TECHNISCHE
UNIVERSITÄT
WIEN

DIPLOMARBEIT

Turning FPT Decision Methods into Enumeration Algorithms with FPT-Delay

ausgeführt am

Institut für
Logic and Computation
TU Wien

unter der Anleitung von

Univ.Prof. Dr.techn. Reinhard Pichler

und der Mitwirkung von

Dipl.Ing. Timo Camillo Merkl, BSc

durch

Daniel Unterberger, BSc
Matrikelnummer: 01604891

Wien, am 18. März 2024

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

Parameterized algorithms are a well-studied subject in algorithmics and complexity theory used to identify tractable fragments of hard problems. Yet they have been primarily been examined in the context of decision and optimization problems. However, in many practical contexts the output of multiple or even all solutions fitting certain restrictions is required.

Research on enumeration problems has come up with several methods that can typically be used to design efficient algorithms for finding all solutions to a given problem. Such algorithms have been mainly investigated with the goal of achieving polynomial time or polynomial delay complexity.

Despite the successes in both fields, their combination has received almost no attention so far. Therefore, the goal of this thesis is to advance the study of parameterized enumeration algorithms. More concretely, we aim at developing enumeration algorithms with fixed-parameter tractable delay for a variety of hard enumeration problems.

To this end we first present a theoretical grounding of parameterized complexity, enumeration complexity and their combination. Then we explore how and where methods for finding enumeration algorithms can be applied to parameterized problems, as well as transforming methods used for parameterized algorithms to methods for enumeration. We use these to solve the associated enumeration problems for several parameterized decision problems.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Kurzfassung

Parametrisierte Algorithmen sind ein gut erforschtes Gebiet der Algorithmik und Komplexitätstheorie, das verwendet wird um Instanzen schwieriger Probleme zu identifizieren, die effizient lösbar sind. Allerdings wurden diese hauptsächlich im Kontext von Entscheidungs- und Optimierungsproblemen untersucht. In der Praxis wird jedoch oft die Ausgabe von mehreren oder sogar allen Lösungen, die gewisse Eigenschaften erfüllen, benötigt.

Die Analyse von Aufzählungsproblemen hat zu einigen Methoden geführt, die oft verwendet werden können um alle Lösungen eines gegebenen Problems zu finden. Solche Algorithmen wurden vorwiegend entwickelt um polynomielle Zeit oder polynomielle Verzögerung zu erreichen.

Trotz der Erfolge in beiden dieser Bereiche hat deren Kombination bisher kaum Aufmerksamkeit erlangt. Daher ist das Ziel dieser Arbeit Fortschritte im Gebiet der parametrisierten Aufzählungsalgorithmen zu machen. Spezifisch ist es das Ziel Aufzählungsalgorithmen mit FPT ('fixed-parameter-tractable') Verzögerung für eine Vielfalt an schwierigen Aufzählungsproblemen zu entwickeln.

Zu diesem Zweck präsentieren wir zuerst den theoretischen Hintergrund von parametrisierter Komplexitätstheorie, Aufzählungsproblemen und deren Kombination. Dann untersuchen wir sowohl wie Methoden für Aufzählungsalgorithmen auf parametrisierte Probleme angewendet werden können als auch die Umwandlung von Verfahren für parametrisierte Algorithmen auf Aufzählungsalgorithmen. Damit lösen wir das entsprechende Aufzählungsproblem von einigen parametrisierten Entscheidungsproblemen.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acknowledgements

I want to thank Professor Reinhard Pichler, Timo Camillo Merkl and Professor Nadia Creignou for introducing me to this topic, sparking my interest in this entire field of study and helping me greatly throughout the writing process of this thesis.

I want thank my family and friends, who have supported me through my entire studies, without whom I would not have come so far.

Finally, I want to give special thanks to my brother, who always believed in me and whom I could always rely on.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am 18. März 2024

Daniel Unterberger

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Contents

1 **Introduction** 1
1.1 Goals and Results 2
1.2 Structure 3

2 **Preliminaries** 5

3 **Parameterized Algorithms** 7
3.1 Formal Definitions 8
3.2 Approaches to finding solutions 13
3.2.1 Kernelization 13
3.2.2 Branch and Bound 15
3.2.3 Iterative Compression 16

4 **Enumeration Algorithms** 20
4.1 Formal Definitions 20
4.2 Approaches to finding solutions 23
4.2.1 Backtracking 25
4.2.2 Binary Partition 26
4.2.3 Reverse Search 27

5 **Parameterized Enumeration** 30
5.1 Formal Definitions 30

6 **Enum-Kernelization** 35
6.1 Examples 35
6.1.1 Vertex Cover 36
6.1.2 Feedback Arc Set in Tournaments 37
6.1.3 d-Hitting Set 38

7 **Binary Partition** 40
7.1 Branch and Bound 40
7.1.1 Vertex Cover 41
7.1.2 Feedback Vertex Set in Tournaments 42
7.2 Flashlight Algorithms 43
7.2.1 Vertex Cover 44
7.2.2 Closest String 45

8 **Instance Search** 48

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

8.1 Examples . . . 49
8.1.1 Vertex Cover . . . 49
8.1.2 Feedback Vertex Set in Tournaments . . . 50
8.1.3 Integer Linear Programming . . . 51

9 Miscellaneous **53**
9.1 Colour Coding . . . 53
9.1.1 Longest Path . . . 53
9.1.2 Subgraph Isomorphism . . . 55
9.2 Iterative Compression . . . 55
9.2.1 Vertex Cover . . . 56
9.3 Dynamic Programming . . . 57
9.3.1 Steiner Tree . . . 57

10 Conclusion **60**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 1

Introduction

Many mathematical problems that appear in practical or theoretical contexts, like graph theory or operations research, are generally hard to solve (i.e. NP-hard) and thus there is, under usual assumptions of complexity theory, no efficient (i.e. polynomial time) algorithm to solve the problem. One approach of tackling such problems is to use inexact methods: Using heuristics can confirm the existence of a solution in polynomial time for decision problems, but may fail to find a solution even if one exists. Likewise approximation algorithms for optimization problems can find suboptimal solutions that are within guaranteed bounds of the optimum and can often be found efficiently.

But even in cases where only definitive or optimal solutions are of interest, for many actual instances of such problems we can still achieve some sense of tractability due to structural properties of the relevant problem instances or because only solutions of a certain type are of interest. Examples of such properties include the maximal degree of any node or the size of the solution for graph problems. Under some of these properties even NP-hard problems like Vertex Cover or Feedback Vertex Set can be solved quickly.

To formalize such notions, we consider problem instances to consist of a description I and additionally a parameter $k \in \mathbb{N}$ that describes some attribute of the instance I. We can then analyze the complexity of such problems through the lense of multivariate complexity, i.e. we want to bound the runtime of algorithms on instances of size n by $O(f(k, n))$ for certain functions f.

Of particular interest are so-called fixed-parameter tractable ('FPT') problems, which admit algorithms that, for a problem instance of size n and a parameter k, can solve the problem with a runtime of $f(k)n^{O(1)}$. This means that the combinatorial explosion (and the exponential running time) is confined to the parameter, so even large problem instances can be solved efficiently if the parameter is small (e.g. bounded by a constant). Parameterized algorithms and parameterized complexity have evolved into a well studied field with many applications in different fields of mathematics - especially graph theory and logic [9][27][17][18][16][15].

FPT algorithms have primarily been applied to decision and optimization problems. However, for many practical problems it is not possible to fully formalize notions of optimality for solutions, and thus we need to find all solutions to evaluate them individually. Moreover, there are also application (such as evaluations of database queries) where the user is naturally interested in finding all solutions. In these cases it is both important that

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

every solution is found and that no solution is output several times.

This presents a challenge to classical notions of complexity: even simple problems, which can obviously be solved within polynomial or even linear time, can have an exponential number of (optimal) solutions and thus every algorithm enumerating them has a long running time. On the other hand, some difficult problems can have a small number of solutions but due to the inherent difficulty of finding any solution have a comparable running time. While there is an obvious difference in the challenges and the approaches needed for solving such problems, we cannot meaningfully separate such problems on the basis of total running time as a function of the input size.

In the theory of enumeration algorithms a broad spectrum of methods have been developed to solve such problems and to classify their computational complexity [4][30][6][21]. In particular, the definition of tractability has to reconsidered. A natural consideration is to require polynomial time in the combined size of the input and the output. However, this gives no guarantee on the time needed to find the first couple solutions. The primary area of interest in this case are algorithms that guarantee polynomial-delay, i.e. the time spent between the output of any two solutions (and before the first and after the last solution) is bounded by a polynomial of the size of the problem instance.

Obviously the enumeration problem (i.e. listing all solutions) is at least as hard as the respective decision problem (i.e. does there exist a solution). Thus, in contrast to decision algorithms, there has not been a great deal of interest in enumeration algorithms for hard problems. One way of defining a relaxed form of tractability for such problems is through the viewpoint of parameterized complexity: By bounding the delay between any two solutions using the parameter k, we can achieve a similar sense of tractability to classical enumeration algorithms. In particular, we are interested in algorithms that can guarantee a delay bounded by $f(k)n^{O(1)}$.

There have been a few attempts of combining these two promising fields of parameterized complexity and enumeration algorithms. For instance, [25] described different problem classes for parameterized complexity, [10] considered enumeration algorithms for parameterized algorithms with FPT runtime, and [8] extended the framework of kernelization to enumeration algorithms. However, the systematic exploration of FPT-delay algorithms for enumeration problems is a vastly under-explored topic.

1.1 Goals and Results

The goal of this thesis is the design of FPT-delay algorithms for hard enumeration problems. To accomplish this, our primary interest is exploring how FPT decision methods for hard problems can be translated into methods for their respective enumeration problems. To this end, we examine FPT decision problems in [9] and use different approaches used for enumeration algorithms to find FPT-delay algorithms to solve them. We often utilize either the structure of the existing FPT algorithm or use the algorithm directly (or with slight variation to better fit the structure of the enumeration problem) at different steps to find solutions.

While the categories we examine are not always cleanly separable, we aim to present approaches that often allow FPT decision algorithms of certain types to be transformed into enumeration algorithms or utilize the existence of decision algorithms directly to find

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

all solutions.

For many FPT-algorithms the structure of the computation implicitly explores the space of all (subset-minimal or -maximal) solutions, and thus it can easily be adapted to generate an FPT-delay algorithm. Sometimes, to prevent the repeated output of solutions, we alter the structure of the instance slightly (e.g. using colourings on a graph to mark vertices that should not be chosen). Particular care has to be taken so the alteration of instances does not impair the applicability of the algorithmic ideas of the original FPT algorithm.

However, in several cases the FPT algorithm requires the problem instance to be of a certain form, and as a first step transforms a given instance I into an instance I' satisfying these requirements. Typically solvability and optimality are unaffected, but not all solutions are necessarily preserved. Often there is not an immediate way of reconstructing all solutions of I from solutions of I'. Thus, while the structure of the algorithm would allow us to find all solutions of I', we cannot always solve the original problem using this approach.

For such problems or when the structure of the FPT-algorithm did not lend itself to enumeration, we were sometimes able to apply solution approaches for enumeration problems or original methods to find FPT-delay enumeration algorithms. In such cases we used the existing FPT decision algorithm directly (or with slight alterations) to find solutions or to ensure we do not explore cases that do not contain solutions.

In summary, by starting from well-known FPT-algorithms for decision problems and modifying and adapting them to the specifics of enumeration problems, we have managed to create FPT-delay enumeration algorithms for a variety of problems.

1.2 Structure

We first introduce the general theory of parameterized algorithms in Chapter 3, including some background on the different complexity classes for parameterized problems and both positive and negative results for tractability and presenting some approaches that have been used to find FPT algorithms.

Chapter 4 is focused on introducing enumeration problems and their respective complexity classes and theoretical results, and then exploring some methods that have been used to solve them with some examples.

We then give a brief overview of the formal foundations of parameterized enumeration algorithms in Chapter 5, primarily focused on FPT-delay algorithms.

Chapter 6 explores the method of enum-kernelization and presents FPT-delay enumeration algorithms for different problems based upon their respective kernelization algorithms.

Then we focus on two different methods (with some examples, like Closest String and Feedback Vertex Set in Tournaments) that can be broadly categorized as binary partition in Chapter 7. We explore both the methods of branch and bound and flashlight search. The method of flashlight search is then also used in Chapter 8 to generate a first solution and then construct new instances to find the remaining solutions in a method we have

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

called instance search.

In Chapter 9 we focus in some approaches that are not as widely applicable, but still allow the generation of some FPT-delay enumeration algorithms.

Lastly, we conclude the thesis with a short overview of the main results and difficulties, including some unsolved problems. We then point to possible avenues for future research.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Chapter 2

Preliminaries

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

In this thesis, we will be primarily focused on complexity theory and graph problems, and thus we will fix some notations and definitions that are common in graph theory.

We also assume that the reader is familiar with some basic definitions and terminology from complexity theory, including complexity classes like P and NP, and concepts such as polynomial time reductions. As is typical in this context, we will denote the set of all finite strings using an alphabet Σ, including the empty string ϵ, as Σ^*.

For a nonempty set V and a set E of subsets $e \subseteq V$ of cardinality two, we call a tuple $G = (V, E)$ an (undirected) graph, V the set of vertices and E the set of edges. If E instead consists of ordered pairs of elements in V, we call $G = (V, E)$ a directed graph. Unless stated otherwise any graph we consider will be an undirected graph. We will denote both the directed egde (u, v) and the undirected edge $\{u, v\}$ as uv. For a graph G, we sometimes denote the set of vertices of G as $V(G)$ and the set of edges of G as $E(G)$. We sometimes allow E to be a multiset, i.e. to contain multiple copies of the same edge, and to include loops (edges from v to itself). We call such graphs multigraphs.

For a vertex $v \in V$, we will call vertices $u \in V$ with $uv \in E$ neighbours of v. We denote the set of all neighbours of v as $N(v) = \{u \in V : uv \in E\}$ (the graph will be clear from the context), we call $N(v)$ the (open) neighbourhood and $N[v] = N(v) \cup \{v\}$ the closed neighbourhood of v. We call $d(v) = |N(v)|$ the degree of the vertex v. For directed graphs, we instead use $N_+(v) = \{u \in V : uv \in E\}$ and $N_-(v) = \{u \in V : vu \in E\}$. We call $d_+(v) = |N_+(v)|$ the indegree and $d_-(v) = |N_-(v)|$ the outdegree of the vertex v.

For a graph $G = (V, E)$ and a set $W \subseteq V$, we define $G[W] = (W, E_W)$ with $E_W = \{uv \in E : u, v \in W\}$. We call $G[W]$ the subgraph of G induced by W.

Similarly, for a set $W \subseteq V$ of vertices we denote $G[V \backslash W]$ as $G - W$, and for a set $F \subseteq E$ of edges we denote $G - F = (V, E \backslash F)$. If W or F consists of a single element x, we will also write $G - x$ instead of $G - \{x\}$.

We call a sequence $S = (v_1, \cdots, v_l)$, where for all $1 \leq i \leq l - 1$ $v_i v_{i+1} \in E$ a walk. We will also consider such a walk S to consist of the respective edges $v_i v_{i+1}$. If all edges in S are distinct, we call S a trail, and if all vertices in S are distinct we call S a path. We call S a walk (or trail/path) from v_1 to v_l, and we will call a trail S from v to itself a cycle. If S is contained in a directed graph, we will also refer to it as a directed cycle.

For any two vertices $u, v \in V$, we say v is reachable from u if there exists a path from u to v. For undirected graphs, we define the (connected) component of v as $C(v) = \{v\} \cup \{u : u \text{ is reachable from } v\}$. It is clear that $v \in C(u) \Leftrightarrow u \in C(v)$.

Lastly, we say a graph $G = (V, E)$ containing no cycles is a forest, and we say G is connected if $C(v) = V$ for all $v \in V$. If a graph is both a forest and connected we call it a tree.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 3

Parameterized Algorithms

Many computational problems that arise in theoretical and practical settings turn out to NP-hard, and thus there is no efficient (i.e. polynomial time) algorithm for solving a problem unless P = NP. Often a heuristical approach can be applied to either sometimes find a solution, if it exists (but also sometimes miss it) in decision problems or to find a solution that, while suboptimal, is within guaranteed bounds of the optimal solution for optimization problems. For the purposes of this chapter we will primarily consider decision problems as optimization problems can often be phrased through the lens of decision problems (e.g. instead of asking what is the maximal score among all solutions, we ask whether there is a solution with a score higher than some $s \in \mathbb{N}$). We base this chapter mainly on the definitions, algorithms and results presented in [9] and [27].

However, it is possible to solve even NP-hard problems quickly if certain guarantees about the structure of the input or restrictions on the solution are imposed. Such structures can often be given by the surrounding context of the setting to which the problem will be applied in practice.

We will motivate this approach by considering a parameterized version of the Vertex Cover problem, which is known to be NP-complete.

Let $G = (V, E)$ be a graph. A vertex cover $C \subset V$ is a set such that for every edge $e \in E$ there is at least one $v \in C$ adjacent to e. The Vertex Cover problem parameterized by k is whether there exists a vertex cover of size k.

By the definition of vertex covers, for $uv \in E$ any solution S for (G, k) either contains u or v and thus either $S \backslash \{u\}$ is a solution for $(G - u, k - 1)$ or $S \backslash \{v\}$ is a solution for $(G - v, k - 1)$. Thus we can recursively branch over these possibilities up to a depth of k as long as there are edges remaining, and either we find a solution along some branch and thus have a yes-instance, or we do not and have a no-instance.

We will set $Sol := \emptyset$ and then repeatedly apply the following steps:

- If $E = \emptyset$, then (G, k, Sol) is a yes-instance,
- for any $uv \in E$, recursively apply this algorithm to $(G - u, k - 1, Sol \cup \{u\})$ and $(G - v, k - 1, Sol \cup \{v\})$, and (G, k, Sol) is a yes-instance iff either branch is a yes-instance,
- if $k = 0$ and $E \neq \emptyset$, (G, k, Sol) is a no-instance.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The runtime of this algorithm is bounded by the number of instances we need to consider, which in this case is bounded by $O(2^k)$, and together with some time for handling the graph structure at each step we get an overall bound of $O(2^k \cdot n^c)$ for some constant c, thus if we are only interested in small vertex covers (e.g. of constant size) we can solve the problem quickly.

As an example of more structural restrictions for problem instances, we consider the Dominating Set problem: Let $G = (V, E)$ be a graph. We call a set $S \subseteq V$ a dominating set if, for each $v \in V$, either $v \in S$ or there exists an $u \in V$ such that $uv \in E$. If both the maximal degree and the size of S are bounded by k, we can construct an algorithm for (G, k) as follows:

For every vertex v, S must contain either v or some neighbour u of v. Thus we first set $Sol = \emptyset$, then apply the following steps:

- If $V = \emptyset$ then (G, k, Sol) is a yes-instance,
- for any $v \in V$, we recursively apply these steps to $(G - u, k - 1, Sol \cup \{u\})$ for every $u \in N[v]$. Then (G, k, Sol) is a yes-instance if and only if any of these new instances is a yes-instance,
- if $k = 0$ and $V \neq \emptyset$, (G, k, Sol) is a no-instance.

