Efektivní předzpracování a parametrizované algoritmy

MI-PAM

Učební textík

Ondřej Suchý

May 7, 2014

Contents

Chapter 1

Introduction

Classical complexity treats the problems according to whether they admit an algorithm solving them in polynomial time in terms of the input size or not. Unfortunately, many (if not most) interesting problems turned to be NP-complete, which means that a polynomial algorithm optimally solving them is fairly improbable.

This turned the attention to approximation algorithms which should be able to output a solution to the problem that is reasonably close to the optimal one in time polynomial in the input size. But for many applications such an approximate solution is not suitable. Also measuring the running times only in terms of the input size effectively ignores any structure of the instances.

Abrahamson, Ellis, Fellows, and Mata [AEFM89] were the first to propose to study the problems also with respect to some additional measure, the parameter, distinguishing whether there is an algorithm that can optimally solve all instances which have this parameter bounded by k in time $O(f(k) \cdot n^c)$, where f is some function, n is the input size and c is a constant independent of k or whether such an algorithm would require time $O(n^{f(k)})$ for some function f, that is, the exponent depends on k.

This idea was further elaborated by Downey and Fellows [DF92b, DF92a] in a series of papers in which they established the theory of fixed-parameter tractability and completeness. The series culminated in the ground-breaking textbook by Downey and Fellows [DF98], which attracted many people to the field and started a rapid development of the field.

Although since 2004 the International Symposium (formerly workshop) on Parameterized and Exact Computation (IPEC) devoted to results on parameterized complexity and exact moderately-exponential algorithms is organized, the papers involving parameterized complexity are accepted on a wide range of conferences devoted to graph problems, algorithms and complexity.

With the growing number of papers it was more and more clear, that the algorithms used to show fixed-parameter tractability often use techniques specific for the field. The 2006 Niedermeier's book [Nie06] summarizes such techniques,

while the monograph by Flum and Grohe [FG06] from the same year concentrates more to the intractability theory and maps various emerging complexity classes.

Since then the advancement in the field did not stop. Since the beginning it was noted by Downey and Fellows that parameterized complexity can be understood as a framework to measure the effectiveness of polynomial preprocessing. The method most related to this perspective, the kernelization, has now its own theory, and quite recently, in 2009, a tradition of workshops solely devoted to this method was started.

The big advantage of parameterized complexity is that a single problem can be studied from various points of view, using the virtually infinite set of possible parameters. The intractability shown for a problem with respect to a particular parameter does not mean that parameterized complexity was unsuccessful for the problem, but, instead, that more work should be done to reveal more suitable parameters for the problem which can possibly capture its hardness.

Also parameterized complexity enables us to study the complexity of a problem with respect to various parameters and their combination in a multivariate manner as suggested recently by Fellows [Fel09] and Niedermeier [Nie10]. Such a study then helps to understand where the complexity of the problem comes from and why it is so hard to solve.

Chapter 2

Preliminaries

2.1 Sets, numbers, languages, formulas

Through the text we use standard notations. By $\mathbb{N}$ we denote the set of all positive integers (*natural numbers*), whereas $\mathbb{N}_0$ denotes all non-negative integers.

Set of all subset of a set S is denoted $\mathcal{P}(S)$. If S is a set and $r \in \mathbb{N}_0$ then $\binom{S}{r} := \{A \subseteq S \mid |A| = r\}$ denotes the set of all r-element subsets of S. As usual, we let $|A|$ denote the cardinality of the set A. We say that $V_1, V_2, \ldots, V_r$ is *a partition of a set* V if and only if $\bigcup_{i=1}^{r} V_i = V$, and $\forall i, j, 1 \leq i < j \leq r : V_i \cap V_j = \emptyset$.

For a function h and a subset S of its domain, we denote the restriction of h to S by $h|_S$. We use the standard $O()$ notation to compare asymptotical growth of functions, although when talking about exponential functions sometimes the $O^*()$ notation is used, which suppresses the polynomial factors of the functions.

An *alphabet* is any finite set. Most often the alphabet $\{0, 1\}$ can be used for our purposes. Elements of the alphabet are called *symbols*. A *word* or a *string* in an alphabet is a finite sequence of symbols. We denote by Σ^* the set of all words in the alphabet Σ and by $|y|$ the length of the word y, i.e. the number of (occurrences of) symbols in it. A *language* is a set of words.

In a *decision problem* we are given some input and a guestion and the task is to decide whether the answer to this question is yes or no. When studying decision problems we encode them in some suitable alphabet. This way the set of all inputs to the decision problem with the positive answer corresponds to some language over this alphabet. Hence we use the terms decision problem and language as synonyms. We also assume that it is easy to recognize whether a string constitutes an encoding of an input of the problem or not. More details on the decision problems and their encodings can be found in any standard computational complexity textbook, e.g. [AB09].

The *running time* or *time complexity* of an algorithm is the maximum number of steps it performs on inputs of given length. We do not examine the space complexity of algorithms in this text. When talking about algorithms we

describe them and measure their complexity on the RAM (Random Access Machine) model with unit cost per arithmetic operation, with the restriction that the binary encoding of every number involved in the computation must be polynomial in the input size (so we cannot abuse this model). For complexity considerations the Turing Machine (TM) model is used more often. Although the time complexity with respect to this models generally differs, if the question is only whether there is an algorithm running in polynomial time (*polynomial algorithm*), the model chosen does not matter [CR72]. The class of all decision problems having a polynomial algorithm that correctly decides whether the input string is in the corresponding language or not is called P. Such problems are called *polynomially solvable*.

The class NP (stands for nondeterministic polynomial time) contains problems having certificates of solution that can be checked in polynomial time. More precisely for such languages there is a polynomial-time checker algorithm taking inputs formed by two strings, such that for every string in the language there is a *witness* string of size polynomial in the size of the original string that makes the checker accept this pair of strings. By contrast, for a string not in the language, there is no complementary string of polynomial size that would make the checker accept.

A problem is *NP-hard* if every problem in NP can be reduced to it in polynomial time and *NP-complete* if it is NP-hard and in NP. A language $L \subseteq \Sigma^*$ is in coNP if its complement $\Sigma^* \setminus L$ is in NP. Finally, if there is a function $a : \mathbb{N} \to \Sigma^*$ (*advice*), a polynomial $p : \mathbb{N} \to \mathbb{N}$, and a polynomial algorithm $A(x, y, z)$ such that for every $n \in \mathbb{N}$ the length of the string $a(n)$ is at most $p(n)$ and x is in the language L if and only if there is no $y \in \Sigma^*, |y| \leq p(|x|)$ that would make $A(x, y, a(|x|))$ accept, then the language L is in the class coNP/poly. In this case we also say that it is in coNP with a polynomial advice, or it has coNP circuits. Refer to [AB09] for a deeper account on computational complexity.

2.2 Graphs

Our notation in the graph theory is standard, for the terms that we forgot to mention here, we refer the reader to any basic graph theory textbook as Matoušek and Nešetřil [MN09] or West [Wes96].

In most of the text we speak about undirected graphs. A *graph* G is a pair (V, E), where $V = V(G)$ is the set of *vertices* and $E(G) = E \subseteq \binom{V}{2}$ is the set of edges. When we talk about several graphs, we sometimes call the vertices of some of them *nodes* to distinguish them from the vertices of the others. We consider only simple finite loopless graphs thorough the text. We denote the set of all such graphs $\mathcal{G}$. If there is an edge between two vertices u and v, that is $\{u, v\} \in E$, we say that the vertices u and v are *adjacent* or *neighbors* and also that they are endpoints of the edge $\{u, v\}$. A *complement* of a graph G is a graph

$\overline{G} = (V, \binom{V}{2} \setminus E)$.

A graph H is a *subgraph* of a graph G if $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$ and it is an *induced subgraph* if $E(H)$ equals $E(G) \cap \binom{V(H)}{2}$. Conversely, H is a *spanning* subgraph if $V(H) = V(G)$. If S is a subset of the vertex set $V(G)$ then we denote the subgraph $(S, E(G) \cap \binom{S}{2})$ induced by the set S by $G[S]$, while $G \setminus S$ denotes the graph $G[V(G) \setminus S]$ (especially if $v \in V(G)$ then we use $G \setminus v$ in the meaning of $G \setminus \{v\}$). Similarly, if e is an edge of G, then $G \setminus e$ denotes the graph $(V(G), E(G) \setminus \{e\})$.

A *k-clique* is a graph or a subgraph on k vertices having an edge between any two of them. Conversely an *k-independent set* is an induced subgraph on k vertices with no edges. A *path* of length $t \in \mathbb{N}_0$ is a (sub)graph formed by distinct vertices $v_0, v_1, \ldots, v_t$ and edges $\{v_0, v_1\}, \{v_1, v_2\}, \ldots, \{v_{t-1}, v_t\}$. The vertices v_0 and v_t are its endpoints. Finally, a *cycle* of length $t \geq 3$ is a graph or subgraph formed by distinct vertices $v_1, \ldots, v_t$ and edges $\{v_1, v_2\}, \{v_2, v_3\}, \ldots, \{v_{t-1}, v_t\}, \{v_t, v_1\}$. If weights on edges are given then the length of a path refers to the sum of weights of the involved edges.

We call a graph *connected* if there is a path between any two of its vertices. Maximal connected subgraphs of a graph are called *connected components*. If a graph does not contain a cycle as a subgraph then it is called a *forest*. A *tree* is a connected forest. A *distance* $dist_G(u, v)$ between two vertices u and v is the length of a shortest path between them. A *radius* of a graph is the minimum number r such that there is a vertex that is in distance at most r from any other vertex of this graph.

A (proper) *k-coloring* of a graph G is mapping $c : V(G) \to \{1, \ldots, k\}$ such that no two adjacent vertices receive the same color (number). The set of all vertices that receive a particular color in the coloring is called a *color class*. A graph is *k-colorable* if there is at least one proper k-coloring of it. Graphs that are 2-colorable graphs are also called *bipartite* and denoted (V_1, V_2, E) or $(V_1 \cup V_2, E)$, where V_1 and V_2 are the color classes. We also say that V_1 and V_2 are the partitions of a *bipartition* of the vertices.

The set of all vertices adjacent to a vertex v is called the *(open) neighborhood* of v and denoted $N(v)$ while $N[v] = N(v) \cup \{v\}$ denotes the *closed neighborhood* of v. The *degree* $\deg(v)$ of a vertex v is the size of its (open) neighborhood. A vertex is *isolated* if it has empty neighborhood. A graph is *d-regular* if every vertex has degree d. A 3-regular graphs are also called *cubic*. A *matching* is a 1-regular subgraph and it is *perfect* if it is spanning. A graph is *d-degenerate* if every its (non-empty) subgraph (including itself) has a vertex of degree at most d.

