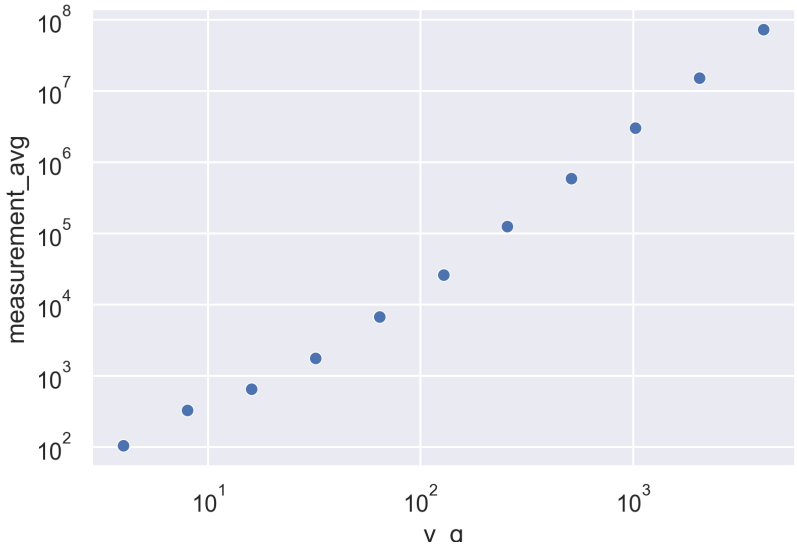

Figure 6.7: running time of the DP of Diaz,Serna & Thilikos in relationship to $|V(G)|$