As the maximal degree for G is bounded by k, $N[v]$ for any vertex v has at most $k + 1$ elements, and therefore we can bound the runtime of this algorithm by $O((k + 1)^k n^c)$ as with the Vertex Cover problem.

Parameterized Algorithms are a way of formalizing this insight and to classify different problems and algorithms such that, under certain restrictions, we can make even hard problems tractable.

3.1 Formal Definitions

Before we consider different approaches of finding algorithms to solving parameterized problems, we will first define them formally and consider different complexity classes and reductions that preserve their membership.

Definition 1 ([9])**.** *A parameterized problem is a language $L \subseteq \Sigma^* \times \mathbb{N}$ for a fixed, finite alphabet Σ. For an instance $(I, k) \in \Sigma^* \times \mathbb{N}$, k is called the parameter.*

We note that some authors define parameterized problems as $L \subseteq \Sigma^* \times \Sigma^*$ to allow certain structures to be used for inputs (instead of just their sizes). As for all problems we examine we only consider parameters expressed as natural numbers we will restrict ourselves to that definition.

Of course the choice of parameter for L can vastly impact the hardness of a given problem. As such, we can not separate a given problem from the parameter we use.

We broadly classify parameterized problems into 3 classes, depending on how efficiently they can be solved.

Definition 2. *A parameterized problem $L \subset \Sigma^* \times \mathbb{N}$ is said to be fixed-parameter tractable (FPT) if there is an algorithm that for $(I, k) \in \Sigma^* \times \mathbb{N}$ correctly decides if $(I, k) \in L$*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

with a runtime of $f(k)|(I,k)|^c$ for a computable function f and a constant c. We will refer to such algorithms as FPT algorithms and the class of all problems admitting FPT algorithms as simply FPT.

For our purposes FPT algorithms will, as polynomial time algorithms are in many unparameterized contexts, be our main focus.

Definition 3. *A parameterized (decision) problem $L \subset \Sigma^* \times \mathbb{N}$ is said to be slice-wise polynomial (XP) if there is an algorithm that for $(I,k) \in \Sigma^* \times \mathbb{N}$ correctly decides if $(I,k) \in L$ with a runtime of $f(k)|(I,k)|^{g(k)}$ for computable functions f and g. We will refer to such algorithms as XP algorithms and the class of all problems admitting XP algorithms as simply XP.*

For both FPT and XP we will henceforth only consider non-decreasing functions f, g, as we could otherwise achieve an upper bound by replacing $f(k)$ with $f^*(k) := \max_{i\in\{1,\cdots,k\}} f(i)$ and $g^*(k) := \max_{i\in\{1,\cdots,k\}} g(i)$ (this will be useful for estimating upper bounds later in this section).

From these definitions it is clear that any problem that is FPT is also XP. While both problems have a polynomial running time for a fixed k, FPT algorithms are in general significantly better the combinatorial explosion is contained solely to k.

For many parameterized graph problems it is easy to arrive at an XP algorithm by brute force: In the Clique Problem, parameterized by the required solution size k, we want to find a clique (i.e. a complete subgraph) in a graph $G = (V,E)$ of size at least k. By simply going through each subset $C \subset V$ of size k and checking whether it constitutes a clique we have an XP algorithm. As we will see later it is unlikely that there exists an FPT algorithm for Clique.

Definition 4. *A parameterized problem L is said to be para-NP hard if, for a fixed k, L is NP-hard.*

An example of a para-NP hard problem is k-Colourability, parameterized by k. In k-Colourability we want to know, for a given graph G, whether it is possible to colour the vertices with k colours such that no two neighbouring vertices have the same colour. As 3-Colourability is already NP-complete there cannot be an XP algorithm for k-Colourability unless P=NP.

Definition 5. *Let $L_1, L_2 \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems. We say a function $f : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ is a parameterized reduction from L_1 to L_2 if there exist computable functions g_1, g_2 such that*

1. *$f(I,k)$ can be computed in $g_1(k) \cdot |I|^c$ time,*
2. *for $f(I,k) = (I', k')$ it holds that $k' \leq g_2(k)$, and*
3. *$(I,k) \in L_1$ if and only if $(I',k') \in L_2$.*

Notably not all reductions between NP problems can be translated to parameterized reductions. To show this, we consider a classical problem reduction from the vertex cover problem to the independent set problem: a set $S \subseteq V$ is an independent set of the graph $G = (V,E)$ if $G[S]$ has no edges. The independent set problem is, for a given graph G and integer k, to decide whether there exists an independent set of size at least k. It is easy to show that $S \subseteq V$ is an independent set if and only if $V \backslash S$ is a vertex cover.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Thus an independent set of size $\geq k$ exists in G if and only if there exists a vertex cover of size $\leq |V| - k$, so this constitutes the unparameterized problem reduction from independent set to vertex cover. However, this reduction is not a parameterized reduction as $|V| - k$ can not generally be bounded by a function of k, thus although vertex cover can be solved quickly based upon the parameter, this reduction does not guarantee a fast solution algorithm for independent set. As we will see later, it is generally assumed that there exists no efficient algorithm for solving the independent set problem (parameterized by the solution size).
In particular, under usual assumptions about parameterized complexity (akin to P≠NP), while certain NP-complete problems are FPT, others cannot be made tractable in this way.

Theorem 1 ([9])**.** *Let $L_1, L_2 \subseteq \Sigma^* \times \mathbb{N}$ be parameterized problems, let L_2 be FPT and let $f : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ be a parameterized reduction from L_1 to L_2. Then L_1 is also FPT.*

Proof. We can construct an FPT algorithm for L_1 as follows: Let (I, k) be an instance of L_1. We then

- compute $f(I, k) = (I', k')$ and
- apply A to (I', k') and output the result.

Applying f constructs $(I'.k')$ within a time of $g(k)|I|^{c_1}$ for a computable function g and constant c_1, and thus $|I'| \leq g(k)|I|^{c_1}$. Applying A to (I', k') outputs (correctly) whether $(I', k') \in L_2$ with a runtime of

$$h(k')|I'|^{c_2} \leq h(g(k))(f(k)|I|^{c_1})^{c_2}$$

for some h, c_2. Added together we have a runtime bounded by $f'(k)|I|^{c_1 c_2}$ for $f'(k) = f(k) + h(g(k))f(k)^{c_2}$.

As $(I, k) \in L_1$ if and only if $(I', k') \in L_2$ this algorithm correctly decides if $(I, k) \in L_1$ in FPT time. □

We also note that parameterized reductions are transitive in the following sense:

Theorem 2 ([9])**.** *Let $L_1, L_2, L_3 \subseteq \Sigma^* \times \mathbb{N}$ be parameterized problems. If there is a parameterized reduction from L_1 to L_2 and from L_2 to L_3, then there exists a parameterized reduction from L_1 to L_3.*

Proof. Let (I, k) be an instance of L_1. Using the parameterized reduction we get an instance (I', k') of L_2 in time $f_1(k)|I|^{c_1}$ with $k' \leq g_1(k)$ and $|I'| \leq f_1(k)|I|^{c_1}$. Applying the second reduction we construct an instance (I'', k'') of L_3 in time $f_2(k')|I'|^{c_2} \leq f_2(g_1(k))(f_1(k)|I|^{c_1})^{c_2}$ such that

$$\begin{aligned} k'' &\leq g_2(k) \leq g_2(g_1(k)) =: g(k), \\ |I''| &\leq f_2(g_1(k))(f_1(k)|I|^{c_1})^{c_2} = \underbrace{f_2(g_1(k))f_1(k)^{c_2}}_{=:f(k)} |I|^{c_1 c_2}. \\ (I, k) \in L_1 &\Leftrightarrow (I', k') \in L_2 \Leftrightarrow (I'', k'') \in L_3 \end{aligned}$$

Thus the concatenation of parameterized reduction is again a parameterized reduction. □

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

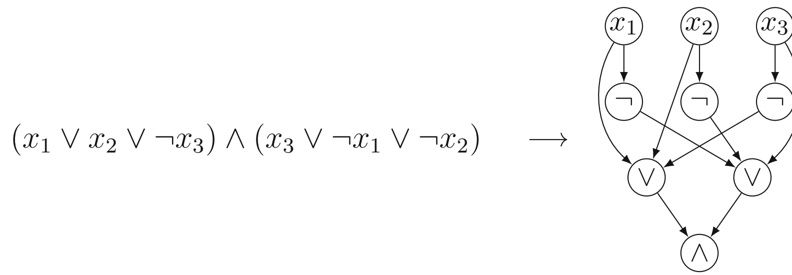

Figure 3.1: A 3SAT instance and the corresponding Boolean Circuit. Either is satisfiable if and only if the other is satisfiable.

We will now briefly touch on parameterized problems and their respective complexity classes which are often assumed not to be FPT. To do so, we will first introduce boolean circuits:

Definition 6. *A Boolean Circuit is a directed, acyclic graph such that nodes are labelled as*

- *input nodes if they have an indegree of 0,*
- *negation nodes if they have an indegree of 1, and*
- *and-nodes or or-nodes if they have an indegree of* ≥ 2.

Additionally there is exactly on node with outdegree 0, which we call the output node. The depth of the circuit is the length of the longest path from any input node to the output node.

We say a Boolean Circuit is satisfiable if there is a way of assigning values in $\{0, 1\}$ to the input nodes such that, following the rules of boolean logic (e.g. an and-node gets the value 1 if all nodes that have egdes to it have value 1), the output node has a value of 1. As the Boolean Circuit Satisfiability Problem is clearly in NP, and we can easily translate a 3SAT instance into a Boolean Circuit, we know that it is NP-complete.

The weight of an assignment to the input nodes is the number of nodes with value 1. In Weighted Circuit Satisfiability (WCS) we are given a Boolean Circuit C and an integer k, and the task is to decide if there is an assignment of weight exactly k satisfying C. We define WCS[$\mathcal{C}$] as the problem restricted to circuits $C \in \mathcal{C}$.

For a given Circuit C we say that nodes with indegree ≥ 3 are large, and otherwise small. The weft of a circuit is the maximal number of large nodes on a path from an input node to the output node. We call the class of all circuits with a depth $\leq d$ and a weft $\leq t$ $\mathcal{C}_{t,d}$.

Definition 7. *For* $t \geq 1$ *W[t] is the class of parameterized problems* P *such that there exists a parameterized reduction from* P *to WCS[*$\mathcal{C}_{t,d}$*] for some* $d \geq 1$.

Theorem 3 ([9])**.** *Independent Set is in* $W[1]$.

Proof. Let $G = (V, E)$ be a graph and let (G, k) be an instance of Independent Set. We construct a circuit C of depth 3 and weft 1 representing (G, k) as follows:

- We create one input node per vertex v in V. We will refer to these nodes as n_v.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

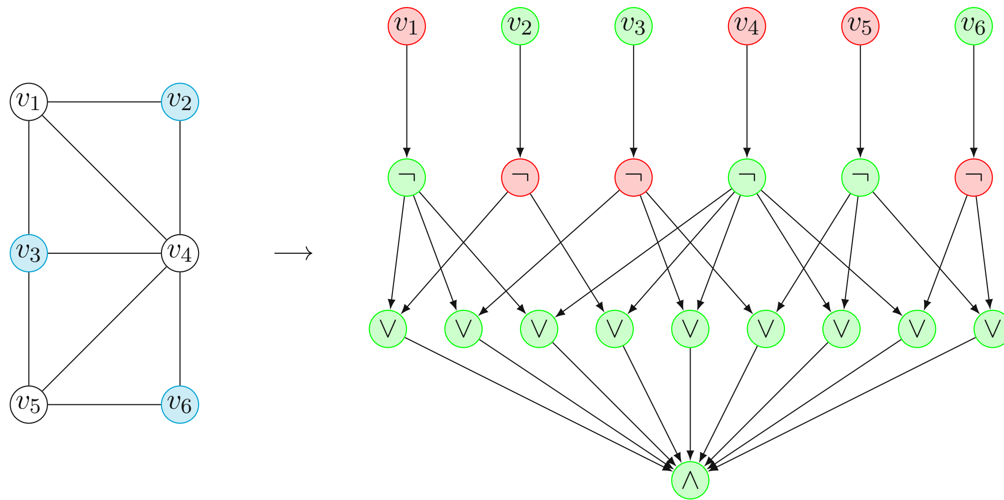

Figure 3.2: A Graph G an the Boolean Circuit of depth 3 and weft 1 representing the independent set problem on G. An independent set of size 3 is highlighted on the left, an assignment of weight 3 on the right, nodes evaluating to 1 in green and to 0 in red.

- We connect each input node to exactly one negation node. We will refer to the node connected to n_v as n'_v.
- For each edge $uv \in E$ we create an or-node connecting n'_v and n'_u.
- We create an output and-node connected to all or-nodes from the previous step.

Let S be an independent set of size k. Then we claim that assigning exactly the nodes in S to 1 satisfies C. By construction, n'_v is 1 if and only if $v \notin S$. As each or-node represents an edge uv, and an independent set cannot contain both u and v, either n'_u or n'_v is 1, an thus each or-node is 1 and therefore the output node is 1. So this is an assignment of weight k satisfying C.

Now let S be the input nodes set to 1 in an assignment satisfying C (of weight k). Then each or-node representing an edge uv must be set to 1 through this assignment, and thus either n'_u or n'_v must be 1. That means for each edge uv S does not contain both u and v, and represents therefore an independent set of size k. □

As there is a simple reduction from Clique to Independent Set in both ways (by switching edges and non-edges), we also get that Clique is in W[1].

We say that a problem Q is W[t]-complete if there exists a parameterized reduction from Q to every other problem in W[t].

Theorem 4 ([11])**.** *Independent Set is W*[1]*-complete.*

It is generally assumed that W[1] $\neq$ FPT, and thus certain problems, like Clique or Independent Set, cannot be solved efficiently.

We want to briefly highlight a result for W[2]-completeness:

Theorem 5 ([9])**.** *Dominating Set (parameterized by the maximal size of a solution) is W*[2]*-complete.*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We highlight this result in contrast to the example at the introduction of the chapter, where we have shown a simple FPT algorithm for Dominating Set if both the size of a solution AND the maximal degree of any vertex are restricted. The tractability of any given problem is highly dependent on the choice of parameter. Thus when introducing new problems we will always define what parameter we choose to examine, even if e.g. the Independent Set problem can vastly differ depending on how we choose to parameterize it. Thus we will always mention, when defining a given parameterized problem, how we parameterize it. Once we have defined a problem, we will consistently view it through the defined parameterization.

3.2 Approaches to finding solutions

There are a number of general approaches that allow us to generate FPT algorithms for a great number of parameterized decision problems. While we cannot cover all past approaches, we will present some popular approaches and show a basic application that leads to tractability.

3.2.1 Kernelization

Intuitively kernelization tries to take a problem and gradually decrease the instance in size by removing superfluous information. In the vertex cover problem, for example, we know that any vertex with $k+1$ neighbours has to be included in any solution, as otherwise all of its neighbours need to be included and thus such a solution would exceed our required bound.
The goal is to find a set of rules such that, after fully exhausting them, the problem size is reduced to a small part that represents the hardest part of the instance, its 'kernel'. On this part we can then use more time consuming algorithms (such as brute force) and still retain, in total, FPT membership.

We formalize these informal descriptions in the following definition:

Definition 8. *Let $L \subseteq \Sigma^* \times \mathbb{N}$ be a parameterized problem. We call a function $f : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ a kernelization algorithm for L if there exist computable functions g, h such that*

1. *$(I, k) \in L$ if and only if $f(I, k) \in L$,*
2. *for $f(I, k) = (I', k')$ it holds that $|I'| \leq g(k)$ and $k' \leq h(k)$, and*
3. *$f(I, k)$ can be computed in polynomial time in $|(I, k)|$.*

We call (I', k') the kernel of (I, k).

As noted in [27], while we generally allow $k' > k$, in all actual instances we consider the parameter in the kernel is at most as big as in the original instance.

The existence of a kernelization algorithm immediately proves the existence of an FPT algorithm, as the size of the kernel is bounded by a function of the original parameter. Surprisingly the reverse is true as well:

Lemma 1 ([3])**.** *If a parameterized problem L is FPT then it admits a kernelization algorithm.*

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Proof. Since L is FPT there exists an algorithm A that correctly decides for an instance (I, k) whether $(I, k) \in L$ or not, with a runtime of $f(k) \cdot |(I, k)|^c$. We now run the first $|(I, k)|^{c+1}$ steps of A: If A terminates within that time, replace (I, k) with an equivalent instance of constant size. Otherwise we can conclude that $|(I, k)|^{c+1} < f(k) \cdot |(I, k)|^c$ and thus $|(I, k)| < f(k)$. Therefore (I, k) is already its own kernel, and our kernelization algorithm can simply output (I, k). □

While the proof relies only on the existence of an FPT algorithm A, it is only constructive if we already have an FPT algorithm for which we know the precise bounds on runtime. As such this proof is mainly of theoretical interest, as it shows there is an equivalence between a problem being FPT and admitting a kernelization algorithm and thus fixed parameter tractability could alternatively be defined through kernelization.

To show a simple application of problem kernels, we will consider the Edge Clique Cover Problem: We are given a graph $G = (V, E)$ and a parameter k, and (G, k) is a yes-instance if there are at most k cliques $C_1, \cdots, C_k \subseteq V$ in G such that for each edge $uv \in E$ there exists a clique C_j having $\{u, v\} \subseteq C_j$. We refer to this as the edge uv being covered by the clique C_j.

We will use 3 reduction rules to reduce the size of our instance [19]:

Rule 1: If v is an isolated vertex, delete v (the new instance is $(G - v, k)$

Rule 2: For an isolated edge uv (i.e. having $d(v) = d(u) = 1$), delete u and v and decrease k by 1, thus making the new instance $(G - \{u, v\}, k - 1)$.

Rule 3: For an edge uv such that $N[u] = N[v]$ (where $N[v] = N(v) \cup \{v\}$), delete v. The new instance is $(G - v, k)$. We only apply this rule where Rule 2 is not applicable.

Rule 1 is safe as there are no edges to cover using such vertices v. Rule 2 is safe as the only clique that can cover uv is the clique $\{u, v\}$, thus any solution must contain it. Rule 3 is safe as we can simply add v to all cliques containing u and thereby create a solution on the original instance.

We claim that after exhausting these rules, if $|V| > 2^k$, then (G, k) is a no-instance.

Proof. Let $C_1, \cdots, C_k$ be a solution for (G, k). Then, for each vertex $v \in V$, we can construct a vector $b_v \in \{0, 1\}^k$, where the i-th entry of b_v is 1 if and only if $v \in C_i$. By this construction there can be at most 2^k different vectors, so if there are more than 2^k vertices left, there exist $u, v \in V$ with $b_v = b_u$.

If, without loss of generality, v has a neighbour w that is not a neighbour of u, then the egde vw must be covered by some clique C_j in our solution. As u is not a neighbour of w, it cannot be contained in the same clique as w, thus $v \in C_j, u \notin C_j$ and therefore $b_v \neq b_u$. Thus u and v have the same neighbourhoods.