An *edge contraction* is an operation which removes an edge from a graph while simultaneously merging together its two endpoints, removing the possibly arisen paralel edges. The graph obtained from a graph G by contacting an edge e is denoted by $G \cdot e$. Note that any connected graph can be contracted to a (graph having) single vertex by means of edge contractions. A graph H is a *minor* of a graph G, if it can be obtained from G by a sequence of vertex deletions, edge

deletions and edge contractions. A graph H is an *induced minor* of G if it can be obtained by a sequence of vertex deletions and edge contractions only. A graph class $\mathcal{C}$ is *(induced) minor closed* if any (induced) minor of a graph in $\mathcal{C}$ is again in $\mathcal{C}$, respectively.

By *subdividing an edge* we mean replacing it by a path of length 2. Note that if we subdivide an edge and then contract any of the two edges of the newly introduced path, we obtain the original graph. A graph G is a subdivision of a graph H if it can be obtained from H by subdividing edges. A graph is *planar* if it can be embedded into a plane without edge crossings. It is well known that a class of planar graphs is proper minor closed and also closed under taking subdivisions. A class of graphs is *proper* if it is nonempty and does not contain all graphs.

A *directed graph (digraph)* is a pair $D = (V, A)$, where $V = V(D)$ is again the set of vertices (or nodes more often this time) and $A(D) = A \subseteq V \times V$ is the set of *arcs*. Although we generally allow loops in this case, they play no role in our consideration - they are of no use nor make any obstacle for the solution of our problems. We use (u, v) to denote the arc directed from the vertex u to the vertex v (*starting* in u and *ending* in v) and in this case we also say that u has an arc *to* v (it is an *in-neighbor* of v) and v has an arc *from* u (it is an *out-neighbor* of u).

The notion of (induced) subgraph works analogously as in the undirected case. A *directed path* from a vertex u to a vertex v is a subgraph of D formed by vertices $u = v_0, v_1, \ldots, v_t = v$ and arcs $(v_0, v_1), (v_1, v_2), \ldots, (v_{t-1}, v_t)$ for some $t \in \mathbb{N}_0$, while a *directed cycle* is a subgraph $(\{v_1, \ldots, v_t\}, \{(v_1, v_2), \ldots, (v_{t-1}, v_t), (v_t, v_1)\})$ for some $t \in \mathbb{N}$. We say that u is *connected to* v and v is *reachable* from u if there is a directed path from u to v. A graph $D = (V, A)$ is *strongly connected* if between each pair of vertices u and v there is a path from u to v and a path from v to u. It is *(weakly) connected* if its *underlying graph* $(V, \{\{u, v\} \mid (u, v) \in A \text{ or } (v, u) \in A\})$ is connected.

In a directed graph $D = (V, A)$, the *in-neighborhood* $N^-(u)$ (*out-neighborhood* $N^+(u)$) of a vertex u is the set of vertices which have arcs directed to u (from u), and the *in-degree* $\deg^-(u)$ (*out-degree* $\deg^+(u)$) of a vertex u denotes the size of the in-neighborhood (out-neighborhood) of the vertex u. The terms neighborhood and degree refer to the union of the in- and the out-neighborhood and the sum of in- and out-degree in this case, respectively. A vertex with $\deg^-(u) = 0$ is a *source*, and a vertex with $\deg^+(v) = 0$ is a *sink*.

2.3 Graph Widths

In this section we introduce some of the graph width measures that are used later in the text. We start by the most renowned one, the treewidth introduced by Robertson and Seymour [RS84].

A tree decomposition of a graph $G = (V, E)$ is a pair (T, σ), where T is a tree and $\sigma : V(T) \to \mathcal{P}(V)$ is a mapping assigning to each node x of the tree a subset V_x of vertices of the graph G called *a bag*, that satisfies the following:

(i) Every vertex $u \in V$ is in some bag,

(ii) for every edge $\{u, v\} \in E$ there is an $x \in V(T)$ such that $\{u, v\}$ is a subset of the bag V_x, and

(iii) for every vertex $u \in V$ if there are two bags V_x and V_y containing u then for every z on the unique path from x to y in T, u is contained in V_z.

The width of the tree decomposition (T, σ) is the size of the largest bag minus one.
The treewidth of a graph $tw(G)$ is the minimum width of a decomposition over all tree decompositions of G.

Restricting the tree involved in the tree decomposition to be a path we obtain path decompositions and the *pathwidth* $pw(G)$ of a graph, another measure introduced by Robertson and Seymour.

Chapter 3

Basic Definitions of Parameterized Complexity

In this chapter we present the basis of parameterized complexity with focus on tractability. As the fixed-parameter tractability became well known approach for solving NP-complete problems, there are several textbooks presenting comprehensively the field (except for some recent developments). The corner-stone was laid by Downey and Fellows in [DF98], more recently Niedermeier [Nie06] focused on the algorithmics and Flum and Grohe [FG06] on the complexity theory.

3.1 Parameterization and Parameterized Problem

Parameterized complexity is a framework to measure hardness of instances of computational problems in a multi-dimensional manner. To this end, we first need a notion of another measure apart from the size of the instance.

Definition 3.1. *A parameterization* of a decision problem $\mathcal{P} \in \Sigma^*$ is a computable function $\kappa : \Sigma^* \to \Sigma^*$.

Example 3.2. Suppose that we are going to investigate the problem of Vertex Cover. Here one is given a graph G and $k \in \mathbb{N}$ and the question is whether there is a set of vertices of size at most k, such that each edge of the graph has at least one endpoint in this set (such a set is called a *vertex cover*). Hence we assume, that an instance of this (decision) problem is an encoding of a pair (G, k) in some alphabet Σ, G being a graph and $k \in \mathbb{N}$. Then the so-called solution-size parameterization gives (the encoding of) k if the input represents a pair (G, k) and (the encoding of) 1 otherwise. If a parameterization is, as in this example, a projection of the input to one of its components, we will usually use the name of this component to denote the parameterization.

Parameterized complexity deals with parameterized problems:

Definition 3.3. By a *parameterized problem* $\mathcal{L}$ we mean any subset $\mathcal{L} \subseteq \Sigma^* \times \Sigma^*$, where Σ is some finite alphabet. An *instance* of the parameterized problem $\mathcal{L}$ is any pair $(x, k) \in \Sigma^* \times \Sigma^*$, where x is called the *main part* and k is the *parameter* (of the instance). We call (x, k) a *yes-instance* for the parameterized problem $\mathcal{L}$ if $(x, k) \in \mathcal{L}$ and a *no-instance* otherwise.

A parameterization provides a connection between classical decision problems and parameterized problems:

Definition 3.4. If $\mathcal{P} \subseteq \Sigma^*$ is a decision problem, and $\kappa : \Sigma^* \to \Sigma^*$ a parameterization, then we denote by κ-$\mathcal{P}$ the parameterized problem $\kappa\text{-}\mathcal{P} := \{(x, \kappa(x)) \mid x \in \mathcal{P}\}$. If the parameterization is clear from the context (especially for the so-called standard parameterization) we often omit its specification and use $\mathcal{P}$ also for the parameterized problem.[1]

Remark 3.5. Some authors [FG06] define the parameterized problem as a pair $(\mathcal{P}, \kappa)$ and require the parameterization to be a polynomial-time computable function $\kappa : \Sigma^* \to \mathbb{N}$. We prefer to use definition as stated above, as we neglect the possible time needed to evaluate the parameterization. There are also some parameterizations for which it is not possible to compute them in polynomial time, justifying our decision (see Section 4.4).

We defer the discussion on whether the parameter must be an integer after the main definition of parameterized algorithmics.

3.2 Fixed-Parameter Tractability

The main goal of parameterized complexity was always to decide which parameterized problems are fixed-parameter tractable and which are not.

Definition 3.6. We say that a parameterized problem $\mathcal{L} \subseteq \Sigma^* \times \Sigma^*$ is *fixed parameter tractable* if there is an algorithm that correctly decides whether an instance (x, k) is in $\mathcal{L}$ in time $f(k) \cdot |x|^c$ for some function $f : \Sigma^* \to \mathbb{N}$ and some constant c. Such an algorithm is called a *parameterized algorithm* or *fpt-algorithm*. The class of all fixed parameter tractable problems is denoted FPT.

Remark 3.7. Distinguishing the cases $f(k) \leq |x|$ and $f(k) > |x|$ one can see that an $(f(k) \cdot |x|^c)$-algorithm also runs in time $(f(k))^2 + |x|^{2c}$. On the other hand, an $(f(k) + |x|^c)$-algorithm also runs in time $2f(k) \cdot |x|^c$, as we assume both $|x|$ and $f(k)$ to be at least one. Hence FPT is also a class of par. problems having $(f(k) + |x|^c)$-algorithms.

[1] As there is no standardized way to denote the parameterization of the problem, some traditional names of decision problems already use the dash. In these exceptional cases we keep the traditional names and, thus, the part before the dash is then not necessarily a parameterization.

Definition 3.8. A parameterized problem $\mathcal{L} \subseteq \Sigma^* \times \Sigma^*$ is in the class XP if there is an algorithm that correctly decides whether an instance (x, k) is in $\mathcal{L}$ in time $f(k) \cdot |x|^{g(k)}$ for some functions f and $g : \Sigma^* \to \mathbb{N}$.

Definition 3.9. We say that a (decision) problem $\mathcal{P}$ is (in)[2] FPT (in XP) with respect to the parameterization κ (or parameterized by κ) if the parameterized problem κ-$\mathcal{P}$ is in FPT (in XP), respectively. Again, if clear from context we sometimes omit the parameterization.

Example 3.10. Vertex Cover is FPT with respect to k.

The fundamental difference between the FPT and XP algorithmic running-times laid the corner-stone of parameterized complexity. It is obvious that FPT $\subseteq$ XP, while it is known that XP $\not\subseteq$ FPT [DF98].

Remark 3.11. We have allowed the parameter to be an arbitrary string. But in some cases (e.g. when talking about polynomial functions of the parameter) it is necessary to restrict the parameter to be formed by a (single) integer. Therefore some authors require the parameter to be an integer [FG06]. On the other, hand it is sometimes natural to consider for example a graph or a pair of numbers as a parameter. It is easy to check that if there is an fpt-algorithm deciding whether $(x, k) \in \Sigma^* \times \Sigma^*$ is a yes-instance of the problem running in time $f(k) \cdot |x|^c$ for some function $f : \Sigma^* \to \mathbb{N}$ and some constant c, then the algorithm also runs in time $f'(|k|) \cdot |x|^c$ for some $f' : \mathbb{N} \to \mathbb{N}$. Hence, we can replace the parameter $k \in \Sigma^*$ by the parameter $|k| \in \mathbb{N}$ if necessary, without affecting the membership of a particular problem in FPT (or XP). However, if the parameter particularly is formed by an r-tuple $(a_1, a_2, \ldots, a_r) \in \mathbb{N}^r$ of natural numbers, we prefer to replace it by the sum $a_1 + a_2 + \cdots + a_r \in \mathbb{N}$, as this preserves the polynomiality of functions of the parameter.