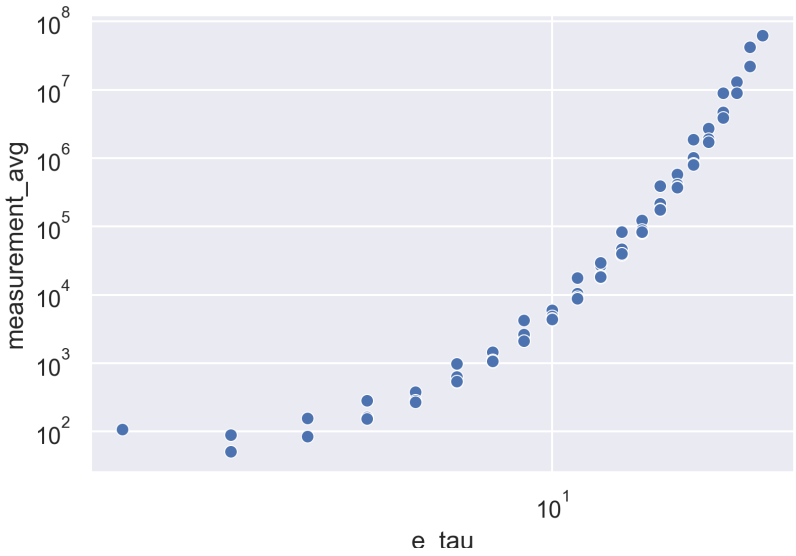

Figure 6.8: running time of the modified DP in relationship to $|E_\tau|$

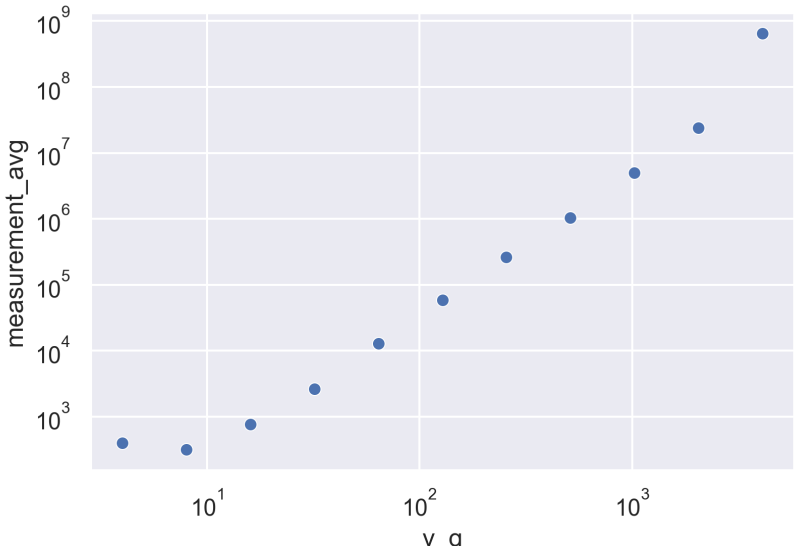

Figure 6.9: running time of the modified DP in relationship to $|V(G)|$

6.1.3 Running Time Comparison

Figure 6.10 shows the running time of the three algorithms plotted against $|E_\tau|$ in a linear representation and figure 6.11 shows the same data in logarithmic representation. It can be seen clearly

that the modified dynamic program is faster than the original dynamic program by Diaz, Serna and Thilikos for the test instances used in the experiment. But this result has to be stated carefully since the last two measurements for the modified dynamic program cant be measured properly. The upper two orange dots are missing their green equivalent. This may be an indication of a much larger running time of the modified dynamic program caused by some unpredictable side effects such as memory overload. The biggest test instance had a set of possible edges with $|E_\tau| = 23$ which would lead to at least $2^{23} \approx 8.3 \times 10^6$ entries stored simultaneously in the table of the root node and this is a lower bound for the space. Hence missing memory may be a real reason for this slowdown.

Figure 6.12 compares the running time of all three algorithm in relation to $V(G)$ where τ was fixed. All three algorithms seem to have the same behaviour when the only changing parameter is $V(G)$. Surprisingly, both dynamic programs are slower than the brute force algorithm for this small fixed τ. The nice tree decomposition used for those experiments has only three possible edges and all edges are contained in one single bag. Therefore it is plausible that the brute force algorithm is faster because both dynamic programs also perform some kind of brute-forcing over all mappings regarding one bag.

Figure 6.13 shows the average running time needed for one graph for each instance plotted against $|E_\tau|$. In the range of the test instances, the modified dynamic program is faster than both other algorithms in the majority of instances.

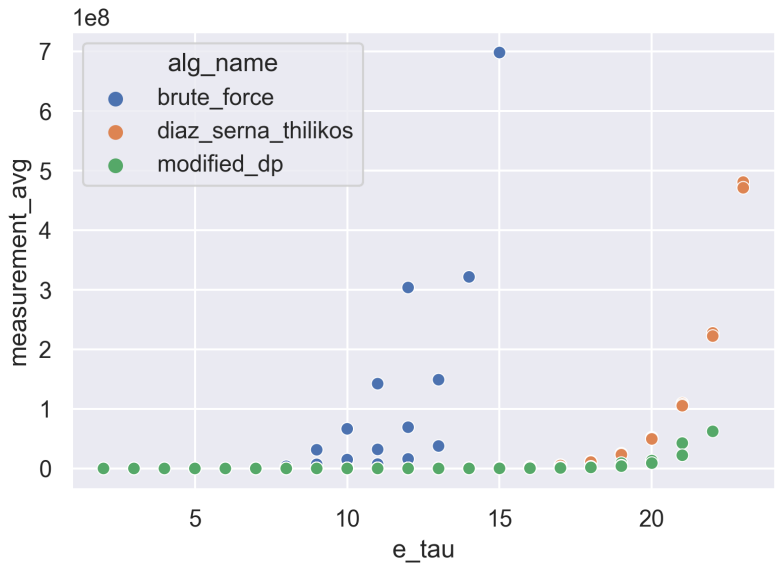

Figure 6.10: running time comparison in relationship to $|E_\tau|$ in linear representation