If $b_v = b_u = \vec{0}$, then u, v would be removed by Rule 1. Otherwise the egde uv must exist (as u, v are contained in some clique), and either $\{u, v\}$ forms an isolated component and is removed by Rule 2, or otherwise either u or v is removed by Rule 3. Thus there can be at most 2^k many vertices in the reduced instance. □

To decide whether there exists an Egde Clique Cover, we can simply go through all ways of choosing $\leq k$ cliques and check whether it constitutes a solution.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

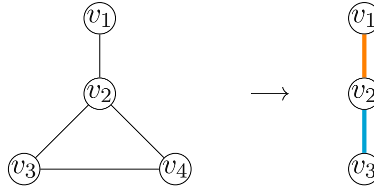

Figure 3.3: Using the reduction rules for Edge Clique Cover on the graph on the left would lead to the graph on the right, with the solution $C_1 = \{v_1, v_2\}, C_2 = \{v_2, v_3\}$ on the reduced instance for $k = 2$, coloured in orange and cyan.

3.2.2 Branch and Bound

We now want to consider the method of Branch and Bound. The basic idea is that we want to find a set of choices such that, if a solution for a given parameterized problem exists, we can find a solution that is represented by some set of decisions. More specifically, often we can build a solution step by step by going through a set of forced decisions.

Many graph problems ask for a subset of vertices (or edges) of size at most k that fulfill certain conditions, and a branch and bound algorithm could be constructed by finding (repeatedly) a set $W \subset V$ such that either no solution exists, or a solution S exists with $S \cap W \neq \emptyset$. Therefore we can set $Sol = \emptyset$ and follow a basic structure:

1. Find a set $W = \{w_1, \cdots, w_l\}$ that does not intersect with Sol but does intersect with a solution S of size $\leq k$ (if it exists),
2. for all $1 \leq j \leq l$ add w_j to Sol, and
3. check whether Sol is a solution to the problem.

By repeatedly applying these steps up to a depth of k we can find S or prove no such solution exists, and thus no solution exists at all.

The Algorithm for Vertex Cover at the start of the chapter is such an algorithm: in this case, for an edge uv not covered by Sol we have $W = \{u, v\}$ and thus can branch over this decision. Note that in general we do not need to add e.g. vertices to a solution one at a time, and better runtimes can often be achieved by considering more sophisticated ways of branching over choices. To illustrate this we will consider a different branching algorithm for Vertex Cover given in [9]. This algorithm is based on the following insights:

- If the maximal degree of a graph is 1, then Vertex Cover is trivial.
- For a non isolated vertex v, every vertex cover contains either v or all neighbours $N(v)$ of v.

The first claim is true as such a graph consists only of isolated vertices (which we can ignore) and isolated components with a single edge uv, where a solution can be found by including either endpoint for all edges.
For the second claim we can simply consider the original argument for a single edge uv on the collection of all edges adjacent to v: for each of them, either v is included in a solution or a given neighbour.

So let $G = (V, E)$ be a graph, and (G, k) be our instance of Vertex Cover. We will set $Sol := \emptyset$ and then repeatedly apply the following steps:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Let $v \in V$ be the vertex of maximal degree in G.

- If $k = 0$ and $d(v) \geq 1$, (G, k, Sol) is a no-instance.
- if $d(v) = 0$ then (G, k, Sol) is a yes-instance,
- if $d(v) = 1$ then (G, k, Sol) is a yes-instance if and only if $(G - v, k - 1, Sol \cup \{v\})$ is, and
- if $d(v) \geq 2$ recursively apply this algorithm to $(G - v, k - 1, Sol \cup \{v\}$ and $(G - N(v), k - d(v), Sol \cup N(v))$. (G, k, Sol) is a yes-instance if and only if either branch is a yes-instance.

To show why this algorithm is an improvement upon the simpler version, we consider how we generally arrive at an upper bound on the running time of a recursive algorithm: For a given computational tree, an upper bound is given by the number of nodes $\tau(k)$ (and therefore instances we consider) times the time spent for each. In our case, the time per instance is always polynomial, thus the runtime is bounded by $\tau(k) \cdot n^c$.

Let $T(k)$ be an upper bound on the number of leaf nodes in a computation tree $\mathcal{T}$ for an instance of size k. We can assume that $\mathcal{T}$ is a full binary tree to arrive at an upper bound. Then the number of total nodes in $\mathcal{T}$ is $2 \cdot T(k) - 1$, and thus there is no substantial difference in asymptotic runtime compared to $\tau(k)$. As $|N(v)| \geq 2$ we arrive at the following recursion:

$$T(k) = \begin{cases} T(k-1) + T(k-2), & k \geq 2 \\ 1, & \text{otherwise} \end{cases}$$

We can now inductively prove that $T(k) \leq 1.6181^k$: For $k = 0, 1$ this is trivial. For $k \geq 2$ we have

$$\begin{aligned} T(k) = T(k-1) + T(k-2) &\leq 1.6181^{k-1} + 1.6181^{k-2} \\ &= 1.6181^{k-2}(1.6181 + 1) \\ &\leq 1.6181^{k-2}1.6181^2 = 1.6181^k \end{aligned}$$

The value of 1.6181 comes from the fact that it is the unique positive root of the equation $x^2 - x - 1 = 0 \Leftrightarrow x^2 = x + 1$. We refer the reader to [9] for a general approach and theoretical background on how to arrive at upper bounds for such recursions and a helpful table for some common values.

3.2.3 Iterative Compression

The method of iterative compression [28] is a technique generally applied to minimization problems on graphs, where a small ($\leq k$) set of vertices or edges should be chosen to fit certain criteria, e.g. vertex cover. Following the terminology of [9], we want to solve $(*)$-COMPRESSION, where $(*)$ is the problem we originally want to solve. $(*)$-COMPRESSION is the task of taking a solution of size $k + 1$ of $(*)$ and either finding a solution of size at most k or concluding that no such solution exist. Usually the start of the algorithm is to take a small subgraph G' of our input, usually of size k, where the solution to the problem is trivial. Then we gradually increase the size of the graph, one vertex/edge at a time, which allows us to easily transform a solution X of size k into a

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

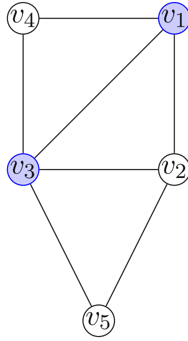

Figure 3.4: An Instance of Feedback Vertex Set and a solution of size 2, marked in blue.

solution of size $k+1$, upon which we then run the compression algorithm. We repeat these steps until we either find a solution for the entire graph or can conclude that none exist.

Instead of solving the $(*)$-COMPRESSION directly, we construct a subproblem $(*)$-DISJOINT: given a solution Z on a graph G of size $k+1$, find a disjoint solution of size k. More precisely, given a solution S of size $k+1$ and a partition $S = C \dot\cup F$, we want to find a solution S' of size k with $S \cap S' = C$.

So we want to be able to do the following 2 steps repeatedly:

1. Transform a solution of size k on G into a solution of size $k+1$ on $G \cup \{v\}$ (or $G \cup \{e\}$), and
2. given a solution S of size $k+1$, iterate over all ways of splitting S into F and C (so at most 2^{k+1} many possibilities) and then solve $(*)$-DISJOINT.

To show an example of Iterative Compression, we consider the Feedback Vertex Set problem [7]. For a graph $G = (V, E)$, we say a set $S \subset V$ is a feedback vertex set if $G - S$ is a forest. The Feedback Vertex Set problem, parameterized by k, is whether there exists a feedback vertex set S of size at most k.

Let $V = \{v_1, \cdots, v_n\}$. We will refer to the induced subgraph $G[\{v_1, \cdots, v_j\}]$ as G_j. It is clear that finding a feedback vertex set of size at most k on G_k or G_{k+2} is trivial, as we can simply choose to include everything (or all but any two vertices).

For a given solution S of size k on G_j, we can create a solution S' of size $k+1$ in G_{j+1} by defining $S' := S \cup \{v_{j+1}\}$. Trivially S' is a feedback vertex set on G_{j+1}, so to find a solution Sol of size k we consider all possible intersections $C = S \cap Sol$ and then consider the disjoint feedback vertex set problem on $(G - C, k - |C|)$. This will solve the original problem as $C \cup T$ is a feedback vertex set of G if and only if T is a feedback vertex set of $G - C$.

As we can remove the intersection, we now need to solve Feedback Vertex Set-DISJOINT. So we are given a graph G, a vertex cover W (of size $k+1$) and a parameter k (we will give such instances in the form (G, W, k)). Let $H := G - W$, we then use reduction rules to ensure structural properties of the problem instance:

Rule 1: Delete all the vertices of degree at most 1 in G.

Rule 2: If there exists a vertex v in H such that $G[W \cup \{v\}]$ contains a cycle, then

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

include v in the solution, delete v and decrease the parameter by 1, i.e. continue on $(G - v, W, k - 1)$.

Rule 3: If there is a vertex $v \in V(H)$ of degree 2 in G such that at least one neighbour of v in G is from $V(H)$, then delete this vertex and make its neighbours adjacent (even if they were adjacent before, the graph could become a multigraph now).

Rule 1 is safe, as no solution needs to contain a vertex of degree 1 (it can be contained in no cycle). Rule 2 is safe as we can only choose vertices in H, and if we do not choose v then we would need to choose some vertex in W. Rule 3 is safe as cycles are preserved and feedback vertex set including such a vertex v could instead include one of its neighbours. Additionally, all 3 rules can be applied exhaustively in polynomial time.

Lemma 2. *Feedback Vertex Set-DISJOINT is solvable in time* $4^k n^{O(1)}$.

Proof. Let (G, W, k) be an instance of Feedback Vertex Set-DISJOINT. If $G[W]$ is not a forest then (G, W, k) is a no-instance. So let $G[W]$ be a forest.
We first apply the reduction rules exhaustively in polynomial time (if this reduces k below 0 then it is a no-instance). We denote the reduced instance simply as (G, W, k) for clarity.
As W is a feedback vertex set, $G[H]$ is a forest. Thus H has a vertex v of degree at most 1 in $G[H]$. As v was not removed in the previous steps, v has at least 2 neighbours in W, and furthermore $W \cup \{v\}$ does not contain a cycle. We now branch over including v in a solution or not, so (G, W, k) is a yes-instance if and only if either $(G - v, W, k - 1)$ or $(G, W \cup \{v\}, k)$ is a yes-instance. We use these steps recursively and thus find a solution if one exists.
To bound the runtime of this procedure we will bound the maximal depth of the computation tree given through this branching. Let $I = (G, W, k)$ and define

$$\mu(I) = k + \gamma(I)$$

where $\gamma(I)$ is the number of components in $G[W]$. We note that none of the reduction rules reduce this measure, as rule 1 and 3 do not decrease k or affect the components in W and rule 2 decreases k by 1 but does not separate components in W (as there was a cycle including v).

In our branching, the vertices in W connected to v lie in different components, as rule 2 has not deleted v. Thus including v in W reduces the number of components by at least 1.

As $|W| \leq k + 1$, W contains at most $k + 1$ components, and thus $\mu(I) \leq 2k + 1$ and decreases by at least 1 every time we branch, thus the branching depth is limited by $2k + 1$, so the total number of instances we have to consider is bounded by $O(2^{2k+1})$ and thus we arive at the required running time. ◻

We will note that we therefore can bound the overall running time of this algorithm:

Theorem 6 ([9])**.** *Feedback Vertex Set can be solved in time* $5^k n^{O(1)}$.

Through a smarter construction (including an additional reduction rule), we can arrive at an even faster runtime.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Theorem 7 ([9])**.** *Feedback Vertex Set can be solved in time* $(1+\varphi^2)^k n^{O(1)} = 3.6181^k n^{O(1)}$*, where* φ *is the golden ratio.*

For Iterative Compression algorithms of this form (considering minimal vertex sets in graphs), there is an even more general way to arrive at such runtime bounds.

We will in later chapters touch on these and some other methods that can often be applied to find FPT algorithms for parameterized problems, but [9] and [27] provide a broader picture on the plethora of methods used.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Chapter 4

Enumeration Algorithms

Many computational problems are not, primarily, concerned with the search for a solution, but with the enumeration of all possible solutions. This can be because the principal task is to list all solutions within certain criteria (like database queries) or because there is no clear indication of optimality within the problem description and thus all candidate solution need to be listed to be able to make a decision based on characteristics that are hard or impossible to formalize. As noted in [30], problems in biology and chemistry (among others) require a set of solutions to be analyzed by experts.

From the classical viewpoint of complexity theory this creates a challenge: While finding a solution to a problem can potentially be done very quickly, listing all solutions can still take a lot of time. This can simply be because there are many solutions, each of which can be found within a reasonable timeframe, or the task of finding all solutions could be difficult because ensuring no solution is lost requires exploring too many possibilities (even when few solutions exist), or the complexity can grow with each solution found as the process of checking whether a solution has already been found is difficult (or at least time consuming). As there are different ways in which enumeration problems can be said to be difficult, classic notions of computational complexity are insufficient for classifying such problems. We base the content of this chapter on [30].

4.1 Formal Definitions

We will first define some basic (but less common) notions of complexity theory, as they more closely relate to the concept of enumeration algorithms than classical notions of complexity that apply to decision problems.

Definition 9. *Let Σ be a finite alphabet. A binary predicate $Q \subseteq \Sigma^* \times \Sigma^*$ is in FP if there exists an algorithm that, for a given $I \in \Sigma^*$, either finds a S such that $Q(I, S)$ holds or correctly concludes that none exist, and runs in polynomial time in $|I|$.*

FP is the functional equivalent to P. As the subject of enumeration problems is concerned with listing solutions explicitly we will mainly concern ourselves with the task of finding a (new) solution. We note that, for many problems in P and NP, solution algorithms do implicitly calculate a solution to decide if an instance is a yes-instance. As a negative example, for a matrix $A \in \mathbb{R}^{n \times n}$, deciding whether that matrix is invertible can be

TU WIEN Bibliothek Your knowledge hub

accomplished by calculating the determinant, which could not be translated into an algorithm for actually calculating A^{-1}.

Definition 10. *A problem $Q \subseteq \Sigma^* \times \Sigma^*$ is in TFNP if there exists an algorithm A that, for $I \in \Sigma^*$, decides in polynomial time if $(I, S) \in Q$, and all S with $Q(I, S)$ have $|S| \leq p(|I|)$ for some polynomial p. Additionally, for all I, $\{y : Q(I, S)\}$ is not empty.*

TFNP can be viewed as consisting of problems in NP∩coNP: For a problem Q in NP∩coNP, we can create a problem $\tilde{Q}$ consisting of instances x with either solutions, or certificate of unsolvability.

We will now formally define enumeration problems:

Definition 11. *Let Σ be a finite alphabet and let $Q \subseteq \Sigma^* \times \Sigma^*$ be a binary predicate. We will refer to the set $\{S \in \Sigma^* : (I, S) \in Q\}$ as $Q(I)$. The enumeration problem $Enum \cdot Q$ is the problem of listing $Q(I)$ on the input I. We will refer to the task of checking whether $S \in Q(I)$ for a given S as $Check \cdot Q$.*

To define the basic complexity class that encompasses all notions mentioned at the start, we first restrict ourselves to problems for which the size of any individual solution is small.

Definition 12. *We say a predicate Q is polynomially balanced if, for all $S \in Q(I)$, $|S|$ is polynomial in $|I|$.*

We can now define the class of problems that correspond to listing solutions for NP problems. Problems falling within this range will be the focus of enumeration problems we examine.

Definition 13. *We say a problem $Enum \cdot Q$ is in $EnumP$ if Q is polynomially balanced and $Check \cdot Q \in P$.*

The first notion of complexity we will consider is output complexity: we want to define the difficulty of an enumeration problem strictly by the size of the problem and its solution set.

Definition 14. *A problem $Enum \cdot Q$ is in $OutputP$ if there is an algorithm A such that there exists a polynomial p that, for all instances I of Q, outputs all solutions with a runtime bounded by $p(|I|, |A(I)|)$.*

Note that, for problem instances with a polynomial number of solutions, an $OutputP$ algorithm solves the enumeration problem in polynomial time with respect to the input. $OutputP$ clearly represents some tractability, but it is not immediately obvious if there are NP-hard problems that are not in $OutputP$. While P=NP obviously implies $OutputP = EnumP$, it turns out that the converse is true as well.

Theorem 8 ([4])**.** *$OutputP = EnumP$ if and only if P=NP.*

There are tasks for which total enumeration is not necessary, for instance if only a certain number of solutions should be generated, or until a solution fitting some informal criteria is found. In such a case generating all solutions takes potentially too much time, especially if the algorithm first generates a large data structure and can only find solutions at the end. The algorithms for such situations must still be up to the task of potentially generating all solutions and thus we cannot restrict ourselves to restricted tasks.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

One way of formalizing this intuition is using incremental time: we say an enumeration algorithm A runs in incremental time $f(m)g(n)$ if A enumerates the first m solutions for a problem x within $f(m)g(|I|)$ time.

Definition 15. *Let $Enum \cdot Q$ be an enumeration problem and A be an enumeration algorithm for $Enum \cdot Q$. For an instance I of $Enum \cdot Q$ we say that the time between outputting the i-th and $i+1$-st solutions in $Q(I)$ is the i-th delay. We define the 0-th delay as the precalculation time, i.e. the time spent until the first solution is output, and the n-th delay (for $n = |Q(I)|$) as the postcalculation time, i.e. the time spent between outputting the last solution and A terminating.*

Definition 16. *A problem $Enum \cdot Q$ in $EnumP$ is in $IncP_a$ if there is an algorithm outputting the first m solution within a time of $O(m^a|I|^b)$ for a constant b. Additionally we define Incremental Polynomial Time as $IncP = \bigcup_{a \geq 1} IncP_a$.*

An alternative way of defining incremental polynomial time is the following:

Definition 17. *A problem $Enum \cdot Q$ in $EnumP$ is in $UsualIncP_a$ if there is an algorithm for which there exist constants b, c such that the i-th delay is bounded by $ci^a|I|^b$ for all $0 \leq i \leq |Q(I)|$.*

Both of these definitions capture, in some sense, the increasing complexity of finding additional solutions. $UsualIncP_a$ more closely describes the task we were interested in at the outset, but it turns out that these descriptions are (almost) equivalent.

Theorem 9 ([5])**.** *For every $a \in \mathbb{N}$, $IncP_{a+1} = UsualIncP_a$.*

We can also relate the classes of $IncP$ and $OutputP$ to classical functional complexity classes:

Proposition 1 ([5])**.** *TFNP = FP if and only if $IncP = OutputP$.*

The proof of this proposition is based upon the problem $AnotherSol_Q$ of finding an additional solution to the problem Q and modifying it into a problem of TFNP.