[2]The letters FPT are often used as an acronym for fixed-parameter tractable. Therefore we often say that a problem is FPT, instead of saying that it is *in* FPT

Chapter 4

Parameterizations

In the previous chapter we have said that parameterized complexity relies strongly on having further measure on the instances of the problem apart from the input size. In this chapter we discuss some of the measures used. We would like to remark that when talking about complexity of graph problems mostly the number of vertices n and sometimes the number of edges m are used to measure the size of the input. These quantities, although closely related to, are also quite different than the actual bit-size of the input. But they are quite unsuitable for our purpose by the following reason.

It is easy to see, that if a parameterization grows monotonically with the input size and is unbounded (or it is lower bounded by such a function) then any decidable problem is FPT with respect to this parameterization. Hence we head for parameterizations unrelated to the input size. Also, as the dependence of running time of fpt-algorithms on the parameter is mostly exponential or worse, there is a strong need for parameterizations that remain small on some reasonable part of the instances that are (though to be) practical.

4.1 Solution-Size and its Dual

As the vast majority of decision problems originate from optimization problems, their instances are usually equipped with a number k representing the size of the sought solution. Or, in view of the definition of the class NP, the size of the witness we search for. This number is widely used; such a parameterization is called the *standard parameterization* or the *parameterization by the size of the solution*.

The disadvantage of this parameterization is that we sometimes cannot expect it to be small. On the other hand, there is usually an input-size related upper bound for the parameter, the answer being trivial above it. Then we can use the parameter saying how far from this upper bound we can get. This is called *the dual parameterization* and the corresponding parameterized problem is called *the*

parametric dual of the original parameterized problem. For example, if we have a graph problem where we search for a vertex subset S of size k satisfying some condition, and the size of the vertex set V is n, then the dual parameterization assigns to such an instance the number $n - k$, that is the size of $V \setminus S$. Or, equivalently, we can keep the parameter to be k and change the question to be "Is there a set of size $n - k$?"

Special case of the solution-size parameterization is when the input is weighted, i.e., we are given weights on the elements that we can use in the solution an the question is, whether there is a solution with total weight not exceeding the given bound (budget). As a rule, in this case, it is necessary to normalize the weights in some sense, as otherwise (in most cases) any instance can be transformed to an instance with the budget 1 (dividing all the weights by the budget), effectively ruling out the possibility of the problem being FPT (unless it is in P) with respect to the parameterization by the budget. Hence we either allow only integer weights, or take the ratio between the budget and the minimum weight used as a parameter (or force the minimum to be 1).

4.2 Parameterization Above Tight Lower Bound

Sometimes there is an unbounded growing function f of the size of the input $|x|$, such that whenever the parameter k is less than this function $f(|x|)$, then the answer is trivial. This means always yes or always no, depending only on the problem considered. As an example consider Max d-Sat. Here one is given a propositional formula in form of conjunction of m clauses, each formed by exactly d literals and the question is whether it is possible to satisfy at least k clauses simultaneously. By a simple probabilistic argument one can show, that it is always possible to satisfy at least $1 - 2^{-d}$ fraction of the clauses. Hence whenever we are asked whether it is possible to satisfy $k \leq m(1-2^{-d})$ clauses we can simply answer yes. Otherwise there is less than $k/(1 - 2^{-d})$ clauses containing less than $d \cdot k/(1 - 2^{-d})$ variables and any brute force algorithm can be used to show the fixed-parameter tractability of the problem with respect to the parameter k.

Of course this algorithm is somewhat unsatisfactory. The question arises, wheather we can get a bit more, than what we are quaranteed. That is to pose a question: "Is it possible to satisfy at least $m(1 - 2^{-d}) + k$ clauses?" or, equivalently, parameterize the problem by $\max\{k - m(1 - 2^{-d}), 1\}$. If the lower bound used is tight, then such a parameterization is called the *parameterization above tight lower bound*. To see that the bound from our example is tight it is enough to consider a formula formed by a blocks of all 2^d possible clauses on some d variables, taking different variables for different blocks. As often the lower bound is obtained by some probabilistic argument, this kind of parameterization is also often called the *parameterization above average*.

There are several positive results for such parameterizations, to see the one

for the above example refer to [AGK+10].

4.3 Further Natural Parameters

Definition of some other problems already shows off some natural parameters. Graph problems often include several "request" that should be satisfied simultaneously. As the number of the request can be often assumed to be small it forms a natural measure to parameterize with.

A typical example of such problem is Steiner Tree, where we are given an edge weighted graph $G = (V, E)$, a subset of vertices T (called *terminals*) and a natural number $p \in \mathbb{N}$. The question is whether there is a connected subgraph of G of weight at most p containing all vertices of T. While p can be regarded as the size of the solution, $|T|$ provides a further natural parameter—we will see further in the text that even much more useful.

Another example can be found in the more recently studied k-Local Search for Traveling Salesperson [Mar08]. In this problem we are given a graph with positive weights on edges and a Hamiltonian cycle in it and the question is whether we can find a Hamiltonian cycle which uses at most k edges not used by the original cycle and its weight is smaller than the weight of the original one. Here the number k can be hardly called size of the solution. Similar parameter is involved in Conservative Coloring where we are given a graph which has all vertices except for one properly colored by k colors and we search for a proper coloring of that graph with k colors which differ from the original one on at most c places [HN10]. Again, c is a natural parameter which does not represent the solution-size.

Further natural parameters appear in the problems where we search for a consensus. For example in one problem in voting systems we are given several linear orders on the candidates representing the votes and we search for a linear order that is reasonably close to the given orders in a given distance measure. Such a problem offer several natural parameters as the number of voters, number of candidates, the sum of the distances to the consensus or the maximum distance and many further. Moreover for each of them there is a natural scenario, where it is reasonable to assume that this parameter will be small [BGN08].

Natural parameters appear also in many other areas, for example for geometrical problems in higher dimensions it is natural to consider the dimension of the problem as a parameter [Kna10]. For string problems the size of the alphabet is often considered, etc.

4.4 Structural Parameters - Graph Widths

In practice, many graph problems come with instances formed by sparse graphs, often having some additional structure. The most obvious sparsity measure is the average degree or the closely related degeneracy (see Section 2.2 for the definitions). Although these parameters are quite useful for many problems, for many others they provide too little structure.

The situation improves when we restrict our attention to the most studied class of sparse graphs — planar graphs. Or more generally we parameterize our problems be the genus of the graph, which is FPT to determine [Moh99]. Although many hard problems become tractable on graphs of bounded genus, still there are many more, that are NP-complete even on planar graphs Hence many other measures — graph widths — were introduced to really catch the structure of graphs. Nowadays there are dozens if not hundreds of such widths.[1]

If we should present some of the widths, we have to start with *the treewidth*. Introduced by Robertson and Seymour [RS84] in 1984 it is the most widely recognized and definitely the most successful of them.

Interestingly it is NP-hard given G and k to decide whether the treewidth of G is at most k [ACP87], while it is FPT parameterized by k as there is an algorithm, whose running time is linear for every fixed k [Bod96]. Since the algorithm actually finds the decomposition in case one exists, this implies the existence of an optimal decomposition with linear number of nodes. While this algorithm is completely impractical due to the huge function $f(k)$ involved in the running time, there are practical algorithms constructing a decomposition of width $3k+2$ if the treewidth of G is at most k [Ree92] and, if this is not sufficient, heuristic approaches are used [BK10].

Hence, if we parameterize the problem by the treewidth, it is convenient (but should be mentioned) to assume that we are given a decomposition, and the complexity of the algorithm is actually measured with respect to the width of the decomposition given. We just have to bear in mind that if we don't provide the algorithm with the optimal decomposition, the bound on the running time of the algorithm being an expression of the treewidth should be actually considered as an expression of the width of the decomposition given.

After the great success of the treewidth, the research aimed both to extend the tractability results to wider classes of graphs as well as to further restrict the class of graphs considered to achieve tractability for further problems. To compare the measures we use the following notion. For two graph measures $\kappa, \kappa' : \mathcal{G} \to \mathbb{N}$ we say that κ is *more restrictive* than κ' (κ' is *less restrictive* than κ) if there is a function $g : \mathbb{N} \to \mathbb{N}$ such that for every graph $G \in \mathcal{G} : \kappa'(G) \leq g(\kappa(G))$. If κ is both less and more restrictive than κ' then we say that they are *equivalently*

[1] Over 40 were mentioned on the 2009 workshop on Graph Classes, Optimization, and Width Parameters (GROW).

restrictive.

The intuitive meaning is that if we have a subclass of graphs such that the values of some width are bounded by a constant on that class, then the values of any less restrictive width are bounded by a constant too. For example the pathwidth is more restrictive than the treewidth. Also if a problem is FPT parameterized by κ (without giving the decomposition) and κ' is more restrictive than κ then the problem is automatically also FPT with respect to κ'.

More related to our results are some widths that are more restrictive than the treewidth. First, restricting the tree involved in the tree decomposition to be a path we obtain *the pathwidth* $pw(G)$ of a graph. The restricted structure of the decompositions can help to devise algorithms, but there are very few examples of problems being intractable parameterized by the treewidth and tractable parameterized by the pathwidth.

Another approach is represented by *the feedback vertex set number* $fvs(G)$, which is the size of the minimum feedback vertex set in the graph G. A subset of vertices S is a *feedback vertex set* for the graph G if the graph $G \setminus S$ is a forest. It may seem strange at first sight to use the result of one optimization problem as a parameter for another problem, but such parameterizations turn to be useful. A natural restriction in this case for the decision version of the "parameterizing" optimization problems is to be FPT with respect to the solution size. We also need it to provide some structure that we can further use. In the case of feedback vertex set number, we can start by doing the brute force on the (small) feedback vertex set and afterwards the rest is just a forest that we can hopefully treat easily. The feedback vertex set number is more restrictive than the treewidth as it can be seen that always $tw(G) \leq fvs(G)$. To see that, consider the width-1 tree decomposition of the forest $G \setminus S$ and add S to every bag.