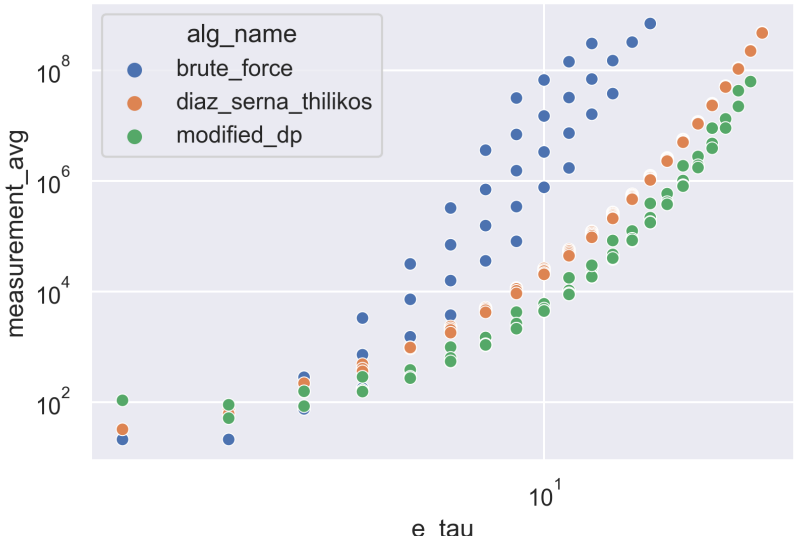

Figure 6.11: running time comparison in relationship to $|E_\tau|$

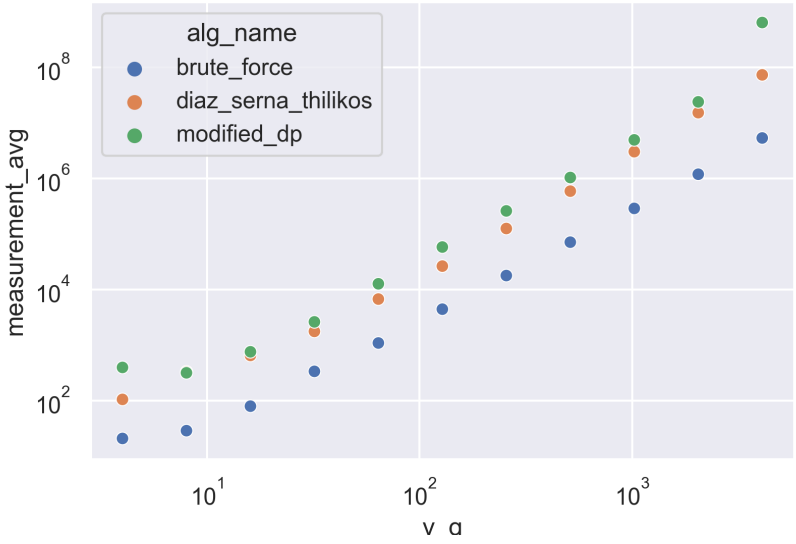

Figure 6.12: running time comparison in relationship to $|V(G)|$

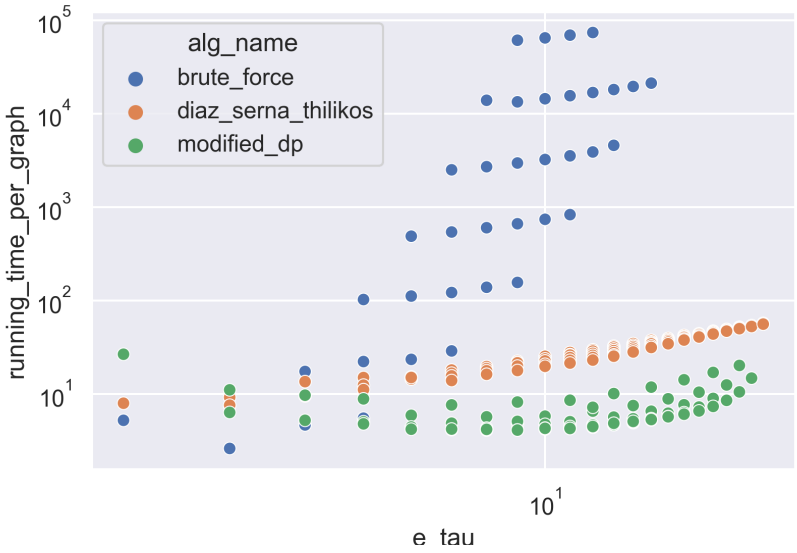

Figure 6.13: average running time per graph in relationship to $|E_\tau|$

6.2 SUMMARY

In summary, the following results can be listed. This work provides a theoretical modification of the dynamic program of Diaz, Serna and Thilikos to solve the extended counting graph homomorphism problem. Additionally, a simple brute force algorithm was presented. All three algorithms together with a file-handler and necessary subroutines have been implemented and evaluated experimentally. The modified dynamic program solves the problem faster than the original one for the majority of test instances. But the modified dynamic program needs exponential more space for the computation and is therefore impracticable for greater instances. To be more precise the modified dynamic program was not able to solve the problem for tree decomposition providing 23 possible edges or more. But within this range, the modified dynamic program yields a faster running time. This algorithm can now be used to create test sets for counting graph homomorphism algorithms.

6.2.1 Further Enhancements

The last section is now dedicated to further ideas and possible modifications. These notes are separated into two parts. The first part consists of possible practical improvements to the implementation and the second part deals with theoretical enhancements. Since all of these ideas are mostly not related, they are simply listed in the following.

Practical Enhancements

- Adjusting the hash functions for the table of the dynamic program may decrease the running time.
- Another question is, whether it is possible to efficiently translate all the computational steps into matrix operations. Those could be efficiently computed by the GPU.
- The implementation of the algorithm works serially. Parallelizing some of the calculations may decrease the running time as well.

Theoretical Enhancements