Observe that we can also consider $IncP_a$ for $a \in \mathbb{R}_+$, and it trivially follows that $IncP_a \subseteq IncP_b$ for all $a \leq b$. To be able to strictly divide these classes within $IncP$, we need to additional assumptions, as P=NP implies $IncP = IncP_1$ and thus no clear separation can be made without proving (or implicitly assuming) P≠NP.
The Exponential Time Hypothesis (ETH) states that there exists a $\varepsilon > 0$ such that no algorithm can solve 3SAT with a asymptotic runtime of $O(2^{\varepsilon n} n^{O(1)})$ [30].

Theorem 10 ([5])**.** *If ETH holds, then $IncP_a \subsetneq IncP_b$ for all $a < b$.*

We will now define the complexity class we are most interested in, the class representing polynomial delay. Likewise, we will later define a corresponding class for FPT algorithms and focus our attention on such algorithms.

Definition 18. *A problem $Enum \cdot Q$ in $EnumP$ is in $DelayP$ if there is an algorithm A such that A outputs all solutions such that the time between outputting the i-th and $i+1$-st solutions*

Note that $DelayP = UsualIncP_0 = IncP_1$, and thus under the ETH we know that $DelayP \subsetneq IncP$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We will consider a related problem to the enumeration problem $Enum \cdot Q$: $ExtSol \cdot Q$ is the problem of deciding if, for a given I and ω_1, there exists a solution $\omega_1\omega_2 \in Q(I)$. We will later consider algorithms that solve such problems as flashlight algorithms, and use them to explore all (and only those) possibilities that contain solutions.

Proposition 2 ([5])**.** *If $ExtSol \cdot Q$ is in P, then $Enum \cdot Q$ is in $DelayP$.*

Recall that for FPT problems, even different NP-complete can fall into different complexity classes, as classical reductions between them cannot always be translated into a reduction that keeps the size of the parameter within polynomial bounds. Similarly we cannot guarantee that a given reduction of the corresponding enumeration problems allows us to solve the original problem.
If the construction of a smaller instance or instance of another problem necessitates that certain solutions are no longer present (and not easily reconstructable), we cannot use it in the context of enumeration problems. We want to restrict ourselves to reductions that keep solutions intact.

Definition 19. *Let $Enum \cdot Q_1$ and $Enum \cdot Q_2$ be two enumeration problems. A parsimonious reduction from $Enum \cdot Q_1$ to $Enum \cdot Q_2$ is a pair of functions f, g such that for all I, $g(I)$ is a bijection between $Q_1(I)$ and $Q_2(f(I))$.*

Clearly, the reduction from Independent Set to Clique is parsimonious in both directions (in fact, the solution sets are identical). However, we need to restrict the runtime such a transformation may take: For decision problems a 'reduction' that can take an arbitrary amount of time can simply solve the problem and then construct an equivalent instance of constant size. Thus we want to restrict bounds of runtime and delay on the functions f and g.

Definition 20 ([30])**.** *Let $Enum \cdot Q_1, Enum \cdot Q_2 \in EnumP$ and f, g two polynomial time computable functions. The pair (f, g) is a polynomial delay reduction if there is a polynomial p such that:*

- $\bigcup_{s \in Q_2(f(I))} g(s) = Q_1(I)$,
- $|\{s \in Q_2(f(I)) | g(s) = \emptyset\}| \leq p(|I|)$, *and*
- *for all* $s_1, s_2 \in Q_2(f(I))$, $g(s_1) \cap g(s_2) = \emptyset$.

Using these rules, we ensure that we can output all solutions (with no repetition), while only at most polynomially many solutions of the reduced instance do not correspond to a solution of the original instance (and thus bounding the delay).
In particular, a polynomial delay reduction from a problem Q to a problem in $DelayP$ allows us to construct a $DelayP$ algorithm for Q, akin to a similar result for parameterized reductions.

4.2 Approaches to finding solutions

In this section we want to broadly consider some methods that can often be used to generate enumeration algorithms with polynomial delay.

To contrast these methods with simple brute force approaches, we first want to consider a simple enumeration problem: Let's say we want to find all sequences s in $\{0, 1\}^n$ such

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

that the number of times 1 occurs in s is even. A simple brute force approach could solve the problem as depicted in Algorithm 1:

Algorithm 1 EvenSequences(n)

```
k = 0
while k < 2^n do
    find the binary representation s_k of k of length n
    if s_k has an even number of ones output s_k then
        Output s_k
    end if
    k = k + 1
end while
```

As this method is relatively efficient for any k and as there are as many sequences with an even number of ones as there are with an odd number (this can be proved by induction), this method will return all sequences within a time of $O(2^n n^{O(1)})$ and thus the algorithm is in $OutputP$, as there are 2^{n-1} such sequences. However, we cannot a priori be sure that there are no large gaps between any two solutions. We could alternatively try to find a method that ensures that only actual solutions will be found, and thus ensuring the delay is small. We note that, for this enumeration problem, the actual delay of a brute force approach is small (even constant) and thus it is in $DelayP$.

To improve this method, we can rely on a simple insight: Any sequence s with an even number of ones, say $2l$, consists of $0^j 1 s'$ for some j, where s' is a sequence of length $n - j - 1$ with $2l - 1$ ones (of course, this implies that $n - j - 1 \geq 2l - 1$, so we can restrict our search space). We can recursively apply this as a method of construction to output all of our sequences in lexicographic order using Algorithm 2:

Algorithm 2 EvenSequencesBetter(n)

```
k = 0
while k ≤ n do
    EvenSequencesOutput(n, k)
    k = k + 2
end while
```

Algorithm 3 EvenSequencesOutput(n, k)

```
if k = 0 then
    return 0^n
end if
for i ∈ {0, ··· n − k} do
    s = EvenSequencesBetter(n − i − 1, k − 1)
    return 0^i 1 s
end for
```

While this description is more complicated, and in our case not necessary, it shows a way of ensuring we only explore actual solutions. Moreover, if we restricted our task to finding a smaller subset of sequences (for example, with $\frac{n}{2}$ ones), we could still utilize

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

this structure with few changes to still get a polynomial delay, whereas the brute force method would first check a great number of unnecessary solutions (and without more careful analysis, check a great number of instances that can't contain solutions, in this case sequences that start with $> \frac{n}{2}$ ones).

We will consider a number of methods and examples from [24] in this chapter.

4.2.1 Backtracking

We first want to consider the method of Backtracking. In Backtracking, we try to go through the space of solutions depth-first lexicographically. We first consider an abstract description of how such an algorithm is structured (see Algorithm 4): We consider solutions which consist of subsets of U.

Algorithm 4 Backtrack(S)

```
Output S
Let π be the index associated to an element x ∈ U
for π(e) > max_{x∈S} π(x) do
    if S ∪ {e} is a solution then
        Backtrack(S ∪ {e})
    end if
end for
```

In this description, we only consider problems for which a subset of a solution is also a solution (or having some algorithm that allows us to check whether there exists a solution containing S in the case of e.g. finding maximal solutions), and so finds all solutions containing S one step at a time. We now want to consider a problem to which this framework can be applied.

Let $U = \{a_1, \cdots, a_n\}$ be a set of positive integers, and b be a positive integer. The Subset Sum Problem is to find all subsets $S \subset U$ whose sum is less than b. We can find all solutions by using the Algorithm 5:

Algorithm 5 SubsetSum(S)

```
Output S
Let i' be the element of S with the largest index
for i > i' do
    if a_i + Σ_{x∈S} x ≤ b then
        SubsetSum(S ∪ {a_i})
    end if
end for
```

Note that each solution takes a time of $O(n)$ to find and thus to output, and as the depth of this computation tree is bounded by n, and each branching step branches into at most n instances, we can guarantee a delay of $O(n^2)$. If we hold back some solutions (such that we output solutions at an even depth going down and at an odd depth going up), we can further decrease the guaranteed delay to $O(n)$. If we additionally order the elements of U first, we can further decrease the delay to $O(1)$.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We call a solution S for the Subset Sum problem maximal if no $S' \supset S$ is a solution. To this end we first sort the elements in U such that they are non-increasing. We give an algorithmic description for finding only maximal solutions in Algorithm 6:

Algorithm 6 MaximalSubsetSum(S, U, b)

```
if Σ_{i∈U} a_i ≤ b then
    return S ∪ U
end if
let a_i be the element in U with the smallest index
if a_i ≤ b then
    MaximalSubsetSum(S ∪ {i}, U\{i}, b − a_i)
end if
MaximalSubsetSum(S, U\{i}, b)
```

The idea of this algorithm is that, if all remaining elements can be included, then they must be included. Otherwise we branch once to not include the largest element, and once to include it if it does not exceed the bound. We needed the elements to be ordered as otherwise we could not guarantee that including all remaining elements forms a maximal solution.

Note that the problem of finding the maximum solution to Subset Sum is NP-hard (refered to as the Knapsack Problem). Part of the reason finding maximal solution is still, relatively, easy, is that checking subset-maximality is easy (can be done in linear time), but to find a maximum solution we need to examine not just remaining elements, but also explore other solutions.

As the depth of our computation tree is bounded by n and we output solutions at the leaves and need at most linear time at each node, we can guarantee a delay of $O(n^2)$.

4.2.2 Binary Partition

In binary partition we want to find efficient ways of partitioning the space of solutions, often by altering instances in ways that preclude solutions of different generated instances to intersect. In the example of finding certain sequences $s \in \{0, 1\}^n$, we could try to find all solutions starting with 1 with a remaining length of $n - 1$ and likewise with 0. In the more structured Algorithm 2 we partition the set of all solutions into those that start with $1, 01, 001, \cdots$. As the name implies, many natural ways of finding such algorithms split the instance into two instances.

We want to consider the $s - t$ Paths problem: We are given a graph $G = (V, E)$ and two vertices $s, t \in V$. The task is to enumerate all paths from s to t in G. We note that finding a single path can be done efficiently, so the main difficulty is ensuring we find every path and do not output duplicates.

Let (G, s, t) be an instance of $s - t$ paths. If there is no path from s to t, then this branch contains no solutions. Otherwise let sv be the first edge of an $s - t$ path. Every path from s to t either has sv as the first edge, or does not use it. We can find all instances of the first case by removing s and finding all paths from v to t, and the second case by simply removing sv. So we can generate the subproblems $(G - s, v, t)$ and $(G - sv, s, t)$, and enumerate all solutions to those problems. Note that for solutions of $(G - s, v, t)$ we

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

need to add sv as the first edge, and therefore we build the actual paths gradually using this approach (we can simply include a set S, initially $\emptyset$, that stores these edges). We can express this in pseudo-code as Algorithm 7. Note that the structure of this algorithm is aimed at paths specifically, and we would need to alter the branching and construction of subinstances to be able to handle trails (i.e. allowing us to visit a vertex multiple times, but using each edge at most once).

Algorithm 7 Paths(G, s, t, S)

```
if s = t then
    Output S
    return
end if
if There is no path from s to t then
    return
end if
choose an edge e = sv
Paths((G − s, v, t, S ∪ {sv}))
Paths((G − sv, s, t, S))
```

4.2.3 Reverse Search

We will now consider the technique of Reverse Search. It relies on constructing a parent-child relation between different solutions such that any solution has a unique parent solution and in such a way that this parent-child relation induces a directed acyclic graph between solutions. We can then, after finding a (set of) root solutions, output them and gradually traverse the entire solution forest/tree. An abstract description of this method is given in [2].

We want to construct a clear parent-child relationship that allows us to consider (and explore) the space of all solutions in the form of a tree. We then can solve the problem by applying the following algorithm structure:

Reverse Search(S)

```
Output(S)
for each S' = child(S) do
    Reverse Search(S')
end for
```

We require here that the child of any solution can be output within a reasonable time (in our case polynomial time), but do not otherwise restrict the structure or depth this method can take.

It is therefore sometimes necessary to hold back some solutions to ensure that the formal bounds of polynomial delay between any two solutions is achieved. One method of ensuring we never enter a drought of solutions (if, in the current branch, too many potential solutions must be checked that all contain no solutions). In this case we can expand this algorithmic structure to ensure that nodes at an odd depth of the computation tree are output on the way down, and nodes of even depth on the way up.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Reverse Search(S,d)

```
if d is odd then
    Output(S)
end if
for each S' = child(S) do
    Reverse Search(S',d+1)
end for
if d is even then
    Output(S)
end if
```

We now want to apply the method of Reverse Search to the problem of Maximal Clique Enumeration, that is, given a graph $G = (V, E)$, find all cliques C such that there is no clique C' in G with $C' \supset C$. As with the Subset Sum problem, the task of finding the clique of maximum size is an NP-hard problem, but finding a maximal clique can be done efficiently (by adding vertices one at a time until no vertex can be added anymore).

The main difficulty in solving the Maximal Clique Enumeration problem (using Reverse Search) is finding a suitable neighbourhood function relating a given solution to others.

So let $V = \{v_1, \cdots, v_n\}$ and let $S_{\leq i} = S \cap \{v_1, \cdots, v_i\}$ for any $S \subseteq V$. We define $C(K)$ as the lexicographically smallest clique including a clique K. Note that $C(K)$ can be efficiently computed by greedily adding vertices in lexicographic order.

Given a maximal clique, we define $P(K)$ as $C(K_{\leq i-1})$ such that i is the largest index satisfying $C(k_{\leq i-1}) \neq K$. It is clear that for the lexicographically smallest clique K_0, no parent $P(K_0)$ can exist. Otherwise we can compute $P(K)$ by gradually removing vertices until the smallest clique including $K_{\leq i-1}$ no longer matches K. As $P(K)$ is uniquely defined, it induces an acyclic graph.

We define $K[v_i] := C((K_{\leq i} \cap N(v_i)) \cup \{v_i\}$ for any vertex v_i and maximal clique K. Then a maximal clique K' is a child $C(K)$ of K if there exists an v_i with $K' = K[v_i]$.

Note that we can compute both $P(K)$ and $C(K)$ efficiently, in time $O(|E|)$, and all children of K can be found applying computation at most $|V|$ times. Thus we can find all maximal cliques using $O(|V||E|)$ time per solution.

Using this relation, we can now simply describe the algorithm as follows in Algorithm 8:

Algorithm 8 EnumMaximalCliques(G, K)

```
output K
for v ∈ V\K do
    K' = K[v]
    if P(K') = K then
        EnumMaximalCliques(K')
    end if
end for
```

Reverse Search can be viewed as a method traversing a graph, which uses a graph of all solutions to an enumeration problem to efficiently enumerate them. In particular, we

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

focus on a graph structure that turns the graph of solutions into a forest, but we can not necessarily expect that a natural way of defining a neighbourhood for solutions constructs a tree.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 5

Parameterized Enumeration

We have used the notion of parameterized complexity to be able to classify and solve NP-hard problems more efficiently (under some conditions), yet we have seen enumeration algorithms utilized only for problems that can be solved within polynomial time (as the enumeration problem is at least as hard as the associated decision problem).

We now want to synthesize concepts of parameterized algorithms and enumeration problems: Our primary area of interest will consist of problems that have a small delay (in this case FPT) between any 2 solutions, as we want to apply enumeration algorithms to FPT problems. To this end, we will first explore different complexity classes for parameterized enumeration (though for the most part, they mirror the complexity classes in the previous chapter) and some important theoretical results.

Our focus in this and the following chapters will algorithms with FPT delay, as they represent the optimum we can achieve for problems we consider (unless P=NP), as we consider (mostly) FPT algorithms for NP problems. We base the content of this chapter on [8].

5.1 Formal Definitions

Before we define the basic notion of parameterized complexity we concern ourselves with, we want to generally formally define parameterized enumeration problems:

Definition 21. *Let Σ be a finite alphabet and $Q \subseteq \Sigma^* \times \mathbb{N} \times \Sigma^*$ be a ternary predicate. For a given (I, k) we define $Q(I, k) = \{S \in \Sigma^* : Q(I, k, S) \text{ holds}\}$. We will refer to this set as $Sol(I)$ in cases where the context is clear.*

For such a predicate Q, we call the problem of outputting every element in $Sol(I)$ (exactly once) a parameterized enumeration problem.

We will call (I, k) an instance of the parameterized enumeration problem, and for simplicity we will refer to the problem as Q. We will occasionally write $(I, k) \in Q$ to mean $Q(I, k) \neq \emptyset$.

Definition 22. *Let Q be a parameterized enumeration problem. We say Q is in OutputFPT if there exists an algorithm A that, for an instance (I, k) of Q, outputs all elements in $Sol(I)$ in time $f(k) \cdot p(|I|, |Sol(I)|)$ for some polynomial p.*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Next, we will restate the notion of delay between solutions output by an algorithm A.

Definition 23. *Let Q be a parameterized enumeration problem and A be an enumeration algorithm for Q. For an instance (I, k) of Q we say that the time between outputting the i-th and $i+1$-st solutions in $Sol(I)$ is the i-th delay. We define the 0-th delay as the precalculation time, i.e. the time spent until the first solution is output, and the n-th delay (for $n = |Sol(I)|$) as the postcalculation time, i.e. the time spent between outputting the last solution and A terminating.*

We will briefly mention the notion of incFPT, mirroring $IncP$ for regular enumeration problems.

Definition 24. *Let Q be a parameterized enumeration problem and A an enumeration algorithm that solves Q. If, for any instance (I, k) of Q, there exists a computable function f and a polynomial p such that the i-th delay for A is given by $f(k) \cdot p(|I| + i)$, we say A is an incFPT-algorithm.*

We mention incFPT-algorithms and OutputFPT mainly to connect these parameterized problem classes with unparameterized enumeration complexity classes. As with enumeration complexity, our main focus will lie within algorithms that have a small delay.

We now formally define two closely connected classes of problems:

Definition 25. *Let Q be a parameterized enumeration problem and A an enumeration algorithm that solves Q.*

1. *If, for any instance (I, k) of Q, there exists a computable function f and a polynomial p such that A outputs all solutions in $Sol(I)$ within a time of $f(k) \cdot p(|I|)$, we say A is an FPT-enumeration algorithm.*
2. *If, for any instance (I, k) of Q, there exists a computable function f and a polynomial p such that A outputs all solutions in $Sol(I)$ with a delay of at most $f(k) \cdot p(|I|)$, we say A is an delayFPT-algorithm.*

We will also occasionally refer to delayFPT-algorithms as FPT-delay.

Definition 26. *The class delayFPT is the class of all parameterized enumeration problems that admit a delayFPT-algorithm.*

We can similarly define OutputFPT and incFPT as classes of problems.

Note that any delayFPT-algorithm for Q is also an FPT-enumeration algorithm when $Sol(I)$ has polynomially many or even $f(k)p(|I|)$ many solutions. If there are too many solutions, then no algorithm can have a total runtime in FPT.

While we cannot hope for a precalculation time below FPT (at least for NP-hard FPT problems), we could further distinguish between different delayFPT-algorithms based upon their delay past the 0-th. In particular, we want a class of problems that only allow a polynomial delay between solutions after the 1st.