As further examples of this kind let us mention *the vertex cover number* $vc(G)$, which is the size of the minimum vertex cover of the graph and *the max leaf number* $ml(G)$, the maximum number of leaves in a spanning tree of a graph. Both of them are more restrictive than both the feedback vertex set number and the pathwidth. To see the later, the result of Kleitman and West [KW91] can be used showing that if the maximum number of leaves in any spanning tree of a graph G is k then G is a subdivision of a graph on at most $4k - 2$ vertices. Hence, maybe, the minimum number of vertices and the minimum number of edges of a graph our graph is a subdivision of would be a better candidate for a parameterization. But $ml(G)$ is used more often, maybe because there is no short name for the other two.

4.5 Multivariate Approach

Parameterized complexity tends not only to treat the problems under different parameterizations, but also in different scenarios, where the interplay between

more parameterizations influences the complexity of the problems. We are always interested in the existence of an algorithm with a certain running time. In such a scenario a parameterization κ can play several different roles (the roles are ordered from those putting the most restrictions on the parameterization to those putting no restrictions):

- It is a constant with a *known value* k_0 - we restrict our attention to instances having the value of κ equal to k_0. Our algorithm quite possibly does not work at all for instances having κ different then k_0 and thus no dependence of the running time on κ is to be examined. As an example consider the parameterization by the graph genus. Devising an algorithm just for planar graphs equals restricting the genus to be 0.

- It is a *constant* but the value is unknown - the algorithm works for all values of κ but the running time can depend on κ quite arbitrarily, in particular κ can appear in an exponent of the polynomial in the running time. A typical example is the number d in d-Hitting Set[2] when we examine families of sets, each having at most d elements. It is supposed that this measure will be very small when applying the scenario, say definitely less than 10.

- It is a *parameter* - then κ can only influence the multiplicative factor in the polynomial running time. With such an influence the algorithm can grow much higher, for some parameterized problems, values of parameter larger than 100 can still give reasonable running times.

- It is just a *variable* without any influence on the running time of the algorithm. This is included only for completeness, as this rather means that κ plays no role in this scenario.

To get better the idea, consider a scenario in which κ_1 has known value k_0, κ_2 is a constant, κ_3 forms the parameter and κ_4 is a variable. Then we are interested in the existence of an algorithm which works for all instances having κ_1 equal to k_0 and its running time on an instance x is bounded by $f(\kappa_2(x), \kappa_3(x)) \cdot |x|^{g(\kappa_2(x))}$ for some functions f and g.

It is important to note, that an FPT result for some scenario also applies to scenarios where each parameterization plays the same or more restricted role. On the other hand, a hardness result for a particular scenario also translates to scenarios that are less restrictive. In particular, an fpt-algorithm with respect to a single parameter also shows that the problem is in FPT with respect to the combination of this parameter and any other. Conversely, a hardness result for a combined parameterization means also hardness with respect to each single

[2] d-Hitting Set: Given a family of sets, each with at most d elements and $k \in \mathbb{N}$, decide, whether there are at most k elements, such that each set in the family contains at least one of them (is *hit*).

parameter involved in the combination. This way the number of scenarios we need to study to get the full picture can be significantly decreased.

Also note that it makes no sense to use several structural parameterizations with the same role if one of them is more restrictive than the other (see Section 4.4). Although there are several hardness results for a combination of two incomparable structural parameters such as the pathwidth and the feedback vertex set number, it seems complicated to use structures provided by two parameters in combined matter to develop an algorithm.

The main purpose of studying different scenarios is not only to provide people solving the problem the best suited tool for their particular case, but also to understand where the hardness of the problem comes from, which is very important from the theoretical point of view. Hopefully understanding of the problem hardness can lead to even better tractability results for it. More ideas about how to examine the problem in the fully multivariate manner can be found in [Fel09] and [Nie10].

Chapter 5

Kernelization Point-of-View

Kernelization is a natural formalization of a notion of effective polynomial preprocessing in terms of parameterized complexity. It is known by many names such as data reduction or reduction to a problem kernel. Of course efficient preprocessing is not only a domain of the parameterized algorithms. As the use of a kernelization makes no harm, it can be used prior to almost any approach for solving the problem, such as heuristics or approximation algorithms.

5.1 Basic Ideas

We present the basic ideas on an example of the now legendary kernelization for Vertex Cover, which is attributed to Samuel R. Buss in [DF98], but nowadays is considered rather folklore. Recall than in Vertex Cover we are given a graph G and a natural k and the question is whether there is a set of at most k vertices in which every edge has at least one endpoint. The standard parameter (considered here) is k.

First observation that can help to reduce the instance is that in an instance of Vertex Cover isolated vertices play no role. Hence we can replace G by $G \setminus I$ in our considerations, where I is the set of isolated vertices in G. This is usually called *a reduction rule*. To see its *correctness (soundness)* it is necessary to check that the instance produced by the rule is a yes-instance if and only if the original one is. In our case this means that G has a vertex cover of size at most k if and only if $G \setminus I$ does so, which is obvious.

Further consider a vertex v having more than k neighbors in G. It has to take part in any vertex cover of size at most k as the cover cannot contain all neighbors of the vertex. Thus G has a vertex cover of size k if and only if $G - v$ has a vertex cover of size at most $k-1$, immediately leading to another reduction rule. Note that, in contrast with the previous rule, this rule uses the value of the parameter - it is *parameter dependent*. Of course parameter independent rules are more desirable, as they can be used also on the optimization problem itself,

not only on its (parameterized) decision variant.

To complete the kernelization we need a *boundary lemma* saying that if the instance is *reduced* (none of the reduction rules can be applied to it any more) then either the answer is trivial, or the size of the instance is bounded in terms of the parameter. If the Vertex Cover instance is reduced with respect to the above two rules, then either it is a no-instance or it has at most $k^2 + k$ vertices, as each vertex is non-isolated and thus must be in the cover or a neighbor of one of (at most) k cover vertices, each of them having degree at most k. Obviously, reduced yes-instance has also at most k^2 edges and thus can be described by $O(k^2 \cdot \log k)$ bits.

Now we are ready to give a formal definition of kernelization:

Definition 5.1. *A kernelization* of a parameterized problem[1] $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$ is a polynomial time evaluable function A that on the input $(x, k) \in \Sigma^* \times \mathbb{N}$ produces an instance $(x', k') := A((x, k)) \in \Sigma^* \times \mathbb{N}$ such that

- (x', k') is in $\mathcal{P}$ if and only if (x, k) is in $\mathcal{P}$, and
- there is a function $g : \mathbb{N} \to \mathbb{N}$ such that $|x'| \leq g(k)$ and $k' \leq g(k)$.

The instance (x', k') is called *problem kernel* and the function g is called *the size of the kernel*. If we talk about $g(k)$-kernel for some function g we always mean, that there is a kernelization with g being the size of the kernel.

5.2 Further examples

First we note that that our first example translates also to a generalization of Vertex Cover called d-Hitting Set: Given a family of sets, each with at most d elements and $k \in \mathbb{N}$, decide whether there are at most k elements, such that each set in the family contains at least one of them (is *hit*). Here, in conflict with our notion (but in accordance with the literature), we consider k to be the parameter and d to be a fixed constant, the case of $d = 2$ is exactly Vertex Cover.

Of course it doesn't make sense to consider elements not contained in any set of the family. Similarly a superset of another set in the family is redundant in the instance. The mentioned "high degree" rule can be generalized to this case by a notion of a sunflower:

Definition 5.2. Sets $S_1, \ldots, S_r$ form *a sunflower* if there is a (possibly empty) set $A := \bigcap_{i=1}^{r} S_i$ (*center*) such that the intersection of any two sets S_i and $S_j, j \neq i$ is equal to A or, equivalently, if the sets $S_i \setminus A$ (*petals*) are disjoint.

[1] We restrict ourselves here to parameters formed by a single integer, as we want to later speak about polynomial functions of the parameter; see also Remark 3.5.

If an instance of d-Hitting Set contains a sunflower with more than k petals then in order to hit each set in the sunflower we have to select an element from the center. This can be represented by adding the center into the instance and removing all supersets of the center (in particular all sets of the sunflower).

Using the so-called Sunflower lemma or the Erdös-Rado Lemma (see [FG06, Lemma 9.7, p. 211]) one can show that an instance which has more than $k^d \cdot d \cdot d!$ sets contains a sunflower with more than k petals that can be found in polynomial time in both k and the size of the instance. Thus if a reduced family contains more sets, it is a no-instance. With the $k^d \cdot d \cdot d!$ sets containing altogether at most $k^d \cdot d^2 \cdot d!$ elements we arrive at an $O(k^d \cdot d^2 \cdot d! \cdot (2 \log d + d \cdot (\log k + \log d)))$-kernel. We mention in one of the next sections that this is asymptotically optimal.

The above example is included since most kernelizations in fact use some kind of high degree rule. It is a *local rule* in the sense that it only examines an (r-)neighborhood of a vertex (or an element) an tries to replace it with something simpler with the same function. On the other hand there are also *global rules* examining the structure of the whole graph at once. We present an example of a crown rule in one less ordinary, although well known application on the parametric dual of Graph Coloring. Here the question is given a graph G on n vertices and $k \in \mathbb{N}$ can G be properly colored by at most $n - k$ colors? It is known as "How to Save k Colors in $O(n^2)$ Steps" [CFJ04].

Definition 5.3. *A crown decomposition* of a graph $G = (V, E)$ is a partition $C \cup H \cup B = V$ of the vertex set, such that

- C is an independent set,
- there are no edges between C and B, and
- there is a matching of size $|H|$ between H and C.

The sets C, H, and B can be thought of as *the crown, the head*, and *the body*, respectively.

Lemma 5.4. *There is an algorithm, that in polynomial time finds either*

- *a matching of size at least $k + 1$, or*
- *a crown decomposition $C \cup H \cup B$ such that $|B| \leq 3k$.*

The proof can be found in [CFJ04] and we omit it here.

To solve the dual of coloring, we run the algorithm on the the complement $\overline{G}$ of the graph G. If we find a large matching, then obviously we can color the matched vertices by a same color, obtaining a coloring with less than $n - k$ colors. If $C \cup H \cup B$ is a crown decomposition of $\overline{G}$ then C is a clique in G that is connected to every vertex of B. Hence each vertex of C needs its own color that, furthermore, cannot be used on B. On the other hand, due to the matching

between H and C (in $\overline{G}$) it is possible to use colors of C for the vertices of H. Thus G can be colored with $n-k$ colors if and only if $G[B]$ can be colored by $|B|-(k-|C|)$ colors. Since B has at most $3k$ vertices, this directly yields a kernel.

We remark, that via the crown decomposition we can similarly get a kernel with $3k$ vertices (of bitsize $O(k^2)$) also for Vertex Cover. A survey of other results on kernelization can be found in [GN07].

5.3 FPT means Kernelization

The following theorem gives a notification of the importance of kernelization for parameterized complexity.