- The set H_τ is characterized by τ and therefore we cannot represent an arbitrary set of graphs with a single τ. The question that arises here is whether we can find for an arbitrary set of graphs $\mathcal{G}$ a minimum set of tree decompositions $\tau_1, \ldots, \tau_m$ such that $\mathcal{G} \subseteq H_{\tau_1} \cup \cdots \cup H_{\tau_m}$. The term *minimum* in this context can be interpreted in several ways. It could mean to minimize m or to minimize $|H_{\tau_1} \cup \cdots \cup H_{\tau_m} \backslash \mathcal{G}|$. Whereby the latter can be solved trivially if self-loops are neglected by simply creating one tree decomposition per edge.
- The speed-up of the modified dynamic program is based on efficiently finding isomorphic subgraphs during the computation by checking the identity of the labelled graphs. Another way of finding may be identifying isomorphic subtrees in the tree decomposition by canonizing them. This is based on the idea that if two subtrees produce isomorphic graphs when looking at the possible edges and we can find such an isomorphism efficiently then we only have to compute only the entries of one of the subtrees.
- Some nice tree decompositions contain redundant information in some of their nodes. The natural question that arises from this observation is whether we can reduce the size of a tree decomposition while keeping the set H_τ identically. More precisely, given a (nice) tree decomposition τ is there another (nice) tree decomposition τ' such that $H_\tau = H'_\tau$ and $|V(\tau')| < |V(\tau)|$. Furthermore, we can ask if there is an efficient routine to compute τ'.

Bibliography

[1] Umberto Bertelè and Francesco Brioschi. On non-serial dynamic programming. *Journal of Combinatorial Theory, Series A*, 14(2):137–148, 1973. URL: https://www.sciencedirect.com/science/article/pii/0097316573900162, doi:https://doi.org/10.1016/0097-3165(73)90016-2.

[2] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *Journal of Algorithms*, 21(2):358–402, 1996. URL: https://www.sciencedirect.com/science/article/pii/S0196677496900498, doi:https://doi.org/10.1006/jagm.1996.0049.