Definition 27. *The class strictDelayFPT is the class of problems that admit enumeration algorithms with a polynomial delay and an FPT precalculation time.*

strictDelayFPT clearly represents an improvement on the basic notion of FPT-delay, but it turns out that any delayFPT-algorithm can be turned into a strictDelayFPT-algorithm:

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Theorem 11 ([8])**.** *delayFPT = strictDelayFPT.*

Proof. Let A be a delayFPT-algorithm for a parameterized enumeration problem Q and let (I, k) be an instance of Q. Let f be a computable function and p be a polynomial such that A outputs all solutions of (I, k) with a delay of $f(k) \cdot p(|I|)$. We define a function $g : \mathbb{N} \to \mathbb{N}$ by

$$g(m) = \max_{(I',k') \in \Sigma^* \times \mathbb{N}, |(I',k')| \leq m} \{s | s \text{ is the running time of } A \text{ on input } (I, k)\}.$$

We can construct an algorithm A' for Q with polynomial delay as follows: On the input (I, k) A' checks whether $|(I, k)| \geq f(k)$. If so, then A already has polynomial delay (as $f(k)p(|I|) \leq |(I, k)| \cdot p(|I|)$) and thus A' simply simulates the algorithm A. Otherwise B simulates A in the precalculation time and finds $Sol(I)$ within a time of $g(f(k))$ and then simply outputs the contents of $Sol(I)$. □

We note that, although the result establishes an important theoretical equivalence, in practice a natural way of enumerating all solutions with a polynomial delay is obviously preferable to one that requires FPT time for each solution.

However, we cannot meaningfully differentiate between the two, and moreover delayFPT represents, for many applications and especially for the problems we will consider, optimality. Thus our focus will primarily be on FPT-delay (and occasionally FPT-enumeration). We note here that many of the problems we consider (and many we do not) can trivially be solved within XP time by using a brute force approach.

We will now briefly state the method of Enum-Kernelization and highlight a theoretical result that mirrors a similar result for FPT algorithms.

Definition 28 ([8])**.** *Let Q be a parameterized enumeration problem over Σ. A polynomial time computable function $K : \Sigma^* \to \Sigma^*$ is an enum-kernelization of Q if there exist:*

1. *computable functions $h, t : \mathbb{N} \to \mathbb{N}$ such that for all $I \in \Sigma^*$ we have $((I, k) \in Q \Leftrightarrow (K(I), t(k)) \in Q)$ and $|K(I)| \leq h(k)$,*
2. *a computable function $g : \Sigma^* \times \mathbb{N} \times \Sigma^* \to \mathcal{P}(\Sigma^*)$, which from a pair (I, ω) where $(I, k) \in Q$ and $\omega \in Sol(K(I))$, computes a subset of $Sol(I)$, such that*
   (a) for all $\omega_1, \omega_2 \in Sol(K(I))$, $\omega_1 \neq \omega_2 \Rightarrow g(I, k, w_1) \cap g(I, k, \omega_2) = \emptyset$,
   (b) $\bigcup_{\omega \in Sol(K(I))} g(I, k, w) = Sol(I)$, $g(I, k, w) = \emptyset$ if $w \notin Sol(K(I))$, and
   (c) there exists an algorithm $\mathcal{A}_g$, which on input (I, ω), where $(I, k) \in Q$ and $\omega \in Sol(K(I))$, enumerates all solutions of $g(I, k, \omega)$ with delay $p(|I|) \cdot f(k)$, where p is a polynomial and f is a computable function.

Like with usual kernelization, a problem has a delayFPT-algorithm if and only if there exists an Enum-Kernelization:

Theorem 12 ([8])**.** *For every parameterized enumeration problem Q, the following are equivalent:*

1. *Q is in delayFPT.*
2. *Q has an enum-kernelization.*

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Proof. Let K be an enum-kernelization for the problem Q computable in time $p(|I|)$ for some polynomial p and every instance (I, k) of Q, and let $Sol(K(I))$ be computable in time $g'(|K(I)|)$ for some computable function g'. Let h, g, f be as in the definition above.

We can generate an FPT-delay algorithm A for Q as follows: For an instance (I, k) of Q

1. compute $K(I)$ in polynomial time $p'(|I|)$,
2. compute $Sol(K(I))$ within time $g'(h(k))$, and
3. for each solution ω in $Sol(K(I))$, use A_g to compute $g(I, k, \omega)$ with FPT-delay $f(k) \cdot p(|I|)$.

Both the precomputation time and the delay is bounded by

$$p'(|I|) + g'(h(k)) + f(k) \cdot p(|I|) \leq (g'(h(k) + f(k)) \cdot (p'(|I|) + p(|I|))$$

and thus we have an FPT-delay algorithm for Q.

Now let Q have an FPT-delay algorithm A with a delay of $f'(k) \cdot p'(|I|)$ for a computable function f' and a polynomial p' and let (I, k) be an instance of Q. Let (I', k') be such that $(I', k') \in Q$ and $\omega' \in Sol(I')$, and let (I'', k'') be an instance with $Sol(I'') = \emptyset$. If no such I' or I'' exist, then there is a trivial kernelization by mapping to $(\epsilon, 1)$. For $Sol(\epsilon) = \emptyset$ we can trivially construct $g(I, k, \omega) = \emptyset$, and otherwise for some $\omega \in Sol(\epsilon)$ we define $g(I, k, \omega) = Sol(I)$ and $g(I, k, \omega') = \emptyset$ otherwise.

We can now compute an enum-kernelization for Q on an instance (I, k), with $n = |I|$, by using the following steps:

1. We simulate the first $p'(n) \cdot p'(n)$ steps of A,
2. if A stops without outputting a solution, set $K(I) = I''$,
3. if a solution ω is found, set $K(I) = I'$ and $g(I, k, \omega) = Sol(I)$ (and $g(I, k, \omega') = \emptyset$ otherwise), and
4. if no solution is found and the algorithm has not stopped, it holds that $p'(n) \leq f'(k)$, thus we set $K(I) = I$ (as $|I| \leq f'(k)$).

Since K in this case can be computed in $O(p'(n)^2)$, and $|K(I)| \leq f'(k) + |I'| + |I''|$, we have a kernelization which fulfills all requirements for an enum-kernelization algorithm. □

We will briefly mention that not every FPT decision problem has an FPT-delay enumeration algorithm for its associated enumeration problem (under the assumption that W[1] $\neq$ FPT). To see this, we consider a W[1]-hard problem, like the Independet Set problem: We then construct a related problem, which we will call the Independent Set or Empty problem, in which we are given a graph $G = (V, E)$ and a parameter k, and $S \subseteq V$ is a solution if either S is an independent set of size at least k or $S = \emptyset$.

Finding a solution to this problem is trivial, so it is in P and therefore FPT. Let us assume there was an FPT-delay enumeration algorithm A for it with a delay of $f(k)n^{O(1)}$ for some computable function f. Either the first or second solution A finds cannot be the empty set, and thus we have found an independent set of size $\geq k$ within a time of $\underbrace{2 \cdot f(k)}_{:= f'(k)} n^{O(1)}$, so the Independent Set problem is in FPT, which contradicts our assumption.

This example is artificial, and relies on the knowledge that the underlying problem (past the first, trivial solution) is not FPT, but it highlights that we cannot in general hope to find FPT-delay enumeration algorithms for every FPT problem we examine.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 6

Enum-Kernelization

We first consider the technique of Enum-Kernelization, which extends the method of Kernelization to FPT-Delay enumeration algorithms and was first studied in [8]. On a basic level, we combine a kernelization algorithm with a method that uses the solutions on the kernel (which we can find efficiently, as the kernel size depends only on k) to find all solutions of the original instance with at most FPT-delay. We repeat the abstract description from [8]:

Let (Q, κ, Sol) be a parameterized enumeration problem over Σ. A polynomial time computable function $K : \Sigma^* \to \Sigma^*$ is an enum-kernelization of (Q, κ, Sol) if there exist:

1. a computable function $h : \mathbb{N} \to \mathbb{N}$ such that for all $I \in \Sigma^*$ we have $(I \in Q \Leftrightarrow K(I) \in Q)$ and $|K(I)| \leq h(\kappa(I))$,
2. a computable function $f : \Sigma^* \times \Sigma^* \to \mathcal{P}(\Sigma^*)$, which from a pair (I, ω) where $I \in Q$ and $\omega \in Sol(K(I))$, computes a subset of $Sol(I)$, such that
   (a) for all $\omega_1, \omega_2 \in Sol(K(I))$, $\omega_1 \neq \omega_2 \Rightarrow f(I, w_1) \cap f(I, \omega_2) = \emptyset$,
   (b) $\bigcup_{\omega \in Sol(K(I))} f(I, w) = Sol(I)$, $f(I, w) = \emptyset$ if $w \notin Sol(K(I))$, and
   (c) there exists an algorithm $\mathcal{A}_f$, which on input (I, ω), where $I \in Q$ and $\omega \in Sol(K(I))$, enumerates all solutions of $f(I, \omega)$ with delay $p(|I|) \cdot t(\kappa(I))$, where p is a polynomial and t is a computable function.

Intuitively, h represents the kernelization routine, $K(I)$ the kernel of the instance I, and f is the method by which we connect a solution in the kernel $K(I)$ to a set of solutions for I. In the problems we examine in this chapter, f describes the way we add unnecessary information (e.g. vertices/edges) to a given solution.

6.1 Examples

We consider some FPT problems where the structure of a kernelization algorithm can be naturally extended into an Enum-Kernelization algorithm.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.1.1 Vertex Cover

We again consider the problem of Vertex Cover parameterized by the size of the solution. Instead of the decision variant we will now consider the associated enumeration Problem: Finding all (inclusionwise minimal) vertex covers of size at most k. We follow the kernelization algorithm in [9]:

We use the following 2 reduction rules until exhaustion:

1. If a vertex v has no neighbours, delete it
2. If a vertex v has $\geq k+1$ neighbours, delete v and decrease k by 1

We will now prove the safety of these reduction rules. If a vertex v has no edges adjacent to it, removing it from any vertex cover containing it would never cause an edge to not be covered, so we can safely remove them.
If v has at least $k+1$ neighbours, to cover all edges adjacent to v, we either need to add v to a vertex cover C or $N(v)$. As $N(v) \geq k+1$, we cannot add it to C as it would exceed our bound k, and since any vertex cover thus has to include v we can delete it and decrease k without affecting solvability. More specifically, any vertex cover C of the reduced instance corresponds to exactly one vertex cover $C \cup \{v\}$ of the original instance. Note that we can check each of these rules in time $O(n)$, and every time we either conclude that we are done or remove a vertex and thus will be applied $\leq n$ times, so this process can be done in time $O(n^2)$.

We now claim that after exhausting these rules, if the reduced graph $G' = (V', E')$ has more than $k^2 + k$ vertices, it is a no-instance.

Proof. As any vertex can have at most k neighbours and each vertex is adjacent to an edge, every vertex can at most cover all edges adjacent to a total of $k+1$ vertices. Since our vertex cover must cover all edges, and has size $\leq k$, we can thus have at most $k(k+1) = k^2 + k$ vertices in G'. □

As the size of G' is bounded by a function of k, we can simply go through all $\leq 2^{k^2+k} =: f(k)$ subsets of V' and check whether they form a vertex cover of size $\leq k$ (This can obviously be done more efficiently, as we only need to check subsets of size $\leq k$).

We can now utilize G' to output all (inclusion-wise minimal) vertex covers of G: If we want minimality, find all inclusion-wise minimal vertex covers of G' (we can check minimality efficiently by seeing if removing any vertex will make it no longer a vertex cover). Then add all vertices removed in rule 2, and output them.
If we instead want to find all vertex covers, we take all vertex covers of G', again add the vertices removed in rule 2 and then add all possible combinations of unchosen vertices in $V \backslash V'$ until we reach size k. Any 2 vertex covers discovered this way will either differ on V' (if they are constructed from different vertex covers on G') or on $V \backslash V'$ (if they come from the same vertex cover on G' but by adding different vertices).
Lastly, as every vertex cover C of G remains a vertex cover when restricted to G', C will be constructed from $C|_{G'}$.

For any output solution, we can either find the next solution by finding a new solution on G' or adding a new combination of superfluous vertices to an existing solution, both

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

of which can be done in time $\leq f(k)$. So we have a preprocessing time of $n^2 f(k)$ and then a delay of $f(k)$.

6.1.2 Feedback Arc Set in Tournaments

A Tournament is a directed graph T such that for every vertex pair $u, v \in V(T)$ either $(u, v) \in E(T)$ or $(v, u) \in E$. A feedback arc set is a set $A \subset E(T)$ s.t. $(V(T), E(T) \backslash A$ contains no directed cycles. The Feedback Arc Set in Tournaments problem parameterized by a natural number k is whether a given Tournament T has a feedback arc set of size $\leq k$ [9]. For our enumeration problem, we want to find all minimal feedback arc sets of T.

For $F \subset E(T)$ we define $rev(F) = \{(v, u) : (u, v) \in F\}$ and $T * F = (V(T), (E(T) \backslash F) \cup rev(F))$. We now first proof the following: $A \subset E(T)$ is an inclusion-wise minimal feedback arc set for T iff A is an inclusion-wise minimal set of edges s.t. $T * A$ contains no cycles.

Lemma 3 ([9])**.** *Let G be a directed graph and F be a subset of $E(G)$. Then F is an inclusion-wise minimal feedback arc set of G if and only if F is an inclusion-wise minimal set of edges such that $G * F$ is an acyclic directed graph.*

Proof. Let $G = (V, E)$ be a directed graph Let F be an inclusion-wise minimal feedback arc set of G. Let us assume that $G * F$ has a directed cycle C and let $f_1, \cdots, f_l$ be the edges in $C \cap rev(F)$ ordered by their appearance in C. As F is a feedback arc set, we know that C cannot consist only of edges in $E \backslash F$ and thus $C \cap rev(F)$ is not empty. Let $e_i \in F$ be the reversed edge of f_i, then by the minimality of F for every e_i there exists a directed cycle C_i with $C_i \cap F = \{e_i\}$ (if every cycle containing e_i also contained another element in F, we could remove e_i from F and find a smaller feedback arc set). We can then construct a closed walk W in G as follows: we follow the cycle C, but when we would traverse and edge f_i we instead traverse $C_i - e_i$, and thus we can construct a cycle using W that does not contain edges in F, thus contradicting our assumption. Therefore $G * F$ does not contain directed cycles.

Let us now assume that F is a minimal set such that $G * F$ does not contain directed cycles, then F must also be a feedback arc set (as $G * F$ is a subgraph of $G - F$). Thus, if F is not an inclusion-wise minimal feedback arc set and contains a feedback arc set $F' \subset F$, then by the other direction of this proof $G * F'$ does not contain a directed cycle, contradicting the minimality of F. □

At first glance, there seems to be no real point to using this theorem as it merely shifts one problem into an equivalent one. However, the crucial property that is preserved is that a graph with reversed edges remains a tournament, thus we can guarantee that after each step of our reduction rules we still remain a tournament. Without this property, checking for directed cycles could take increasingly long and we could no longer guarantee a short execution time.

We follow the kernelization of the book:

Theorem 13 ([9])**.** *Feedback Arc Set in Tournaments admits a kernel with at most k^2+2k vertices.*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Proof. The previous Lemma implies that the task of finding a minimal feedback arc set can be reframed as a search for a minimal set of edges to reverse to eliminate all cycles. A tournament contains a directed cycle iff it contains a triangle, a directed cycle of length 3. We therefore consider the following reduction rules:

Rule 1: If an edge e is contained in at least $k+1$ triangles, reverse e and reduce k by 1.

Rule 2: If a vertex v is not contained in any triangle, delete v from T.

Rule 1 is safe as no two triangles in a tournament can share more than a single edge, so if e wasnt reversed, at least one edge of each of the $k+1$ triangles would have to be reversed, which would exceed our capacity. To proof the safety of Rule 2, we define $N_+(v)$ as the set of all vertices that can be reached from v, and $N_-(v) = V(T)\backslash(N_+(v) \cup \{v\})$. Let X_1 be the subgraph induced by $N_+(v)$ and X_2 be the subgraph induced by $N_(v)$. As v is contained in no cycle, (X_1, X_2) is a partition of $V(T)\backslash\{v\}$. By construction, no edge goes from X_1 to X_2, so no edge running from X_2 to X_1 will be contained in a cycle and thus cannot be contained in a minimal feedback arc set. This means that any minimal feedback arc set A can be split into A_1 and A_2, where A_i only contains edges of X_i, so v will never be used and as such can be deleted.

After exhaustively applying these rules, if the graph contains more than $k^2 + 2k$ vertices we can conclude that there is no solution. This is because any edge e can be contained in at most k triangles due to Rule 1. In addition to the vertices adjacent to e, every one of these triangles contains an additional vertex, so every deleted edge the graph can contain at most $k+2$ vertices. □

In this reduced instance we can now exhaustively go through every possible subset of vertices and check whether they form a minimal feedback arc set. As this instance only depends on k, this takes $2^{O(k^2)} \cdot k^{O(1)}$ time. Our solutions then consist of every solution on the kernel together with the reversed edges from Rule 1. The total time of this algorithm, including the reduction rules, is $2^{O(k^2)} \cdot k^{O(1)} n^{O(1)}$, which also implies FPT delay.

6.1.3 d-Hitting Set

For a family $\mathcal{A}$ of subset of a set U, a hitting set S is a set that has non-empty intersection with every $A \in \mathcal{A}$. The d-Hitting Set Problem, parameterized by k, is to decide if a given family $\mathcal{A}$ which contains only subsets of size at most d of a universe U, admits a hitting set of size $\leq k$.

We closely follow the reduction rule in [9] (from [14]). To define the reduction rule we first need the following constructions:

Definition 29. *A sunflower with k petals and core C is a family of k sets $S_1, \cdots, S_k$ with $S_i \cap S_j = C$ for $i \neq j$.*

Theorem 14 ([13])**.** *Let $\mathcal{A}$ be a family of sets (without duplicates) over a universe U, such that each set in $\mathcal{A}$ has cardinality exactly d. If $|\mathcal{A}| > d!(k-1)^d$, then $\mathcal{A}$ contains a sunflower with k petals and such a sunflower can be computed in time polynomial in $|\mathcal{A}|$, $|U|$, and k.*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Proof. We proof the statement inductively on d. For $d = 1$ the statements holds trivially. Now let $d \geq 2$ and $|\mathcal{A}| \geq d!(k-1)^d$, and let $\mathcal{G} = \{S_1, \cdots, S_l\} \subset \mathcal{A}$ be an inclusion-wise maximal set of pairwise disjoint sets in $\mathcal{A}$. If $l \geq k$, then $\mathcal{G}$ is a Sunflower of size k with core $\emptyset$, so we assume $l < k$. Then by construction $S = \bigcup_{i=1}^{l} S_i$ fulfills $|S| \leq d \cdot (k-1)$, and because $\mathcal{G}$ is maximal every set $A \in \mathcal{A}$ has nonempty intersection with S. Therefore, there exists an $u \in U$ contained in at least

$$\frac{|\mathcal{A}|}{|S|} > \frac{d!(k-1)^d}{d(k-1)} = (d-1)!(k-1)^d - 1$$

different sets in $\mathcal{A}$. By taking these sets and removing u from each, we get a family $\mathcal{A}'$ of sets of size $d-1$ of size $> (d-1)!(k-1)^{d-1}$. By the induction hypothesis, there exists a sunflower $\{S'_1, \cdots, S'_k\}$ with core C' in $\mathcal{A}'$. Therefore $S'_1 \cup \{u\}, \cdots, S'_k \cup \{u\}$ is a sunflower in $\mathcal{A}$ with core $C' \cup \{u\}$.