Theorem 5.5. *A decidable parameterized problem is FPT if and only if it has a kernelization.*

Proof. First suppose that $\mathcal{P}$ is FPT, that is, there is an algorithm running in time $f(k) \cdot n^c$. The promised kernelization works as follows: If $f(k) \leq n$ then it solves the instance in $f(k) \cdot n^c \leq n^{c+1}$-time and outputs some (constant size) trivial instance being in the language if and only if the input one is. Otherwise $n < f(k)$ and it outputs the same instance without a modification.

For the other direction it is enough to use any algorithm (ensured by the decidability) on the output of the kernelization. □

Having the above theorem in hand it may seem that having a kernelization is just another name for being FPT. The important thing here is the size of the kernel. From the above theorem we only get super-polynomial kernels (for NP-hard problems). In fact, as noted by Bodlaender [Bod09], we cannot get a constant size kernel for NP-complete problem (unless P=NP). What we can get are kernels of polynomial size as we have seen in our example and such kernels are of broad interest, as not only they often lead to the best known fpt-algorithms for a particular problem, but as we have mentioned earlier, they can be used in virtually any approach to solve the problem.

5.4 On the Non-Existence of Polynomial Kernels

As we have already said, a (polynomial) kernelization for a problem is more valued than just fixed-parameter tractability. The theory of parameterized hardness is capable of showing fixed-parameter intractability (see Chapter 7) but it is not fine-grained enough to provide any evidence that for a fixed-parameter tractable problem there is presumably no polynomial kernel, or to even show that some

kernel is asymptotically optimal in terms of its size. In this section we present the framework that is capable of such results (under certain complexity theoretic assumptions). We need the following definition of an algorithm doing an "OR" of several instances:

Definition 5.6. *A composition algorithm* for a parameterized problem $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$ is an algorithm that receives as input a sequence $((x_1, k), \ldots, (x_t, k))$, with $(x_i, k) \in \Sigma^* \times \mathbb{N}$ for each $1 \leq i \leq t$, uses time polynomial in $\sum_{i=1}^{t} |x_i| + k$, and outputs $(y, k') \in \Sigma^* \times \mathbb{N}$ with

- $(y, k') \in \mathcal{P}$ if and only if there is an $1 \leq i \leq t$ such that $(x_i, k) \in \mathcal{P}$ and
- k' is polynomial in k.

A parameterized problem is *compositional* if there is a composition algorithm for it.

We also need the following notion:

Definition 5.7. *An unparameterized version* of the parameterized problem $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$ is the language $\tilde{\mathcal{P}} := \{x \Diamond 1^k \mid (x, k) \in \mathcal{P}\}$, where $\Diamond \notin \Sigma$, and 1^k is a unary encoding of k with 1 being an arbitrary symbol of Σ.
It is important that in the unparameterized version of the problem the parameter forms a lower bound for the input size, as it is encoded in unary. For most graph problems this is the case anyway, thus taking the unparameterized version makes no difference.

The main theorem of the framework states:

Theorem 5.8 (Bodlaender, Downey, Fellows, and Hermelin [BDFH09]; Fortnow and Santhanam [FS08])**.** *Let $\mathcal{P}$ be a compositional parameterized problem whose unparameterized version $\tilde{\mathcal{P}}$ is NP-complete*[2]*. Then, if $\mathcal{P}$ has a polynomial kernel then NP⊆coNP/poly. this would imply a collapse of the polynomial hierarchy to the third level.*

To use the framework, it is necessary to show the composionality of the problem considered. For many graphs problems this is easy, the composition algorithm can simply return the disjoint union of the input graphs and leave $k' := k$. This works for example for k-Path, where one asks, whether a given graph G contains a path of a given length $k \in \mathbb{N}$, parameterized by k. But there are also problems for which the compositional algorithm is fairly complicated (cf. [DLS09]), sometimes surprisingly also using the positive results known for the problem.

It is also possible to transfer the result obtained on one problem to another problem for which we were unable to design compositional algorithm directly. For that purpose the following transformation is used:

[2]As we mostly deal with NP-hard problems, this requirement is fulfilled as long as the parameter can be upper-bounded by some polynomial of the input size.

Definition 5.9 (Bodlaender, Thomassé, and Yeo [BTY08])**.** Let $\mathcal{P} \subseteq \Sigma^* \times \mathbb{N}$ and $\mathcal{Q} \subseteq \Sigma^* \times \mathbb{N}$ be parameterized problems. We say that $\mathcal{P}$ is *polynomial parameter reducible* to $\mathcal{Q}$, written $\mathcal{P} \leq_{Ptp} \mathcal{Q}$, if there exists a polynomial time computable function $f : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ and a polynomial p, such that for all $(x, k) \in \Sigma^* \times \mathbb{N}$

- $(x, k) \in \mathcal{P}$ if and only $(x', k') := f(x, k) \in \mathcal{Q}$ and
- $k' \leq p(k)$.

The function f is called *polynomial parameter transformation.*

Proposition 5.10 (Bodlaender, Thomassé, and Yeo [BTY08])**.** *Let $\mathcal{P}$ and $\mathcal{Q}$ be the parameterized problems and $\tilde{\mathcal{P}}$ and $\tilde{\mathcal{Q}}$ be the unparameterized versions of $\mathcal{P}$ and $\mathcal{Q}$ respectively. Suppose that $\tilde{\mathcal{P}}$ is NP-complete and $\tilde{\mathcal{Q}}$ is in NP. If there is a polynomial parameter transformation from $\mathcal{P}$ to $\mathcal{Q}$, then if $\mathcal{Q}$ has a polynomial kernel then $\mathcal{P}$ also has a polynomial kernel.*

We believe that the usage of this proposition does not need any example, but we mention that such a reduction exists showing that k-Leaf Out-Branching has no polynomial kernel [FFL$^+$09]. In k-Leaf Out-Branching we are given a directed graph and $k \in \mathbb{N}$ and the question is whether the directed graph contains a subgraph in which every vertex except for one has in-degree exactly 1 and k vertices has out-degree 0. Another complicated transformation can be found in Dom et al. [DLS09], where it is shown that for certain problems, the suitably colored version of the problem has a kernel if and only if the uncolored version does.

Recently Dell and van Melkebeek [DvM10] proved by a method to some extent similar to the above framework, that the $O(k^d \log k)$-kernel for d-Hitting Set mentioned in Section 5.2 is presumably optimal up to a logarithmic factors. Namely they have shown, that there is no kernel of size $O(k^{d-\epsilon})$ for any $\epsilon > 0$, unless NP$\subseteq$coNP/poly. Furthermore they have shown that there are no $O(k^{2-\epsilon})$-kernels for a class of graph deletion problems, where the task is to delete at most k vertices from a given graph to obtain a graph that fulfill some fixed subgraph-hereditary graph property. Feedback Vertex Set, Bounded-Degree Deletion or Planar Deletion are examples of such a problems (Here the task is to delete at most k vertices to obtain a forest, a graph with bounded-degree, and a planar graph, respectively). We believe that this approach will provide many further tight kernel lower bounds in the future.

5.5 Notion of Kernelization Relaxed

As the number of problems unlikely to have polynomial kernels grows, several approaches to relax the notion of kernelization were made. The easiest way to do this is to desist from the somewhat artificial requirement, that the result of the procedure must be an instance of the same problem. More formally:

Definition 5.11. *A bikernelization* from a parameterized problem $\mathcal{P}$ to a parameterized problem $\mathcal{Q}$ is a polynomial time evaluable function A that on the input $(x,k) \in \Sigma^* \times \mathbb{N}$ produces an instance $(x',k') := A((x,k)) \in \Sigma^* \times \mathbb{N}$ such that

- (x',k') is in $\mathcal{Q}$ if and only if (x,k) is in $\mathcal{P}$, and
- there is a function $g : \mathbb{N} \to \mathbb{N}$ such that $|x'| \leq g(k)$ and $k' \leq g(k)$.

The instance (x',k') is called *a problem bikernel* and the function g is called *a size of the bikernel*.

Note that, similarly to Theorem 5.5, existence of a bikernel from $\mathcal{P}$ to a decidable problem implies that $\mathcal{P}$ is FPT. Moreover, if $\mathcal{P}$ and $\mathcal{Q}$ are NP-complete and the parameters are upper bounded by the size of the input, then $\mathcal{P}$ has a polynomial kernel if and only if there is a polynomial bikernel from $\mathcal{P}$ to $\mathcal{Q}$. Hence, this notion does not bring too much new.

In fact, the notion of bikernelization is very close to the so-called *annotated kernels*. *An annotation* is additional information added to handle partially solved instances during the run of an algorithm. An instance of the original problem can be viewed as a special instance of the annotated problem. On the other hand, although no polynomial kernel is known for the original problem, there can be a polynomial kernel for the annotated problem (sometimes called *annotated kernel*), as in this case some information can be stored in the annotation.

A well known example is the case of Dominating Set in Planar Graphs where we are given a planar graph and $k \in \mathbb{N}$ and the question whether there is a set of at most k vertices, such that each vertex not in this set has a neighbor in it (is dominated). Here the annotation divides vertices into two groups — those that are already dominated by vertices taken into a solution and removed from the graph — these vertices are left in the graph as they can still dominate some of the other vertices — those not dominated yet, which form the other group. This way we can obtain a linear kernel for (Annotated) Dominating Set in Planar Graphs [DF98]

Another approach is based on the observation that for practice it is still desirable to produce polynomially many (in terms of the input size) polynomial (in terms of the parameter) kernels. Such a reduction is often called *Turing kernelization*. Fernau et al. [FFL$^+$09] showed that there is a cubic kernel for Rooted k-Leaf Out-Branching a variant of k-Leaf Out-Branching, where the root of the sought out-branching is given. Hence k-Leaf Out-Branching can be transformed into n Turing kernels, each of size $O(k^3)$, although it was shown unlikely to have a standard polynomial kernel.

There is an active research in the area. The already mentioned results of Dell and van Melkebeek [DvM10] show that the existence of a Turing kernelization asking $n^{1-\epsilon}$ queries each of size polynomial in k, for some $\epsilon > 0$ would imply the collapse of the polynomial-time hierarchy.

5.6 Similar approach - Win/Win

After applying the reduction rules, kernelization basically branches into two cases according to the size of the reduced input: If the reduced input is too large than the answer is trivial. If it is small, then the running time of an algorithm we have will be hopefully affordable. It is not necessary to lower exactly the input size. The only requirement is that the low value of the decisive measure provides us with a (known) reasonable algorithm. This approach is sometimes called *Win/Win* that is we "win" if the measure is high and also if it is low. The measure used is mostly some graph width.