[3] Christian Borgs, Jennifer Chayes, Laszlo Lovasz, Vera T. Sos, and Katalin Vesztergombi. *Counting Graph Homomorphisms*, pages 315–371. Springer, topics in discrete mathematics (eds. m. klazar, j. kratochvil, m. loebl, j. matousek, r. thomas, p. valtr) edition, February 2006. URL: https://www.microsoft.com/en-us/research/publication/counting-graph-homomorphisms/.

[4] Radu Curticapean, Holger Dell, and Dániel Marx. Homomorphisms are a good basis for counting small subgraphs. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. ACM, jun 2017. URL: https://doi.org/10.1145%2F3055399.3055502, doi:10.1145/3055399.3055502.

[5] Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh. *Parameterized Algorithms -*. Springer, Berlin, Heidelberg, 2015.

[6] Víctor Dalmau and Peter Jonsson. The complexity of counting homomorphisms seen from the other side. *Theoretical Computer Science*, 329(1):315–323, 2004. URL: https://www.sciencedirect.com/science/article/pii/S0304397504005560, doi:https://doi.org/10.1016/j.tcs.2004.08.008.

[7] Martin Dyer and Catherine Greenhill. The complexity of counting graph homomorphisms. *Random Structures & Algorithms*, 17(3-4):260–289, 2000. doi:https://doi.org/10.1002/1098-2418(200010/12)17:3/4<260::AID-RSA5>3.0.CO;2-W.

[8] Josep Díaz, Maria Serna, and Dimitrios M. Thilikos. Counting h-colorings of partial k-trees. *Theoretical Computer Science*, 281(1):291–309, 2002. Selected Papers in honour of Maurice Nivat. URL: https://www.sciencedirect.com/science/article/pii/S0304397502000178, doi:https://doi.org/10.1016/S0304-3975(02)00017-8.

[9] J. Flum and M. Grohe. *Parameterized Complexity Theory -*. Springer Science & Business Media, Berlin Heidelberg, 2006.

[10] Martin Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM*, 54(1), mar 2007. doi:10.1145/1206035.1206036.

[11] Geňa Hahn and Claude Tardif. *Graph homomorphisms: structure and symmetry*, pages 107–166. Springer Netherlands, Dordrecht, 1997. doi:10.1007/978-94-015-8937-6_4.

[12] Pavol Hell and Jaroslav Nešetřil. On the complexity of h-coloring. *Journal of Combinatorial Theory, Series B*, 48(1):92–110, 1990. URL: https://www.sciencedirect.com/science/article/pii/009589569090132J, doi:https://doi.org/10.1016/0095-8956(90)90132-J.

[13] George Karypis. Metis - a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. 2011.

[14] Ton Kloks. *Treewidth - Computations and Approximations*. Springer Science & Business Media, Berlin Heidelberg, 1994.

[15] Emil Ruhwald Nielsen, Otto Stadel Clausen, and Elisabeth Terp Reeve. Counting subgraph patterns in large graphs. 2021.

[16] Neil Robertson and P.D. Seymour. Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1):39–61, 1983. URL: https://www.sciencedirect.com/science/article/pii/0095895683900795, doi:https://doi.org/10.1016/0095-8956(83)90079-5.

[17] Neil Robertson and P.D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984. URL: https://www.sciencedirect.com/science/article/pii/0095895684900133, doi:https://doi.org/10.1016/0095-8956(84)90013-3.

[18] Neil Robertson and P.D Seymour. Graph minors. ii. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986. URL: https://www.sciencedirect.com/science/article/pii/0196677486900234, doi:https://doi.org/10.1016/0196-6774(86)90023-4.

[19] Marc Roth and Philip Wellnitz. Counting and finding homomorphisms is universal for parameterized complexity theory, 2019. URL: https://arxiv.org/abs/1907.03850, doi:10.48550/ARXIV.1907.03850.