We can also use the steps of this proof as a polynomial time Algorithm to find such a sunflower by recursion. □

We now separate $\mathcal{A}$ into subfamilies of sets of the same size $\mathcal{A}_{d'} = \{A \in \mathcal{A} | |A| = d'\}$. If any $\mathcal{A}_{d'}$ contains more than $d'! \cdot k^{d'}$, then there exists a sunflower of size $k+1$ with core C. As we can choose only k elements, and any 2 petals only intersect in C, we can replace the $k+1$ petal sets with C, because every solution must take an element of C, which also covers the petals. If $C = \emptyset$, then the instance has no solution.
If all $\mathcal{A}_{d'}$ no longer contain sunflowers, our instance only contains $d'! \cdot k^{d'}$ for every $\mathcal{A}_{d'}$, so in total $\leq d \cdot d! \cdot k^d = f(k)$ many elements. On this kernel, we can simply try all possible solutions ($\leq 2^{f(k)}$) and output them if they constitute a hitting set. If we only want e.g. subset minimal solutions, we can check minimality before outputting each solution.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 7

Binary Partition

The method of binary partition consists of transforming an instance I of a problem into two (or more) instances I_1, I_2 s.t. $\mathrm{Sol}(I) = \mathrm{Sol}(I_1) \dot\cup \mathrm{Sol}(I_2)$, and applying this transformation repeatedly until either the solution sets have size 1 (or sometimes size 0) or an efficient algorithm can output all solutions to a given instance with a small delay. Usually this efficient algorithm consists of taking a given solution and adding unnecessary information in a structured way, see section 7.1.

We will motivate the method of Binary Partition by analyzing the simple Branch and Bound algorithm for Vertex Cover at the beginning of Chapter 3:

We are given a graph $G = (V, E)$ and the parameter k. We will set $Sol = \emptyset$ and then repeatedly apply the following steps:

- If $E = \emptyset$, then (G, k, Sol) is a yes-instance,
- for any $uv \in E$, recursively apply this algorithm to $(G - u, k - 1, Sol \cup \{u\})$ and $(G - v, k - 1, Sol \cup \{v\})$, and (G, k, Sol) is a yes-instance iff either branch is a yes-instance,
- if $k = 0$ and $E \neq \emptyset$, (G, k, Sol) is a no-instance.

The key insight from this algorithm is that every solution contains at least one end vertex of any given edge, thus every solution must exist on either side of the computation tree. However, this algorithm is not quite fit for Binary Partition, as a solution that contains both will be present on both sides.

Throughout this section, we will colour our graphs using red and blue to indicate whether a choice is allowed and therefore excluding choices that were made in another branch, thereby ensuring

7.1 Branch and Bound

In this section we mostly consider a natural extension of given FPT algorithms that consider, in some sense, all available possibilities, and thereby we construct an enumeration algorithm. As noted in the introduction of this chapter, the structure of Vertex Cover, for example, allows us to simply take an existing FPT algorithm and use it for enumeration.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

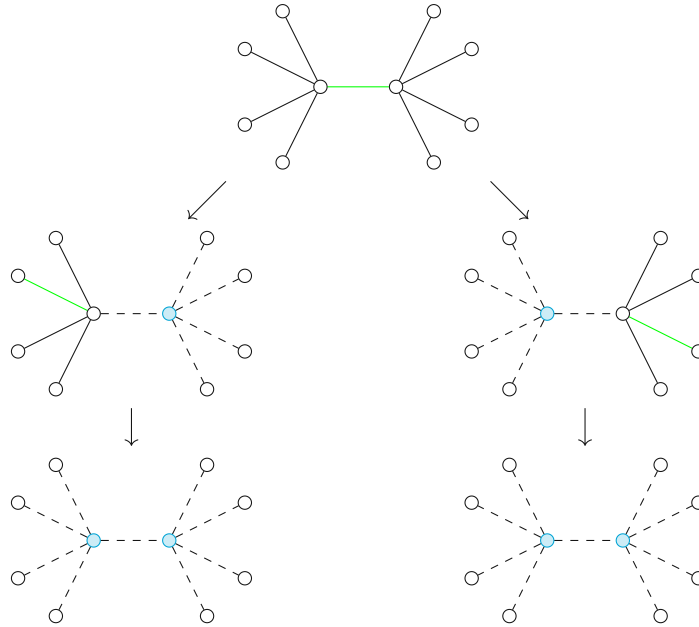

Figure 7.1: Searching for a vertex cover of size 2 in this graph would lead to the only solution coming up in 2 different branches of the FPT algorithm.

Some FPT algorithms use reduction rules to reduce the size of an instance and thereby often 'squish' several solutions into a single one, and prevent a clear way of reconstructing all solutions.

7.1.1 Vertex Cover

We again consider the problem of Vertex Cover, and the associated enumeration Problem: Finding all (inclusion-wise minimal) vertex covers of size at most k.

We start by colouring all vertices of our graph blue. We will gradually colour vertices red to ensure they will no longer be chosen in a given branch.
Since we know that for any edge $e = uv$ every vertex cover either contains u, v or both. Based on this, we construct a branching algorithm as follows:

We choose a random edge $e = uv$. If both u and v are coloured red, we return that the current branch is not solvable. Then we branch over 3 possibilities:

- if v is blue, delete v, adding it to the solution set, colour u red and decrease k by 1
- if u is blue, delete u, adding it to the solution set, colour v red and decrease k by 1
- if both vertices are blue, adding both to the solution set and decrease k by 2

At every step of the branching we check whether there are any edges left, and stop if the set is already a vertex cover. This branching will result in a tree of depth $\leq k$, with a branching factor of at most 3. Every minimal vertex cover will be in one branch of this tree, but as not every branch will lead to a minimal vertex cover, we need to check at the end of the branching if any vertex can be dropped s.t. the solution remains a vertex cover, and if so, do not output it. So the total time of this algorithm is $3^k \cdot k \cdot n^{O(1)}$. Note that this also implies that there are at most $O(3^k)$ different inclusion-wise minimal vertex covers of size $\leq k$. More precisely, as our branching factor of size 3 was mainly

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

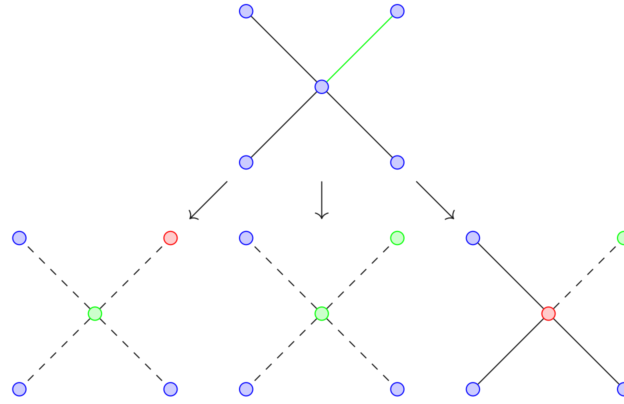

Figure 7.2: Searching for all inclusion-wise minimal vertex covers of size ≤ 2 using this algorithm will also find a non-minimal solution, and some branches will not lead to a solution. Green edges and vertices represent the chosen edge and solution vertices along each branch, dashed edges are covered by the current solution set.

used to avoid overlap, we can also find all inclusion-wise minimal vertex covers with a binary branching tree up to depth k, so there can in fact at most be $O(2^k)$ many such vertex covers.

By colouring unchosen vertices red, we can ensure that 2 different branches will not output the same solution. More specifically, any two leaves of the branching tree will contain solutions that have chosen vertices that the other has coloured red. If we now want to find all vertex covers, we can take all leaves of the tree and add all possible combinations of blue vertices to the leaf solutions (e.g. ordered lexicographically and by size) up to a total size of at most k.

7.1.2 Feedback Vertex Set in Tournaments

The Feedback Vertex Set in Tournaments problem, parameterized by k, is whether for a given Tournament T there exists a set X of size $\leq k$ s.t. $T - X$ has no directed cycles. In the associated enumeration problem, we are interested in all (inclusion-wise minimal) Feedback Vertex sets.

We will use the same observation as we did for feedback arc sets: if there is a cycle in a tournament, there also is a cycle of size 3. So we can in time $O(n^3)$ either find a triangle or conclude none exist. We will now proceed in a manner similar to a Vertex Cover:

We set $S := \{\}$, and colour all of our vertices blue. Then we repeatedly apply the following steps:
We find a directed triangle in $V \backslash S$. If none exist, check for all $v \in S$ if $S \backslash \{v\}$ is also a feedback vertex set. If not, output S.
Otherwise, for a triangle $T = \{v_1, v_2, v_3\}$, we branch over all possibilities a feedback vertex set can intersect with T:

- if v_i is blue, add it into S, colour the other vertices red and decrease k by 1,
- if v_i, v_j are blue, add them into S, colour the remaining vertex red and decrease k by 2,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

- if all vertices are blue, add them into S and decrease k by 3.

As with Vertex Cover, in this branching any 2 leaves will have chosen vertices that the other has coloured red, and thus if we want all possible feedback vertex sets of size at most k, we can simply add unchosen blue vertices to any leaf solution and output them in a structured manner.

As we have branching factor of up to 7 with a depth $\leq k$, and need to find a triangle for every step, we can find all inclusion-wise minimal feedback vertex sets in total time $O(7^k n^3)$. As we cannot guarantee how many solutions there are, we cannot ensure a delay less than that, even in the general case.

More precisely, we can construct arbitrarily big instances of Feedback Vertex Set in Tournaments that have only a single solution. We can construct a tournament with only a single feedback vertex set of size 1 as follows: Let $V_1 = \{v_1, \cdots, v_{n_1}\}$, $V_2 = \{w_1, \cdots, w_{n_2}\}$ and let $V = V_1 \cup V_2 \cup \{u\}$. We construct a tournament $T = (V, E)$ by choosing

$$E = \{v_i v_j : i < j\} \cup \{w_i w_j : i < j\} \cup \{v_i u : i \leq n_1\} \cup \{u w_i : i \leq n_2\} \cup \{w_i v_j : \forall i, j\}.$$

Intuitively, we first create edges along a linear ordering $(w_1, w_2, \cdots, w_{n_2}, v_1, \cdots, v_{n_1})$ (i.e. $x_1 x_2 \in E$ if and only if $x_1 < x_2$ for all $x_1, x_2 \in V_1 \cup V_2$) and use the vertex u to create a bridge back from any v_i to w_i, thereby forming triangles.

This is a tournament, as between any two vertices there exists a directed edge. All triangles in this graph are of the form $\{u, w_i, v_j\}$ (as V_1 and V_2 are linearly ordered, and all edges between them are from V_2 to V_1), and moreover all such sets of vertices are triangles. If both n_1 and n_2 are ≥ 2, then $\{u\}$ is the only Feedback Vertex Set of size 1 (see 7.3 for an example of such a graph).

To now construct a graph having only a single solution of size k, we take k copies of this Tournament $T_1, \cdots, T_k$ and for vertices $v_1' \in T_i$ and $v_2' \in T_j$ include the edge $v_1' v_2'$ if and only if $i < j$. This way, the only triangles can exist within a given T_i, and by construction in any T_i we need to take its copy u_i of u, so $\{u_1, \cdots, u_k\}$ is the only Feedback Vertex Set of size $\leq k$.

7.2 Flashlight Algorithms

Flashlight algorithms use the same basic structure of branching as Branch and Bound, but additionally uses a flashlight routine that determines quickly whether the current branch will contain solutions or not. As we consider only problems known to be FPT, we will use existing FPT algorithms as our flashlight routines.
Note that we previously only considered problems where the depth of the branching and the size of the entire computation tree was bounded by a function of k, as otherwise we could face long stretches of finding no solutions, which then could be exponential in n. Using a flashlight routine, even if the depth or branching factor of our tree is bounded only by $p(n) \cdot f(k)$ for some polynomial p, we can go through the tree depth first and thereby ensure FPT-delay.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

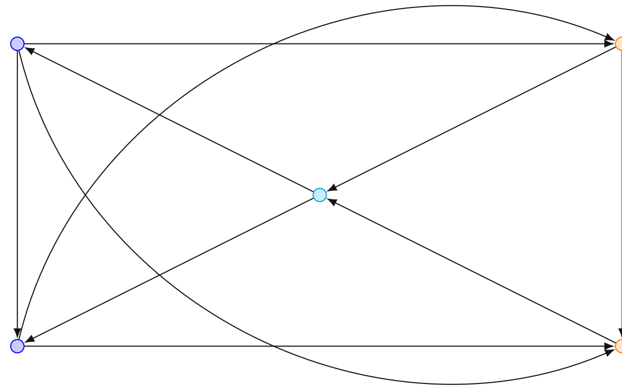

Figure 7.3: Tournament with only a single Feedback Vertex Set of size 1, with $n_1 = n_2 = 2$. u is given in cyan, the vertices in V_2 are given in blue and vertices in V_1 in orange.

7.2.1 Vertex Cover

To construct a flashlight algorithm for Vertex Cover, we will utilize the structure of the Branch and Bound Enumeration algorithm, but at each step use a slight variation of the original Branch and Bound FPT decision algorithm A to see if a branch will contain solutions: We are given a Graph $G = (V, E)$ where all vertices $v \in V$ are either coloured red or blue. If G contains an edge uv, if both u and v are red, then that branch contains no solution. If exactly one of u and v is blue, add it to our solution Set, delete it from G and reduce k by 1. If both are blue, branch as in the original algorithm. Using A we can check whether any instance contains a solution in time $2^k \cdot n^{O(1)}$.

We can now proceed as follows: For our graph G, let uv be an arbitrary edge in E. We now branch over choosing u and not v, v and not u or both, as before, and if G contains no edges, enumerate all possibilities of adding the remaining vertices to the solution up to size k (for this algorithm, we only consider the problem of finding all vertex covers, not inclusion-wise minimal ones). However, before recursively applying our algorithm, we run A on the resulting instance to check if this branch contains a solution. We only add it to the instance stack if it does. By doing this, we explore our computation tree depth first, only in branches that contain solutions, and as we have a depth of k, we can guarantee that between any 2 solutions, we apply A at most k times, so we have a delay of $O(k \cdot 2^k \cdot n^{O(1)})$, which is less than the slightly worse guarantee of $O(3^k \cdot n^{O(1)})$ we had before. Of course, the overall runtime of this algorithm is possibly worse (e.g. if our instance has only a constant number of solutions then the total computation time would be lower, but if $O(2^k)$ many solutions exist the total runtime could reach $O(4^k n^{O(1)})$), as it copies its structure but uses a lot of somewhat superfluous computations. As such, all arguments for correctness are identical to our previous case. See Algorithm 9 for a simple pseudo-code describing this algorithm.

Note that we could also have used e.g. a kernelization algorithm as our flashlight algorithm, and depending on the algorithm we get different guarantees on the delay. In all our cases the delay will remain FPT, as the entire computation tree has a size depending only on k. Also, we only use this method to generate all solutions, not minimal solutions,

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

because our flashlight algorithm cannot ensure that a minimal solution will be contained in any given branch. Of course, as the total running time of the branching is FPT, we can check minimality at the end of each branch, but this just results from the overarching structure of the branching.

Algorithm 9 VC-using-Flashlight(G, k, Sol, F, c)

```
if G − Sol does not contain any edges and c is odd then
    return Sol
end if
Let uv be an arbitrary edge in E
if u ∉ F and FlashlightVC(G, k, Sol ∪ {u}, F ∪ {v}, c + 1) = 'yes' then
    VC-using-Flashlight(G, k, Sol ∪ {u}, F ∪ {v})
end if
if v ∉ F and FlashlightVC(G, k, Sol ∪ {v}, F ∪ {u}, c + 1) = 'yes' then
    VC-using-Flashlight(G, k, Sol ∪ {v}, F ∪ {u})
end if
if u, v ∉ F and FlashlightVC(G, k, Sol ∪ {u, v}, F, c + 1) = 'yes' then
    VC-using-Flashlight(G, k, Sol ∪ {u, v}, F)
end if
if G − Sol does not contain any edges and c is even then
    return Sol
end if
```

Using the same method (including the colouring), we can get a $O(k \cdot 3^k \cdot n^{O(1)})$ delay algorithm for Feedback Vertex Set in Tournaments, again with a potentially worse overall runtime, see Algorithm 10 for a description of this algorithm. Note that, as with Algorithm 9, we do not check for size of the solution, as we include this check in the Flashlight algorithm.

7.2.2 Closest String

Lastly we now want to consider the Closest String problem: given a set of strings, is there a string with distance at most k to all other strings? Unlike most other problems we consider in this paper, this problem is not (primarily) a graph problem, and as such we cannot exploit natural structural properties as we do for graph problems. We mainly rely on an existing algorithm for the problem presented in [20], and otherwise only use a natural branching structure to explore all possibilities.

We are given a Set $S = \{x_1, \cdots, x_n\}$ of n strings of length L over an alphabet Σ and an integer k. We want to find all strings of length L that have a Hamming distance $\leq k$ to every string in S. We additionally assume that we are only interested in strings s such that, for each $1 \leq i \leq L$, there is sone $x \in S$ with $s[i] = x[i]$.
Basic Idea: Fix increasingly long prefixes of strings (filling the rest with w.l.o.G. the letters from x_1) and use a flashlight algorithm to ensure there exists a solution with that prefix. We follow the Reduction Rules of the FPT algorithm: We can think of our strings as being contained in a $n \times L$ matrix. We call a column of this matrix bad if it contains at least two different symbols from Σ, and good otherwise.
Rule 1: Delete all good columns.
By our assumption, the strings we look for will already be determined in all good columns,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 10 FVST-using-Flashlight($Stack$)

```
while Stack is not empty do
    Remove the top instance (G, k, Sol, F) from Stack
    if G contains no triangles then
        return Sol
    end if
    Let {u, v, w} be an arbitrary triangle in G
    if FlashlightFVST(G, k, Sol ∪ {u}, F ∪ {v, w}) = 'yes' then
        Add (G, k, Sol ∪ {u}, F ∪ {v, w}) to Stack
    end if
    if FlashlightFVST(G, k, Sol ∪ {v}, F ∪ {u, w}) = 'yes' then
        Add (G, k, Sol ∪ {v}, F ∪ {u, w}) to Stack
    end if
    if FlashlightFVST(G, k, Sol ∪ {w}, F ∪ {u, v}) = 'yes' then
        Add (G, k, Sol ∪ {w}, F ∪ {u, v}) to Stack
    end if
    if FlashlightFVST(G, k, Sol ∪ {u, v}, F ∪ {w}) = 'yes' then
        Add (G, k, Sol ∪ {u, v}, F ∪ {w}) to Stack
    end if
    if FlashlightFVST(G, k, Sol ∪ {u, w}, F ∪ {v}) = 'yes' then
        Add (G, k, Sol ∪ {u, w}, F ∪ {v}) to Stack
    end if
    if FlashlightFVST(G, k, Sol ∪ {v, w}, F ∪ {u}) = 'yes' then
        Add (G, k, Sol ∪ {v, w}, F ∪ {u}) to Stack
    end if
    if FlashlightFVST(G, k, Sol ∪ {u, v, w}, F) = 'yes' then
        Add (G, k, Sol ∪ {u, v, w}, F) to Stack
    end if
end while
```

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

and they wont add to the Hamming distance.
Rule 2: If there are more than nk bad columns, then it is a no-instance.