We give an easy illustrative example for Max Leaf, where given a graph and $k \in \mathbb{N}$ we search for a spanning tree of the graph with at least k leaves, k being the standard parameter. We start by doing (arbitrarily) a bread-first search from some vertex. If the tree obtained has at least k leaves, we won. Otherwise each layer contains at most $k-1$ vertices. Note that the edges only appear inside layers and between two consecutive layers. Hence we can easily obtain a path-decomposition of width $2k-2$, forming a bag from each two consecutive layers. As Max Leaf is FPT with respect to the pathwidth (by a dynamic programming on the path decomposition [Bod93], see Section 6.2 for such algorithms), we win in this case also.

Chapter 6

Further Algorithmic Methods

In this chapter we present the most important methods that are used to show fixed-parameter tractability.

6.1 Bounded Search Trees

If there is a method, that is similarly important in showing fixed-parameter tractability with respect to the solution size (or its dual) as kernelization, it is definitely the method of Bounded Search Trees. Or, if you want, you can call it *Branching* or *Recursive algorithms*. As there is not so much theory related to this method, we cut its presentation in space.

The basic idea is very simple: Identify a set of objects, one of which must be in the solution and try all possibilities to add one of them to a solution set. Continue recursively searching for a one-smaller solution on each of the partially solved instances. This gives a *search tree* of the recursive calls. This works for the primal parameterization, if the dual parameterization is used, we talk about objects that will not be a part of the solution

The important thing making this a special case of brute-force algorithms is that the size of the search tree can be bounded in terms of the parameter. As the depth of the tree is usually bounded by the solution size, it remains to bound the number of solution possibilities in each call of the procedure. The set of objects is usually constant size, but sizes bounded by the parameter are good as well. The number of leaves of the search tree is then bounded by the k-th power of the size of the set, while the number of internal vertices in the tree is upper-bounded by the number of leaves. The time needed inside one recursive call (corresponding to one node of the tree) is usually polynomial.

More involved algorithms usually add more than one object at once into the solution (at least in some branches) doing the analysis of the size of the tree more complicated. Also several different branching rules for different situation usually form the whole algorithm. As the branching algorithms are widely used

in the area of moderately exponential exact algorithms for hard problems, the ways to compute (an upper bound for) the running time can be found in any standard textbook on algorithms (see for example [KT05]). For most of the complicated branching algorithms it is hard to come with lower bound matching (or approaching) the upper bounds and thus the actual time complexity is rather unknown, which makes it harder to compare different branching algorithms.

The easiest example is the folklore $O(2^k \cdot n)$-algorithm for Vertex Cover. It is directly suggested by the definition of the problem — as each edge has to have at least one endpoint in the cover, we try both possibilities, delete the appropriate vertex from the graph (together with the incident edges), and continue recursively on the rest, searching for a cover of size at least by one smaller. Obviously there is no cover of size 0 for a graph with at least one edge, while for an edgeless graph the empty set is a cover of size 0. It is immediate that such a search tree has at most 2^k leaves and, thus, $O(2^k)$ nodes, the time needed to process each node being proportional to the number of vertices.

The algorithm can be improved by employing the idea that either a vertex is a part of the cover, or all its neighbors are. In this case, the bigger the degree of the vertex, the larger the progress we make (in one of the branches). Hence it is preferable to process the vertices of the highest degrees first. Once there is no vertex of degree at least 3 anymore, the minimum vertex cover for the graph can be found in linear time. Hence whenever we branch, one branch has the parameter reduced by at least 3, while the other by one. The resulting tree has $O(1.4656^k)$ nodes, the time spent in each node is still linear, yielding an $O(1.4656^k \cdot n)$-algorithm. This can be further improved to $O(1.4656^k \cdot k^2 + k \cdot n)$ by first using the Buss' Kernelization. Also note that $1.4656^k \cdot k^2$ is $O(1.4657^k)$ and therefore we can omit the factor polynomial in k as the base of the exponent was already rounded up.

Further example is d-Hitting Set, which also admits a natural search tree algorithm. Here for $d \in \mathbb{N}$ fixed constant and $k \in \mathbb{N}$ parameter we are given a family of sets, each with at most d elements and we should find at most k elements that hit every of the given sets. That is, the solution must contain at least one element out of each set. There is nothing easier than to take one set and try each element as the one that hits this set. Delete all sets hit by this element, decrease the parameter and continue recursively. We arrive at an $O(d^k)$ search tree for the problem.

Observe that a solution containing more elements of the set is considered in several branches of algorithm — in each branch that corresponds to some of the elements finally taken into the solution. We can actually partially avoid this inefficiency. To this end, we need the following reduction rule: Whenever an element u is contained in each set, where v is contained, delete v from all sets (without changing the value of the parameter), as it is never worse to take u whenever v should be taken. Now we force our algorithm to only consider the solutions containing the first element of the set (in some order) in the first

branch and delete it (as unusable) in all other branches. For each of the elements u different from the first one, there is always a set that contains the first element but not the element u. This set is not hit by the element u and has at most $d-1$ elements as the first element was deleted from it. Therefore it allows for better branching in the recursive call. This approach suggested in [NR03] brings the base of the exponent down to

$$\alpha(d) = \frac{d-1}{2}\left(1+\sqrt{1+\frac{4}{(d-1)^2}}\right),$$

which is a significant improvement at least for small values of d. For example $\alpha(3)$ roughly equals to 2.41.

It is important to note, that in the above example, the rule has to be applied at the beginning of each recursive call, as otherwise the condition can be violated during the execution. This increases the polynomial time needed in each node. On the other hand, this is greatly balanced by the improvement of the exponential factor. Furthermore this time can be decreased using the kernelization from Section 5.1. Usually (as in this case) the reduction rules are also used in between the branchings, which is sometimes called *interleaving* (of the kernelization and the search tree). This usually improves the performance both practically and theoretically. Quite typically an fpt-algorithm is formed by a set of rules, some of them being reduction rules (hopefully yielding a kernelization) and some of them being branching rules.

6.2 Dynamic Programming and Algorithmic Meta-Theorems

Dynamic programming is definitely the most successful technique for problems parameterized by something else than the solution-size or its dual. It is based on finding the solutions for the subproblems of the original problem, storing them in a table and then combining the solution for smaller subproblems to obtain a solution for larger subproblems.

Although the success of the method is very much connected with the success of the treewidth and other structural measures, we prefer to start by an example for Steiner Tree parameterized by the number of terminals to be connected.The algorithm we present is a modification of the famous Dreyfus-Wagner Algorithm [DW72] as presented in [DYW$^+$07].

In Steiner Tree we are given a graph $G = (V, E)$ with integral weights on edges $w : E \to \mathbb{N}$, a set of terminals $T \subseteq V$ and integers $p \in \mathbb{N}$. The question is whether there is a tree of cost at most p containing all the terminals.

We use two tables S and D: For any $X \subseteq T$ and $r \in V$ the table entry $S(r, X)$ will store the smallest weight of a tree that contains vertices of $X \cup \{r\}$

(for a better imagination it can be viewed as rooted in r) and the table $D(r, X)$ will store an auxiliary number that in most cases equals the weight of a smallest tree (rooted in r) that contains vertices of $X \cup \{r\}$ in which either r has degree at least two or $r \in X$. As the answer is trivial for $|T| \leq 1$, we assume $|T| \geq 2$.

The algorithm proceeds through all sets $\emptyset \neq X \subseteq T$ from the smaller to the larger ones and for each of them does the following two things: First, if X is a singleton then we set $D(r, X) := S(r, X) := 0$ for $\{r\} = X$ and $D(r, X) := S(r, X) := \infty$ for each $r \in V \setminus X$. Otherwise, for every $r \in V$, we set

$$D(r, X) := \min_{\emptyset \neq Y \subsetneq X} (S(r, Y) + S(r, X \setminus Y)). \tag{6.1}$$

Second, we obtain the values of $S(r, X)$ for every $r \in V$ as

$$S(r, X) := \min_{v \in V} (D(v, X) + dist_G(r, v)), \tag{6.2}$$

where $dist_G(r, v)$ is the length of the shortest path between r and v in G. This is done by running Dijkstra's Algorithm [Dij59] on $S(r, X)$ initializing $S(r, X) := D(r, X)$ for every $r \in V$. The result of the whole algorithm is obtained as $S(r, T \setminus \{r\})$ for an arbitrary $r \in T$.

The overall correctness of the algorithm follows from the claim, that the algorithm correctly computes $S(r, X)$ for every $r \in V$ and $\emptyset \neq X \subseteq T$. This is easily seen if $X = \{x\}$ as in this case Dijkstra's Algorithm in fact computes the shortest path from r to x. We further proceed by the induction on the size of the set X.

Next we show that whenever the algorithm assigns a value t to a cell $S(r, X)$ or $D(r, X)$ of the tables, there is a connected subgraph of weight at most t containing the vertices $X \cup \{r\}$ justifying that. If the value of $D(X, r)$ is set according to the equation 6.1 then we obtain such a graph as the union of the graphs for $S(r, Y)$ and $S(r, X \setminus Y)$. Similarly if the recurrence 6.2 is used, the graph is obtained as the union of the shortest path from r to v with the graph for $D(v, X)$.

Finally suppose that $|X| \geq 2$, the claim holds for every nonempty proper subset of X and there is a tree T' containing $X \cup \{r\}$ of weights smaller than $S(r, X)$. Denote by v the vertex closest to r in T' that is either in X or of degree at least tree in T' and P_v the (possibly trivial) path between r and v.

If $v \in X$ then $T' \setminus (V(P_v) \setminus v)$ forms a tree for $\{v\} \cup (X \setminus \{v\})$ and hence is lower bounded by $S(v, (X \setminus \{v\})$, due to our assumptions, as $\emptyset \neq (X \setminus \{v\}) \subsetneq X$. Thus $w(T') \geq S(v, (X \setminus \{v\}) + w(P_v) = S(v, (X \setminus \{v\}) + S(v, \{v\}) + w(P_v) \geq D(v, X) + dist_G(r, v) \geq S(v, X)$ — a contradiction.

Otherwise $T' \setminus V(P_v)$ has more components. Then denote Y the subset of X contained in the first component. The subtree of T' induced by the first component together with v is a tree for $Y \cup \{v\}$ and therefore is lower bounded by $S(v, Y)$ due to our assumptions, as $\emptyset \neq Y \subsetneq X$. Similarly, removing the first component and $P_v \setminus v$ we obtain a tree for $(X \setminus Y) \cup \{v\}$ lower bounded by

$s(v, X \setminus Y)$. Thus $w(T') \geq s(v,Y) + s(v, X \setminus Y) + w(P_v) \geq D(v,X) + dist_G(r,v) \geq S(r,X)$, contradicting our assumptions.