For the Flashlight algorithm, we fill the missing characters from our prefix string with characters from e.g. x_1 (we will refer to the resulting string as t), then follow the FPT algorithm closely: we look for a string x with distance $\geq k+1$ to our proposed solution, and choose $k+1$ columns where x and t differ. If all of them are contained in the prefix columns J, it is a no-instance. otherwise, choose a column $i \notin J$ and change $t[i] := x[i]$. We branch over all such choices, adding i to J and then continue. Because each change will increase the distance from t to x_1, the depth of this branching is bounded by k, and we have a branching factor of at most $k+1$, so we can bound the runtime of this algorithm by $O(nk \cdot (k+1)^k)$.

Using this algorithm, we explore the space of all possible prefixes depth first, only traveling along paths that contain a solution. Therefore we can find all solutions with a delay of $O(n^2 k \cdot nk \cdot (k+1)^k)$.

Note that, if we want to find all strings such that the maximal distance to all other distance is minimal, we can run the original FPT algorithm $\log_2 k$ times to find the smallest $k' \leq k$ that is still solvable, and then run the enumeration algorithm for this k'.

TU Bibliothek, WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Chapter 8

Instance Search

We will now consider the method of Instance Search. It relies mainly on using existing FPT algorithms to find a solution S and, based on S, construct a set of instances that contain all solutions but S. The basic structure is thus similar to Binary Partition (and particularly Flashlight Algorithms), but using a technique that allows for finding a solution at each decision step ensures, among other things, that the depth of the branching does not need to lie within almost any bounds.

The name of this method is in reference to Reverse Search, that utilizes a similar algorithmic structure for finding solutions but does not rely on a fixed parent-child relation between solutions.

The basic structure of Instance search can be described as follows: Let (I, k) be an instance. For an Instance Search algorithm we first need to find a solution S (or conclude correctly that none exist), and construct a set of instances $I_1, \cdots I_{g(k)n^{O(1)}}$. We require that the set of new instances is bounded as we will traverse the computation tree depth first, and having to check too many child instances could prevent an FPT-delay algorithm.

We require that $Sol(I) = \{S\} \cup \left(\bigcup_j Sol(I_j)\right)$ to ensure that no solution is lost, and $Sol(I_i) \cap Sol(I_j) = \emptyset \Leftrightarrow i \neq j$ so nothing is output more than once.

We will combine these steps together with an instance stack $Stack$. We can describe the algorithm abstractly as follows:

InstanceSearch($Stack$)

Remove the top instance I from $Stack$
Let $S = Flashlight(I, k)$
Output S
Let $I_1, \cdots, I_l$ be the child instances of I
for $1 \leq j \leq l$ **do**
    if $Flashlight(I_j, k) =$ 'yes' **then**
        Add I_j to $Stack$
    end if
end for

We note that we primarily utilize the Stack to ensure that the delay between any two solutions is small, even though it does not generally reduce the total computation time

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

(or ensure that solutions are found faster). This could alternatively be ensured by holding back some solutions.

8.1 Examples

We first consider the problem of Vertex Cover to introduce the structure of these algorithms and to provide an introductory example.

8.1.1 Vertex Cover

Let $G = (V, E)$ be a graph and (G, k) be an instance of VC. We will generalize the problem of VC to allow us to control the kinds of solutions that may exist: We given a graph G, a parameter k and two sets C and F with $C \cap F = \emptyset$. Then (G, k, C, F) is a yes-instance iff there exists a set S such that $S \cap F = \emptyset$, $|S \cup C| \leq k$ and $S \cup C$ is a vertex cover of G. We will call this problem Vertex Cover-DISJOINT (VC-DISJOINT), and it can be solved using a slight variation on e.g. a Branch and Bound algorithm for VC. In particular, we can delete C from G, set $k' := k - |C|$ and only branch along choices that do not choose vertices in F. We will call this algorithm A_{VCD}, and by the same arguments as the original Branch and Bound algorithm for VC we achieve a worst-case runtime of $O(2^{k'} n^{O(1)})$. We give a more detailed description of this algorithm below. Note that, while the current algorithm is written as a decision algorithm, it can be easily altered to provide a solution, if it exists.

VC-DISJOINT(G, k, C, F)

---

if $C \cap F \neq \emptyset$ or $|C| > k$ **then**
    return 'no'
end if
if $|C| = k$ or $E = \emptyset$ **then**
    if $G - C$ contains no edges **then**
        return 'yes'
    else
        return 'no'
    end if
end if
Let uv be an arbitrary edge in E.
$s_1 \leftarrow$ VC-DISJOINT$(G, k, C \cup \{u\}, F)$
$s_2 \leftarrow$ VC-DISJOINT$(G, k, C \cup \{v\}, F)$
if $s_1 = s_2 =$'no' **then**
    return 'no'
else
    return 'yes'
end if

We can now transform our original instance (G, k) into $(G, k, \emptyset, \emptyset)$ and proceed as follows: Given an instance (G, k, C, F), we find a solution S by applying A_{VCD} and return it. We now branch over all the ways another Solution could differ from S:
For all $T \subsetneq S$, construct the instance $(G, k, C \cup T, F \cup (S \backslash T))$, and then run A_{VCD} to

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

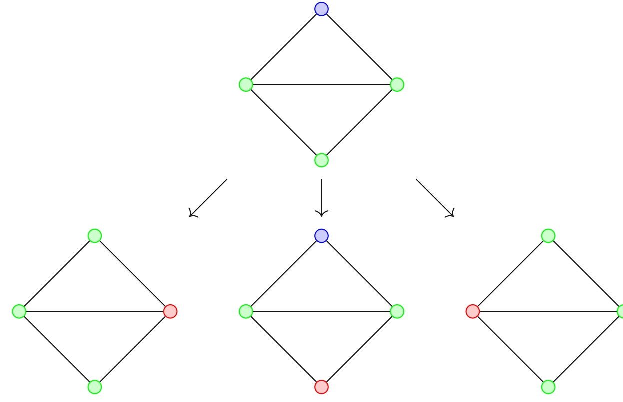

Figure 8.1: A full computational tree of the branching for $k = 3$ given the inital solution found above and only depicting nodes containing solutions, vertices in F coloured red, solutions coloured green. Note that the solution of choosing the top 3 vertices is not considered in this branching and would have to be constructed by expanding the middle solution.

check whether a solution exists. Add instances which do contain solutions to a stack or queue R, then repeat this process for the next instance in R.
This algorithm will not necessarily output all solutions (see 8.1), but it will return all inclusion-wise minimal ones (and more), as we consider all possible overlaps and A_{VCD} only branches while there are edges remaining. Additionally, because any two branches of our computation tree have vertices in F that the other has in C, they can never output the same solutions. Now let L_1, L_2 be nodes of our computation tree such that L_2 is below L_1, and let S be the solution output for L_1 and (G, k, C, F) be the instance at L_2. By construction, $S \cap F \neq \emptyset$, and thus A_{VCD} at L_2 will never output S. Thus no solution is output twice.
Lastly, to output all possible solutions, for any solution S output by the algorithm we need to add all possible combinations of vertices in $V \backslash S \cup F$ to S up to the budget of k. We exclude F to ensure that in this step no solution will be returned twice.
The Delay of this procedure is dominated by running A_{VCD} up to twice per node, which obviously becomes less intensive the bigger C is, but as we start with $C = \emptyset$ we cannot guarantee a delay below $O(2^k n^{O(1)})$.

8.1.2 Feedback Vertex Set in Tournaments

We again consider the problem of finding all Feedback Vertex Set in a Tournament. Using a similar approach as we did in the Vertex Cover problem, we can likewise branch over possible intersections and then utilize a flashlight algorithm to generate new solutions

Let $G = (V, E)$ be a Tournament and (G, k) be an instance of FVST. Using the same Flashlight algorithm we did before on $(G, k, \emptyset, \emptyset)$, we can find a solution S for (G, k).

So let our instance now be (G, k, C, F) and S be a solution, i.e. $S \cup C$ is a Feedback Vertex Set and $|S \cup C| \leq k$. We can then branch over all $< 2^{k-|C|}$ ways of partitioning S into F' and C', $F' \neq \emptyset$. For each such partition (C', F') run $FlashlightFVST(G, k, C \cup C', F \cup F')$ and add instances which contain solutions to a stack or queue.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

As we need to consider all possible child nodes in the computation tree, we need to use the flashlight algorithm up to $O(2^k)$ many times before we can find the next solution, and thus cannot guarantee a delay below $O(2^k \cdot 3^k \cdot n^{O(1)})$.

By using a variation on the original FPT Kernelization algorithm using colours, we can also construct a Reverse Search algorithm for Feedback Arc Set in Tournaments. As the main work in the FPT algorithm is the kernelization process, this would not decrease the guaranteed delay and would, likely, just increase the overall runtime. It would be more efficient to just use the Kernel and use this for branching, but the Enum-Kernelization already does this.

8.1.3 Integer Linear Programming

We will now consider the Problem of Interger Linear Programming, which is an important problem in Logistics and Operations Research as many practical problems can be stated in the form of linear inequalities. This is, by itself, not particularly surprising as the problem is NP-complete and thus we can state all NP problems in the form of an integer linear programming question, but integer programming often provides a natural and intuitive way of describing optimization problems.

We are given a matrix $B \in \mathbb{Z}^{m \times p}$ and a vector $b \in \mathbb{Z}^p$. Our instance is a yes-instance if there exists a vector $x = (x_1, \cdots, x_p) \in \mathbb{Z}^p$ s.t. $Bx \leq b$. INTEGER LINEAR PROGRAMMING (ILP) is FPT parameterized by the number of variables p [22][23]. We will assume that, for problems we consider, the solution space is finite.

Using this algorithm A as a blackbox, we construct a flashlight algorithm to solve the associated enumeration problem with FPT delay. Let (B, b) be our instance of ILP. We first use A to generate a solution $x' = (x'_1, \cdots, x'_p)$ or conclude that no solution exists. We then construct and branch over several sub-instances: We add equations of the form $x_j = x'_j, x_i \; R \; x'_i$ for $R \in \{<, >\}$ and $\forall j < i$ (so the new solution is identical in the first $i-1$ variables, but differs on the i-th variable). At each new instance we use A to either generate a new solution or conclude that the current branch contains no solutions. If we generate a new solution x'', we branch in a similar way, at each step keeping only the most restrictive constraints, i.e. equality constraints and the lowest upper and highest lower bound. Thus at each step of the branch we have at most $2p$ additional constraints, so the runtime remains FPT w.r.t. the original input.

At each step our algorithm has a branching factor of $\leq 2p$, but as the depth of this solution tree is unbounded, we need to hold back some of our solutions to avoid entering a large number of branches which do not contain solutions. So at each node we proceed as follows:

- run A on the current instance, generating a solution x'
- for each child, run A on the restrained instance. Add each child that contains a solution to a queue/stack of to be explored instances
- Output x', then recursively apply these steps to instances in the queue/stack

Since the branching factor is bounded by $2p$, we need to run A at most $2p + 1$ times before the next solution is generated. By construction, any 2 branches will have mutually

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

excluding constraints which partition our solution space, every solution is output exactly once. Thus all solution of ILP can be generated with FPT delay.

Note that in the algorithm as described above, solutions are generated twice, once when considering an instance on its own and once when considering it as a child of another instance. This would not notably decrease our worst case delay but a sensible implementation would avoid this.

As we have no inherent restrictions on the size of our solution space, both depth-first as well as breadth-first search approaches to this problem could require exponential space. If every variable has at most $q(|I|)$ many possible values for some polynomial q, then the depth of the branching is likewise bounded by $q(|I|)$, thus a depth-first approach would require only polynomial space.

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Chapter 9

Miscellaneous

Some of the problems we consider in this paper have solution methods that, while still general, are very specific as to the structure of the problem - in our case, graph problems that allow specific approaches due to their structure.

9.1 Colour Coding

Colour Coding [1] is a technique we will only consider in the context of randomised algorithms in [9]: We will first colour a graph randomly and then only look for solutions that satisfy certain properties (in our case, every vertex of a solution having a different colour or all having the same colour and differing from all its neighbours). Of course such an approach will not necessarily return any solution as we work probabilistically. It is in fact quite unlikely, for any specific solution, that it will be coloured correctly in a given colouring. However, it turns out that, for the problems we consider in this paper, there exists a well suited methods for derandomising such an algorithm: We can create a set of colourings F such that any possible solution will be coloured correctly by some colouring in F.

At first this can still be a problem, as any solution might be found under several different colourings in F. We will present an inductive argument why we can still find a way to return each solution once with a small maximal delay (in our case, FPT). This is based on [29].

9.1.1 Longest Path

We are given a graph $G = (V, E)$ and an integer k. We want to find all paths of length $k-1$ ('k-path') in G. To do this, we follow the derandomized colour coding approach of [9]: We can find a family of colourings $\mathcal{F}$ from V to $\{1, \cdots k\}$ such that, for any $v_1, \cdots, v_k \in V$ there exists a colouring $f \in \mathcal{F}$ with $f(v_i) \neq f(v_j)$ for $i \neq j$. There exists such a family of size $e^k k^{O(logk)} \log n$ that can be found in time $e^k k^{O(logk)} n \log n$, i.e. FPT time [26]. Therefore, for any k-path P in G, there exists a colouring in $\mathcal{F}$ that colours every vertex in P with a different colour. For a fixed colouring f, we can find all such paths as follows:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Following the original FPT algorithm, we inductively construct $PATH(S, u)$ for $u \in V$ and $S \subset [k]$, which is true if and only if there exists a path of length $|S| - 1$, that ends in u and uses every colour in S. For $|S| = 1$, this is true iff $S = \{f(u)\}$, else

$$PATH(S, u) = \begin{cases} \bigvee PATH(S \backslash \{f(u)\}, v), & \text{for } vu \in E \\ false, & \text{else} \end{cases}$$

To be able to construct every path, not just confirm the existence, we also need to save all such predecessors $v \in V$ to the tuple (S, u). We can then take any vertex $x \in V$ and look at all predecessors in $PATH(\{1, \cdots, k\}, u)$, then continue depth-first in $PATH(\{1, \cdots, k\} \backslash \{f(u)\}, v)$. By construction, we only look into along branches that will yield a valid path.
Depending on whether or not we want mirrored paths to count as the same path, we can impose an arbitrary ordering $<$ on the colours and only output those paths $(v_1, \cdots, v_k)$ that fulfill the condition $f(v_1) < f(v_k)$.

If we simply went through all such colourings and listed colourful paths within each colouring, we would potentially list the same path several times. To avoid this, we will proceed as follows: we define the use of this algorithm on the colouring f as A_f. We can tell whether A_f will output a k-path P in linear time by checking if P is colourful within f. Let $\{f_1, \cdots, f_m\}$ be our set of colourings.
If $m = 2$, we run A_{f_1}, and for every solution P we find we check if A_{f_2} would also find this solution. If not, we simply output P and continue running A_{f_1}.
If A_{f_2} would find this solution, we instead run A_{f_2} until we find a solution P'. We output P', and then continue running A_{f_1} and repeat this process. If A_{f_1} has found all its solutions and A_{f_2} is not yet finished, we continue running A_{f_2} until all solutions are found.
Since every solution in A_{f_2} is given only after either A_{f_1} has found a solution or has finished running, we will never encounter a case where A_{f_1} tries to call A_{f_2} although it has already finished.
We will now argue correctness of this algorithm: By assumption, every solution is found either by A_{f_1} or A_{f_2}. Since A_{f_2} outputs all its solutions, and A_{f_1} exactly those that are not given by A_{f_2}, every solution is output exactly once. If f_i has at most FPT runtime and A_{f_1} and A_{f_2} have FPT delay, this combined algorithm will have delay at most runtime of checking in f_2 + delay of A_{f_1} + delay of A_{f_2}, so FPT delay.
We will now generalize this construction inductively: Assume we can find an algorithm that, for algorithms $\{A_2, \cdots, A_n\}$ with delays $\{d_2, \cdots, d_n\}$ and check algorithms $\{B_3, \cdots, B_n\}$ with runtimes $\{b_3, \cdots, b_n\}$, can construct all solutions exactly once with delay $O(\sum d_i + (n-1)\sum b_i)$. We will call this algoritm A_2'. Then we can construct our algorithm for $\{A_1, \cdots A_n\}$, where A_1 has delay d_1 and B_2 has runtime b_2, by following the construction for 2 algorithms on $\{A_1, A_2'\}$ with delay $O(d_2 + \sum b_i + \sum d_i + (n-1)\sum b_i) = O(\sum d_i + n \cdot \sum b_i)$. If $n, b_i, d_i \leq f(k) \cdot n^{O(1)}$, then we have FPT delay. Note that this construction overcounts the b_i.

We will next consider the Problem of SUBGRAPH ISOMORPHISM, of trying to find a given structure within a graph.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, WIEN Your knowledge hub

9.1.2 Subgraph Isomorphism

In SUBGRAPH ISOMORPHISM, parameterized by k, we are given graphs G, H, where $|H| \leq k$. (G, H) is a yes-instance if H is isomorphic to a subgraph of G. As H can be any graph, including a clique, SUBGRAPH ISOMORPHISM is at least as hard as CLIQUE. As CLIQUE is W[1]-hard and thus unlikely to be FPT, SUBGRAPH ISOMORPHISM is also unlikely to be FPT, and likewise an FPT delay enumeration algorithm probably does not exist. Therefore we include another parameter d, the maximal degree of any vertex in G, and seek to generate an enumeration algorithm with a delay of $f(d,k)n^{O(1)}$. To use a similar colour coding approach to LONGEST PATH, we first need to explore a new way of colouring our graph.

Definition 30. *An (n,k)-universal set is a family $\mathcal{U}$ of subsets of $\{1, \cdots, n\}$ s.t. for all $S \subseteq \{1, \cdots, n\}$ with $|S| \leq k$, the family $\{A \cap S : A \in \mathcal{U}\}$ contains all subsets $S' \subseteq S$.*

We will consider a set $A \subseteq \{1, \cdots, n\}$ as colouring f with

$$f(v) = \begin{cases} 1, & v \in A \\ 2, & else \end{cases}$$

Theorem 15 ([26])**.** *For any $n, k \geq 1$ one can construct an (n,k)-universal set of size $2^k k^{O(\log k)} \log n$ in time $2^k k^{O(\log k)} n \log n$.*

Using this theorem, we can construct an $(n, k(d+1))$-universal set $\mathcal{U}$ in FPT time. Let $\hat{H}$ be a subgraph of G isomorphic to H, then there exists a colouring $f \in \mathcal{U}$ s.t. $f(v) = 1$ for $v \in \hat{H}$ and $f(v) = 2$ for $v \in N(\hat{H}) \backslash \hat{H}$. For all connected components H_i of H, we can check whether there exists an isomorphic subgraph $\hat{H}_i$ in G which is coloured 1 and having all its neighbours be coloured 2, by checking all monocoloured components of size $|H_i|$. Once all isomorphic components for all H_i have been found, we can create a bipartite graph $\tilde{G} = (S \cup T, \tilde{E})$ where S contains a vertex for every component of H and T contains a vertex for every component coloured 1 in G under f, and $s \in S$ is connected to $t \in T$ iff they represent isomorphic graph components (we can find these in time $O(k^2 n^2)$ using brute force). In this way, an isomorphic $\hat{H}$ corresponds to a maximal matching in $\tilde{G}$ of size $|S|$. By enumerating these matchings (which we can find in polynomial time) we can find all isomorphic $\hat{H}$ under any given colouring, and using the same overall approach as in LONGEST PATH, we can construct an FPT delay algorithm solving SUBGRAPH ISOMORPHISM.

9.2 Iterative Compression

We will quickly restate the basic formulation of iterative compression and then mention how to generalize it:

For a problem $(*)$ we want to solve, we instead solve $(*)$-COMPRESSION: $(*)$-COMPRESSION is the task of taking a solution of size $k+1$ of $(*)$ and either finding a solution of size at most k or concluding that no such solution exist. Usually the start of the algorithm is to take a small subgraph G' of our Input, usually of size k, where the solution to the problem is trivial. Then we gradually increase the size of the graph, one vertex/edge at a time, which allows us to easily transform a solution X of size k into a solution of size

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

$k+1$, upon which we then run the compression algorithm. We repeat these steps until we either find a solution for the entire graph or can conclude that none exist.

Instead of solving the ($*$)-COMPRESSION directly, we construct a subproblem ($*$)-DISJOINT: given a solution Z on a graph G of size $k+1$, find a disjoint solution of size k. More precisely, given a solution S of size $k+1$ and a partition $S = C \dot\cup F$, we want to find a solution S' of size k with $S \cap S' = C$.

So we want to be able to do the following 2 steps repeatedly:

1. Transform a solution of size k on G into a solution of size $k+1$ on $G \cup \{v\}$ (or $G \cup \{e\}$), and
2. given a solution S of size $k+1$, iterate over all ways of spliiting S into F and C (so at most 2^{k+1} many possibilities) and then solve ($*$)-DISJOINT.

If we want to transform such an algorithm into an enumeration algorithm, we only need to solve the corresponding enumeration problem ENUM-($*$)-DISJOINT (i.e. given a solution S of size $k+1$ and a partition $S = C \dot\cup F$ find all solutions S' of size k having $S \cap S' = C$). We can simply run the original iterative compression algorithm until the last step, where we first consider the full graph G and a solution S, and for every possible $C \subset S$ run an ENUM-($*$)-DISJOINT algorithm. If the original algorithm has a runtime of d_1 and the disjoint enumeration algorithm has a delay of $\leq d_2$, then all solutions to ($*$) can be enumerated with a delay of $d_1 + d_2$ (or more precisely, a preprocessing time of $d_1 + d_2$ and afterwards a delay of d_2), so if both are FPT this gives us an FPT-delay enumeration algorithm.

Note that not all problems using this method of guessing an intersection with a given solution necessarily need a compression routine. In [9], a 2-approximation algorithm is used to either rule out a solution of size k for VERTEX COVER, or finding a solution of size $\leq 2k$, and using a VERTEX COVER-DISJOINT algorithm.

As an example, we will again consider the Vertex Cover Problem:

9.2.1 Vertex Cover

Let $G = (V, E)$ and (G, k) be our instance of Vertex Cover. Let $(v_1, \cdots, v_n)$ be an arbitrary ordering on V, and let $G_j = (V_j, E_j)$ be the induced subgraph $G[v_1, \cdots, v_j]$ of G. Then V_k is a solution to Vertex Cover of size k on G_k.

For any Solution S of size k on G_j, $S \cup \{v_{j+1}\} =: S'$ is a solution of size $k+1$ on G_{j+1}, as any edges involving V_{j+1} are automatically covered. For the decision variant of VERTEX COVER-DISJOINT, we can now simply check whether $C \cup N(F)$ is a Vertex Cover of size $\leq k$ for all possible choices of C, as every edge is adjacent to a vertex in $C \cup F$, so to cover all edges without choosing a vertex in F we need to include all its neighbours. In that case we can repeat this process with $C \cup N(F) \cup \{v_{j+2}\}$ on G_{j+2}.

Now let S be a solution of size $k+1$ on G, and $C \subset S$. We want to find all Vertex Covers S' having $S \cap S' = C$ and $|S'| \leq k$. As with the decision variant of this problem, we know that if there exists such a vertex cover, then $C \cup N(F)$ is a vertex cover of size $\leq k$. More specifically, all such vertex covers must include $C \cup N(F)$ by the construction of the problem. Thus, if we want to find all inclusion-wise minimal vertex covers, we iterate

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

over all $C \subset S$ and check whether $C \cup N(F)$ is a minimal vertex cover of size $\leq k$. If we want all vertex covers we can take additional vertices, as with previous algorithms.

Note that this algorithm does not necessarily construct a solution of size $k+1$ in each step: If $C \cup N(F) < k$ then we can skip the construction of a new solution, as $C \cup N(F) \cup \{v_{j+1}\}$ has already size $\leq k$. Also, as in each of the $< n$ steps of the Compression Routine we need at most $2^{k+1} n^{O(1)}$ many steps, and the final step has a delay of $\leq 2^{k+1} n^{O(1)}$ we achieve an overall delay of $O(2^{k+1} n^{O(1)})$.

As noted before in this section, we can also find a Solution S to Vertex Cover using a polynomial time 2-approximation algorithm (which consist of simply choosing both vertices for every edge we consider until none are left) and use S for guessing an intersection, but doing this we have to check up to $O(4^k)$ many possible intersections between a solution of size k and S, which has up to $2k$ vertices. Thus we cannot guarantee a delay below $O(4^k n^{O(1)})$ using this method.

9.3 Dynamic Programming

In Dynamic Programming, we usually solve a problem by setting up a data structure that allows us to explore the space of all possibilities by building solutions step by step: In optimization problems, this is often done by storing the optimal solution $g(m)$ of size m or including only the first m elements, and calculating $g(l)$ elements using the optima $g(l-1), g(l-2), \cdots, g(1)$.

Dynamic Programming is often applied to problems that can be constructed piecewise (e.g. a solution of size l can be constructed from solutions of size $l-1$ and an additional element or from solutions of size $l-m$ and m).

For our purposes, we can use this data structure by marking how these values were constructed. For instance, if we calculate $g(l) = \max_{m \leq l} h(l-m, m)$ (where h describes how to combine solutions of size $l-m$ and m to a solution of size l using $g(l-m)$ and $g(m)$), we can mark for which m this maximum is achieved. We can then reconstruct all solutions of size l by branching over all such m, and output solutions for $g(l-m)$ and $g(m)$. We will apply this basic structure to the FPT algorithm for the Steiner Tree problem in [9].

9.3.1 Steiner Tree

We are given an undirected Graph $G = (V, E)$, a set of terminals $K \subset V$ and a weight function $w : E \to \mathbb{R}_{>0}$. In STEINER TREE we are trying to find the weight of a minimal tree T in G that contains all terminals in K, parameterized by $|K| = k$. We call such trees Steiner trees of K.
We will assume $|K| > 1$, as the task is trivial otherwise, and that G is connected, as a Steiner tree can only exist if for any $v, w \in K$ there is a path from v to w in G. Additionally, we will assume that ever $v \in K$ has only one neighbour w in G, where $w \notin K$. Otherwise we add a vertex v' to G, connect it only to v with $w(v, v') = 0$ and continue on $K' = K \backslash \{v\} \cup \{v'\}$.

We will closely follow the FPT algorithm in [9] based on [12]: For all nonempty subsets

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

$D \subseteq K$ and vertices $v \in V \backslash K$, let $T[D, v]$ be the weight of a minimal Steiner tree of $D \cup \{v\}$. If $|D| = |\{t\}| = 1$, the minimal weight of a Steiner tree of $D \cup \{v\}$ is the distance from v to t.
For $|D| \geq 2$, we use the following Lemma:

Lemma 4 ([9])**.** *For every $D \subseteq K$ with $|D| \geq 2$ and $v \in V \backslash K$, it holds that*

$$T[D, v] = \min_{\substack{u \in V \backslash K \\ \emptyset \neq D' \subset D}} \{T[D', u] + T[D \backslash D', u] + dist(u, v)\}.$$

Proof. We first want to show

$$T[D, v] \leq \min_{\substack{u \in V \backslash K \\ \varnothing \neq D' \subset D}} \{T[D', u] + T[D \backslash D', u] + dist(u, v)\},$$

so let $u \in V$ and $\emptyset \neq D' \subsetneq D$. Let H_1 and H_2 be the Steiner Trees for $D' \cup \{u\}$ and $D \backslash D' \cup \{u\}$ respectively, which represent the values of $T[D', u]$ and $T[D \backslash D', u]$, and let P be the shortest path from u to v. Then $H_1 \cup H_2 \cup P$ is a connected subgraph of weight

$$\begin{aligned} T[D, v] &\leq w(H_1 \cup H_2 \cup P) \\ &\leq w(H_1) + w(H_2) + w(P) = T[D', u] + T[D \backslash D', u] + \mathrm{dist}(u, v). \end{aligned}$$

Therefore the inequality holds.

For the other inequality let H be a Steiner tree for $D \cup \{v\}$ of minimal weight. We will consider this tree rooted at v, then there exists some vertex u_0 in H that has at least 2 children and is closest to v among all such vertices (this must exist as $|D| \geq 2$ and that every terminal vertex has degree 1). This also implies that $u_0 \notin K$.

Now let u_1 be a child of u_0 in H, then we can partition the tree H into

1. the Path P from u_0 to v in H,
2. the subtree H_1 rooted at u_1 together with the edge $u_0 u_1$, and
3. the subtree H_2 consisting of the subtree rooted at u_0 excluding H_1.

Now let $D' = V(H_1) \cap K$ be the terminals in H_1, then $D \backslash D' = V(H_1) \cap K$. As H is a Steiner tree, so of minimal weight, $\emptyset \neq D' \neq D$ as either $H \backslash H_1$ or $H \backslash H_2$ would be a tree containing all terminals of smaller weight in contradiction to its minimality. Because of the minimality we can also conclude that $T[D', u_0] = w(H_1)$ and $T[D \backslash D', u_0] = w(H_w)$ and that P is the shortest path from u_0 to v, therefore we get

$$\begin{aligned} T[D, v] = w(H) &= T[D', u_0] + T[D \backslash D', u_0] + dist(u, v_0) \\ &\geq \min_{\substack{u \in V \backslash K \\ \emptyset \neq D' \subset D}} \{T[D', u] + T[D \backslash D', u] + dist(u, v)\}. \end{aligned}$$

□

Using this construction, we can calculate all $T[D, v]$ and then $\min_{v \in V \backslash K} \{T[K, v]\}$ to find the minimal overall weight. To use this construction to generate all minimal solutions, we need to store, for each $T[D, v]$ with $|D| \geq 2$, which choices of D' and u led to the minimal overall weight. Generating the tree in the base case $|D| = 1$ comes down to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

simply finding the shortest path, and otherwise, assuming we can recursively find a tree T_1 corresponding to $T[D', u]$ and T_2 to $T[D \setminus D', u]$, we simply connect T_1 and T_2 via u to v using the shortest path from u to v in G.
To ensure that no solution is output twice, we impose an arbitrary ordering $<$ on $V \setminus K$, and only construct trees through vertices u with $u < v$. As there will be a vertex v in every Steiner Tree that is maximal in this ordering, it will be output when $T[D, v]$ is used to construct a tree.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 10

Conclusion

In this thesis we have systematically explored ways to transform FPT algorithms for decision problems to FPT-delay algorithms for enumeration problems. It has turned out that many FPT algorithms naturally explore the space of all candidate solutions and can thus easily be adapted into enumeration algorithms. Then there have been problems where the FPT decision algorithms required several modifications and extensions in order to be turned into enumeration algorithms with FPT-delay. A few problems even required an individual approach to solve by utilizing methods fitting for their fundamental structure. However, we have also encountered several FPT decision problems in [9] where the design of an FPT-delay algorithm for the corresponding enumeration problem has been left open. These was in particular the case with algorithms that work by simplifying the problem instances in a way that does not allow for all solutions to be reconstructed. For instance, the algorithms for Egde Clique Cover or Feedback Vertex Set (among others) do not allow a clean translation into enumeration algorithms.

We have applied the method of Enum-Kernelization from [8] to a variety of problems. In the cases we considered, we first applied existing kernelization algorithms, and by utilizing a brute force method on the kernel and reconstructing the steps used in the original instance we managed to get FPT-delay algorithms for several parameterized problems.

For Binary Partition, we utilized the method of Branch and Bound for FPT-decision algorithms to naturally extend them to FPT-delay enumeration algorithms (e.g. Feedback Vertex Set in Tournaments). Even for cases where such a structure did not exist or was not usable, we have shown different approaches to turn existing FPT decision algorithms into flashlight algorithms to generate FPT-delay enumeration algorithms on a custom computational structure.

We used a related but noticeably different computational structure in Instance Search to mimic the enumeration method of Reverse Search, by repeatedly constructing new instances based on existing solutions to find every solution quickly but without overlap. To do this, we used existing FPT algorithms to find solutions at each step, and then constructed new instances that partitioned the remaining solutions.

Finally we considered some less common techniques for constructing FPT algorithms, like Iterative Compression, which have a narrower focus (specifically graph problems of certain forms). We showed how to transform such FPT algorithms into FPT-delay

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

enumeration algorithms in a general form and applied them to problems including the Longest Path and Steiner Tree problems.

Future Work. Many structural parameters for graphs allow a great number of problems to become tractable. One particular parameter that we did not examine (as it would have exceeded the scope of this thesis) but has a wide range of applications is treewidth, which is a natural way of defining how close a given graph is to a tree. As otherwise hard problems, like Dominating Set, are FPT when parameterized under treewidth [9], the methods used for such FPT algorithms could be analyzed and applications for enumeration algorithms examined.

On the other hand, while the structure of Instance Search mirrors the structure of Reverse Search, an exploration into applying Reverse Search algorithms, or the Supergraph method in general, could lead to FPT-delay algorithms for new problems.

It would also be of interest to more closely examine problems that we did not solve, to either prove that FPT-delay algorithms are not possible or, more likely, to find an approach that allows for their construction.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Bibliography

[1] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, jul 1995.

[2] David Avis and Komei Fukuda. Reverse search for enumeration. *Discret. Appl. Math.*, 65(1-3):21–46, 1996.

[3] Liming Cai, Jianer Chen, Rodney G. Downey, and Michael R. Fellows. Advice classes of parameterized tractability. *Annals of Pure and Applied Logic*, 84(1):119–138, 1997. Asian Logic Conference.

[4] Florent Capelli and Yann Strozecki. Incremental delay enumeration: Space and time. *Discret. Appl. Math.*, 268:179–190, 2019.

[5] Florent Capelli and Yann Strozecki. Enumerating models of DNF faster: Breaking the dependency on the formula size. *Discret. Appl. Math.*, 303:203–215, 2021.

[6] Florent Capelli and Yann Strozecki. Geometric amortization of enumeration algorithms. In Petra Berenbrink, Patricia Bouyer, Anuj Dawar, and Mamadou Moustapha Kanté, editors, *40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023, March 7-9, 2023, Hamburg, Germany*, volume 254 of *LIPIcs*, pages 18:1–18:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[7] Jianer Chen, Fedor V. Fomin, Yang Liu, Songjian Lu, and Yngve Villanger. Improved algorithms for the feedback vertex set problems. In Frank Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Algorithms and Data Structures*, pages 422–433, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[8] Nadia Creignou, Arne Meier, Julian-Steffen Müller, Johannes Schmidt, and Heribert Vollmer. Paradigms for parameterized enumeration. *Theory Comput. Syst.*, 60(4):737–758, 2017.

[9] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.

[10] Peter Damaschke. Parameterized enumeration, transversals, and imperfect phylogeny reconstruction. In Rodney G. Downey, Michael R. Fellows, and Frank K. H. A. Dehne, editors, *Parameterized and Exact Computation, First International Workshop, IWPEC 2004, Bergen, Norway, September 14-17, 2004, Proceedings*, volume 3162 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2004.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[11] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness II: on completeness for W[1]. *Theor. Comput. Sci.*, 141(1&2):109–131, 1995.

[12] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1(3):195–207, 1971.

[13] P. Erdös and R. Rado. Intersection theorems for systems of sets. *Journal of the London Mathematical Society*, s1-35(1):85–90, 1960.

[14] J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[15] Jörg Flum and Martin Grohe. Describing parameterized complexity classes. *Information and Computation*, 187(2):291–319, 2003.

[16] Jörg Flum and Martin Grohe. Model-checking problems as a basis for parameterized intractability. *Proceedings - Symposium on Logic in Computer Science*, 1, 03 2005.

[17] Fedor V. Fomin, Petr A. Golovach, William Lochet, Pranabendu Misra, Saket Saurabh, and Roohani Sharma. Parameterized complexity of directed spanner problems. *Algorithmica*, 84(8):2292–2308, 2022.

[18] Fedor V. Fomin, Daniel Lokshtanov, Fahad Panolan, Saket Saurabh, and Meirav Zehavi. Hitting topological minors is FPT. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 1317–1326. ACM, 2020.

[19] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier. Data reduction and exact algorithms for clique cover. *ACM Journal of Experimental Algorithmics*, 13, 09 2008.

[20] Jens Gramm, Rolf Niedermeier, and Peter Rossmanith. Fixed-parameter algorithms for CLOSEST STRING and related problems. *Algorithmica*, 37(1):25–42, 2003.

[21] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.

[22] Hendrik W. Lenstra Jr. Integer programming with a fixed number of variables. *Math. Oper. Res.*, 8(4):538–548, 1983.

[23] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12(3):415–440, 1987.

[24] Andrea Marino. *Analysis and Enumeration Algorithms for Biological Graphs*, volume 6 of *Atlantis Studies in Computing*. Atlantis Press, 2015.

[25] Arne Meier. Incremental FPT delay. *Algorithms*, 13(5):122, 2020.

[26] Moni Naor, Leonard J. Schulman, and Aravind Srinivasan. Splitters and near-optimal derandomization. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*, pages 182–191. IEEE Computer Society, 1995.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[27] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006.

[28] Bruce Reed, Kaleigh Smith, and Adrian Vetta. Finding odd cycle transversals. *Operations Research Letters*, 32:299–301, 07 2004.

[29] Yann Strozecki. *Enumeration complexity and matroid decomposition.* PhD thesis, Université Paris Diderot - Paris 7, 2010.

[30] Yann Strozecki. Enumeration complexity. *Bull. EATCS*, 129, 2019.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.