As to the time complexity, the equation 6.1 yields two table lookups for each combination of $\emptyset \subsetneq Y \subsetneq X \subset T$ and each $r \in V$, hence the total time needed to evaluate the recurrence 6.1 can be bounded by $O(3^{|T|} \cdot n)$. Further, in each of the $2^{|T|}$ iterations of the cycle we execute once Dijkstra's Algorithm, which can be implemented to run in time $O(n \log n + m)$ [FT87]. Hence the whole algorithm runs in time $O(3^{|T|} \cdot n + 2^{|T|}(n \log n + m))$.

As we have said, dynamic programming is, among parameterized algorithmics, mostly used in connection with the treewidth. To this end, usually the following modified decomposition is used:

Definition 6.1. A tree decomposition (T, σ) of a graph $G = (V, E)$ is called *nice* if it has a distinguished root and each node x is either a leaf or

- it has exactly one child y and there is a vertex $v \in V$ such that $V_x = V_y \cup \{v\}$ (such a node is called *introduce* node), or
- it has exactly one child y and there is a vertex $v \in V_y$ such that $V_x = V_y \setminus \{v\}$ (*forget* node), or
- it has exactly two children y and z such that $V_x = V_y = V_z$ (*join* node).

We use G_x to denote a subgraph of G induced by the vertices of V_x and all vertices that appear in the bags of the subtree of T rooted in x.

Due to the following observation we can always assume that the decomposition we are given is a nice decomposition with $O(k \cdot n)$ nodes.

Observation 6.2. *Given a tree decomposition (T, σ) of width k with $O(n)$ nodes, we can modify it to a nice tree decomposition with $O(k \cdot n)$ nodes in $O(k \cdot n)$ time.*

We demonstrate the use of dynamic programming on a problem parameterized by the treewidth on Independent Set. In Independent Set we are given a graph and search for a maximum size independent set. A subset of vertices is called *independent* if no two of the vertices are connected by an edge. We also assume that the tree decomposition of width $tw(G)$ is given on the input.

We associate with each node x a table A_x indexed by all subset of V_x. The number stored in the table on index S will represent the maximum independent set I in G_x that intersect V_x exactly in S or $-\infty$ if S itself is not an independent set.

We fill the tables from leaves of the decomposition to the root assuming that by the time we process a node, all its children were already processed (this is usually called *bottom-up fashion*). For the leaves we set $A_x(S) := |S|$ if S is an independent set and $A_x(S) := -\infty$ otherwise. If x is an introduce node with child

y, where the vertex v is introduced, we set again $A_x(S) := -\infty$ if S is not an independent set. Otherwise if v is contained in S, we set $A_x(S) := A_y(S \setminus \{v\}) + 1$ and for $v \notin S$ we let $A_x(S) := A_y(S)$. Similarly for a forget node forgetting vertex v and S an independent set we set $A_x(S) := \max\{A_y(S), A_y(S \cup \{v\})\}$. Finally, let x be a join node with children y and z and $S \subseteq V_x$. If either $A_y(S) = -\infty$ or $A_z(S) = -\infty$ then set $A_x(S) := -\infty$. Otherwise we let $A_x(S) := A_y(S) + A_z(S) - |S|$. The size of the maximum independent set in the graph is then the maximum number stored in the table of the root.

The algorithm obviously runs in time $O(2^k \cdot (k^2 + k \cdot n))$ as there are at most 2^{k+1} subsets of each V_x. The correctness follows from that the algorithm correctly fills all the tables. This is obvious for the leaves. Now we prove that for the other nodes under the assumption that it was already proven for all their children. For a forget node the independent set in G_x intersecting V_x in S intersects V_y in $S \setminus \{v\}$ and, hence, the correctness of the table directly follows from the correctness of the tables of the children. Similarly for the introduce node, the maximum independent set in G_x either contains v or not.

For the join node, assume that there is an independent set I in G_x intersecting V_x in S such that $|I| > A_x(S)$. Then $I \cap V(G_y)$ is an independent set in G_y intersecting V_y in S and thus $|I \cap V(G_y)| \leq A_y(S)$. Similarly $|I \cap V(G_z)| \leq A_z(S)$ and hence $|I| \leq A_y(S) + A_z(S) - |S| = A_x(S)$ –a contradiction. To finish the proof, note that if I_y is an independent set in G_y and I_z is an independent set in G_z both intersecting $V_x = V_y = V_z$ in S, then $I_y \cup I_z$ is an independent set in G_x as there are no edges between $G_y \setminus V_x$ and $G_y \setminus V_x$ (vertices of an edge are contained in one bag and, hence, can appear in only one subtree rooted at x).

There are many results similar to the example we gave. In fact, for any problem, that is expressible in the so-called Monadic Second-Order Logic (MSOL), a similar algorithm exists.

The *Monadic Second Order Logic (MSOL)* over graphs uses the union of the vertex set and the edge set as its domain and there are two unary predicates — $V(x)$ and $E(x)$ to distinguish, whether the object x is a vertex or an edge, respectively — and one binary $I(v, e)$, which is *true*, if and only if v is a vertex and e an edge incident to it. The quantification can be done over objects and sets of objects (unary predicates), but not over relations of higher arity.

Often this variant is called MSO_2 as there is another variant MSO_1, where the domain is only formed by the vertex set and only one binary relation $adj(u, v)$ is available, coding the adjacency of the vertices. This variant is less powerful, as it is unable to quantify over sets of edges.

The following result justifies the importance of the concept of tree decompositions:

Theorem 6.3 (Courcelle [Cou92])**.** *Given an MSO_2 formula φ, there is an algorithm that, given a graph G together with its tree-decomposition of width w, decides whether $G \models \varphi$, that is whether G is a model for φ, in time $O(f(\varphi, w) \cdot n)$,*

where the function f is independent of G (it only depends on w and φ) and n is the number of vertices of the graph G.

The function f involved in the theorem is so huge, that it makes the theorem mainly of theoretical interest. One-purpose dynamic programming algorithms as the one for INDEPENDENT SET are to be used in praxis. The result can be also strengthened in the sense, that in same linear time we can actually also find the maximum of some linear objective function on the cardinalities of the involved set variables. Recall that the tree-decomposition of the graph can be obtained in linear time due to the result of Bodlaender [Bod96].

6.3 Color Coding

It is no surprise that it is often easier to search only for solutions of a certain type. The color coding technique does that by first coloring the vertices of the graph (adjacent vertices can receive the same color) and searching for a *rainbow* solution, that is, a solution in which no two vertices are of the same color. As we no longer search for a solution of certain size, but rather for a solution using once each of the colors, dynamic programming can be usually used for the search. Of course we do not always have to color vertices, there are many other parts of the instances to be colored, such as edges.

The following clever way of coloring ensures that each possible solution is colored in a rainbow manner at least in one of the colorings tried.

Definition 6.4. Let $k, n \in \mathbb{N}$. We say that $\mathcal{H}$ is a *k-perfect family of hash functions* from $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, k\}$ if for every $S \subseteq \{1, 2, \ldots, n\}$ of size k there is an $h \in \mathcal{H}$ such that $h|_S$ is a bijection between S and $\{1, 2, \ldots, k\}$.

Lemma 6.5 (Naor; Alon, Yuster, and Zwick [AYZ95])**.** *There is a k-perfect family of hash functions from $\{1, 2, \ldots, n\}$ to $\{1, 2, \ldots, k\}$ of size $|\mathcal{H}| = 2^{O(k)} \cdot \log n$ that can be constructed in time $2^{O(k)} \cdot n \cdot \log n$.*

To illustrate the technique we present the algorithm for k-PATH as stated in the original paper of Alon, Yuster, and Zwick [AYZ95]. Recall that in k-PATH we search for a path of given length k (parameter) in a given graph. We start by constructing the hash functions as guaranteed by the previous lemma. For each h of the functions constructed we assign to each vertex $v \in V$ the color $h(v)$ in $\{1, \ldots k\}$ and search for a path formed by one vertex of each color.

This is done by a dynamic programming. With each vertex $v \in V$ we associate a table $S_v(A)$ indexed by all non-empty subsets A of the color set $\{1, \ldots k\}$. The value on position $S_v(A)$ determines whether there is a path in G having an endpoint in v an using one vertex of each color in A. The table is initialized by setting $S_v(\{h(v)\}) := \mathit{true}$ and $S_v(A) = \mathit{false}$ for every other singleton set A. Then the sets A are processed from smaller to larger, and $S_v(A)$ is set to *true* if

and only if $h(v)$ is in A and there is a neighbor u of v with $S_u(A \setminus \{h(v)\}) = true$. There is a rainbow path if and only if $S_v(\{1, \ldots, k\})$ is *true* for some $v \in V$.

It is not hard to see that the dynamic programming works correctly and runs in time $O(2^k \cdot k \cdot |E|)$, we omit the proof here. The k-perfectness of the hash family ensures that if a path of length k exists, then it becomes a rainbow path by at least one function in the family and, hence, is found by the dynamic programming. Therefore the whole algorithm runs in time $O(2^{O(k)} \cdot m \cdot \log n)$.

The idea used in the example can be generalized to any class of graphs with low treewidth as follows:

Theorem 6.6 (Alon, Yuster, and Zwick [AYZ95])**.** *Let H be a directed or undirected graph on k vertices with treewidth t. Let $G = (V, E)$ be a (directed or undirected) graph. A subgraph of G isomorphic to H, if one exists, can be found in time $2^{O(k)}|V|^{t+1} \log |V|$.*

6.4 Iterative Compression

Iterative Compression is a method used almost exclusively for vertex deletion problems parameterized by the size of the solution, where the task is to delete some vertices from the graph to achieve some property. It iteratively produces a solution for subproblems that is already one vertex bigger than the solution we search for. Then it compresses the solution to hit the bound. If this is not possible then there is no solution for the whole instance. Otherwise some more vertices are considered in the subproblem so that the solution is again too big and the next iteration is executed.

Our example is for Odd Cycle Transversal, where we aim to delete at most k (parameter) vertices from a given graph G in order to make it bipartite. The algorithm was first proposed by Reed et al. [RSV04] and the simplified version we present appeared in [LSS09].

The main part of an algorithm based on iterative compression is the so-called *compression step*. Here, given a solution of size $k + 1$ we are trying to find a solution of size at most k or to show that no such solution exists. Before presenting it, let us show how to solve the whole problem once we have an algorithm A for the compression step.

Assume $V(G) = \{v_1, \ldots v_n\}$ and for $i \in \{k + 2, \ldots n - 1\}$ denote by G_i the graph $G[\{v_1, \ldots, v_i\}]$. First observe that for the graph G_{k+2} any set of size k constitutes a solution. Let us denote one of them S_{k+2}. Now for $i \in \{k + 2, \ldots n - 1\}$ assume we have a solution S_i of size at most k for G_i and we want to find one of size at most k for G_{i+1}. The set $S'_{i+1} := S_i \cup \{v_{i+1}\}$ is a solution for G_{i+1} of size at most $k + 1$. If it is of size at most k we denote $S_{i+1} := S'_{i+1}$. Otherwise we use the compression step to obtain a solution S_{i+1} of size at most k for G_{i+1} being given the solution S'_{i+1} of size $k + 1$. If there is no such solution for G_{i+1} then obviously there is no solution of size k also for the whole graph G.

At the end we either obtain a size-k solution S_n for $G_n = G$ or we know that there is no solution of size at most k.

It remains to solve the compression step. Assume that we are given a solution S' of size $k+1$ for a graph G and we search for solution S of size k. We start be trying all possible partitions of S' into $T \cup L \cup R$. The vertices in T will be a part of the new solution, while the vertices of L and R will be on the left and the right side of the bipartition provided by the new solution, respectively (they will definitely not be a part of the solution). If there is an edge inside $G[L]$ or $G[R]$ then there is no chance to find a solution corresponding to this partition and we continue with a further partition. Otherwise we make use of the following fact:

Fact 6.7. *If $H = (V_1 \cup V_2, E)$ is a bipartite graph with the partitions V_1 and V_2 then*

- *any trail from V_i to V_i has even length ($i = 1, 2$) and*
- *any trail from V_1 to V_2 has odd length.*

The graph $G \setminus S'$ is bipartite. Let us call the partitions A and B, and denote A_L, B_L the neighbors of the set L in A and B, respectively. Similarly A_R and B_R denote the neighbors of R. We further need the following lemma:

Lemma 6.8. *If X is a subset of $V \setminus S'$ such that $G \setminus (T \cup X)$ is bipartite with partitions V_L and V_R such that $L \subseteq V_L$ and $R \subseteq V_R$, then in $G \setminus (S' \cup X)$ there are no paths between A_L and B_L, A_R and B_R, A_L and A_R, and between B_L and B_R.*

Proof. Assume there is a path between A_L and B_L, then Fact 6.7 implies it has an odd length as it goes from one partition of $G \setminus S'$ to the another. Hence it can be prolonged to an odd trail from L to L in $G \setminus (T \cup X)$ which is a contradiction with $G \setminus (T \cup X)$ being bipartite. Similarly for the other combinations. □

Lemma 6.9. *If X is a subset of $V \setminus S'$ such that in $G \setminus (S' \cup X)$ there are no paths between A_L and B_L, A_R and B_R, A_L and A_R, and between B_L and B_R, then $G \setminus (T \cup X)$ is bipartite with partitions V_L and V_R such that $L \subseteq V_L$ and $R \subseteq V_R$.*

Proof. First note that in this case each path from L to L with internal vertices from $V \setminus (S' \cup X)$ is even. The same holds for such a path from R to R, while any such path from R to L is odd. Thus if $G \setminus (T \cup X)$ is bipartite then the partitions V_L and V_R can be taken such that $L \subseteq V_L$ and $R \subseteq V_R$.

Suppose that there is a cycle C in $G \setminus (T \cup X)$. If $C \cap (L \cup R) \neq \emptyset$ then C is even, as the graph $G \setminus S'$ is bipartite. Otherwise denote $v_1, \ldots, v_t$ vertices of $C \cap (L \cup R)$ in order as they appear on the cycle, and for simplicity set $v_0 := v_t$. The length of the cycle can be counted as $|E(C)| = \sum_{i=0}^{t-1} d_C(v_i, v_{i+1})$, where $d_C(v_i, v_{i+1})$ is the distance between v_i and v_{i+1} taken along the cycle C. It remains to observe that the number of indices i with $v_i \in L$ and $v_{i+1} \in R$ equals the number of i's with $v_i \in R$ and $v_{i+1} \in L$ and hence C is again even. □

Due to the above lemma, to finish the compression step it is enough to find X of size at most $k - |T|$ such that in $G \setminus (S' \cup X)$ there are no paths between the mentioned pairs of sets. But this is equal to finding a cut of size $k - |T|$ in $G \setminus S'$ between $A_L \cup B_R$ and $A_R \cup B_L$. This can be done in time $O(k \cdot m)$ using standard flow techniques [FF56]. As this is done at most 3^{k+1} times in the compression step and the compression step is executed at most n times, we obtain an $O(3^k \cdot k \cdot n \cdot m)$ running-time for the whole algorithm.

Iterative compression was several times the break-through tool on problems resisting the research attacks for a long time before. A survey of known results based on iterative compression can be found in [GMN09].

6.5 Greedy Localization

Similarly to the previous technique, greedy localization uses some solution that is not good enough as a starting point for the search for the desired one. In contrast to the previous method, this time we use an inclusion maximal solution that is found greedily.

The idea is best illustrated on Packing 3-Sets where we are given a system $\mathcal{C}$ of three-element subsets of a finite set S and an integer $k \in \mathbb{N}$. The task is to find a subsystem $\mathcal{C}'$ of at least k mutually disjoint sets. The result is due to Jia et al. [JZC04] and actually uses the idea twice.

The algorithm first greedily locates an inclusion maximal subsystem $\mathcal{C}'_0$ of system $\mathcal{C}$ formed by pairwise disjoint sets. We assume $|\mathcal{C}'_0| < k$ as otherwise we are done. Each set in an optimal solution must contain at least one element of $\bigcup \mathcal{C}'_0$ as $\mathcal{C}'_0$ is maximal. Hence an optimal solution fits into one of the following patterns:

$$\mathcal{C}^* = \{\{a_1, *, *\}, \{a_2, *, *\}, \ldots, \{a_k, *, *\}\}, \text{ where } a_i \in \bigcup \mathcal{C}'_0 \text{ are distinct.}$$

Our algorithm will try all such patterns. Now assume that we have a pattern P with some positions filled and some still carrying a wild-card symbol $*$. We greedily try to fill the pattern into a pattern P'. That is we take the incomplete sets in the pattern one by one and look for a 3-set of $\mathcal{C}$ that contains the elements prescribed by the pattern and the other elements in it are not used anywhere else in the partially filled pattern — that is they are nor in the prescribed pattern P neither they have been already used to fill other set in P'.

If we succeed to fill the whole pattern, we have a solution. Otherwise consider the set S which we were unable to fill. If every its completion contains elements of some other part of P, that means the pattern P cannot be realized. Otherwise to complete this set we have to use some elements already added to the other sets in the pattern P'. As an optimal solution fitting the pattern (if it exists) must add one of these elements into S we try all the possibilities to do so. The

element is added permanently to P and the algorithm recourses on it. This way a solution must be revealed if one exists.

As to the running time of the algorithm, the first greedy search runs in time $O(|\mathcal{C}|)$. As $|\mathcal{C}'_0| < 3k$ there are at most $\binom{3k}{k}$ "initial patterns". We are trying to fill each of them by the recursive algorithm, that first runs a greedy search in $O(|\mathcal{C}| \cdot k)$ time and then possibly runs some recursive calls, each corresponding to an element added to some set. Hence there are less than $2k$ of such calls, and for each of them the pattern is filled by one more element. Hence at latest on level $2k$ of the recursion it is completely full and it either constitutes a solution or some of its sets is not in $\mathcal{C}$ and the pattern can be discarded. Thus the overall running time of the algorithm is $O((3k)^k \cdot (2k)^{2k} \cdot |\mathcal{C}|)$.

6.6 Using the Theory of Minors, Bidimensionality

The theory of minors, developed mainly by Robertson and Seymour in their famous Graph Minors series (started by [RS83]), is nowadays one of the most important parts of the whole graph theory. It is no surprise, that it can be also used in parameterized algorithmics. For the basic definitions related to the minor theory refer to Section 2.2.

The following two essential results of Robertson and Seymour are of special interest also for parameterized algorithmics:

Theorem 6.10 (Robertson and Seymour [RS04])**.** *Any class of graphs has finitely many minimal elements with respect to the minor relation. In particular, if $\mathcal{C}$ is a minor closed class of graphs, then there is a finite set $\mathcal{F}(\mathcal{C})$ of* obstructions *such that $G \in \mathcal{C}$ if and only if there is no $H \in \mathcal{F}(\mathcal{C})$ such that H is a minor of G.*

Theorem 6.11 (Robertson and Seymour [RS95]; shorter proof published recently in [KW10])**.** *There is an algorithm that decides whether H is a minor of G in time $O(f(H) \cdot |V(G)|^3)$, for some function f.*

Putting Theorems 6.10 and 6.11 together we obtain the following corollary:

Corollary 6.12. *Any minor closed graph class $\mathcal{C}$ can be recognized in a cubic time.*

Proof. Due to Theorem 6.10, graph G is in $\mathcal{C}$ if and only if there is no minor H of G in $\mathcal{F}(\mathcal{C})$. By Theorem 6.11, this can be tested in time $O\left(\left(\sum_{H \in \mathcal{F}(\mathcal{C})} f(H)\right) \cdot |V(G)|^3\right)$. It remains to note that $\sum_{H \in \mathcal{F}(\mathcal{C})} f(H)$ is a finite constant since $\mathcal{F}(\mathcal{C})$ is finite. □

It is very easy to use Corollary 6.12 in parameterized complexity. It is enough to show that a set of yes-instances with a particular value of the parameter is

minor closed. The usage in parameterized complexity is very easy, it is only necessary to show that the class of problem instance with bounded parameter value is minor closed.

Example 6.13. Class of graphs having a vertex cover of size at most k is minor closed — hence it can be decided in $O(f(k) \cdot n^3)$-time whether an n-vertex graph G has a vertex cover of size at most k. To see the former it is enough to show that if G has an k-vertex cover C then for every $v \in V(G)$ and every $e \in E(G)$ the graphs $G \setminus \{v\}$, $G \setminus e$, and $G \cdot e$ have vertex cover of size at most k. For the first two $C \setminus \{v\}$ and C constitute such a cover, respectively. If $e = \{x, y\}$ is an edge of G, the vertex z to corresponds to the union of x and y in $V(G \cdot e)$ and $x \in C$ or $y \in C$ then $C \setminus \{x, y\} \cup \{z\}$ is such a cover for third case. Otherwise we can use simply C.

Remark 6.14. In Example 6.13 we have shown, that it can be decided in $O(f(k) \cdot n^3)$-time whether an n-vertex graph G has a vertex cover of size at most k, but we have shown no algorithm for that. We only know that there is a cubic algorithm for each fixed k, but the result gives no way to find the algorithm. Note that our definition of the class FPT requires the existence of a single algorithm for all values of the parameter. The class of problems for which there is a constant c and for each fixed value of the parameter there is an $O(n^c)$-algorithm is often called *non-uniform FPT*.

The above result can be simply generalized to a whole class of graph problems:

Definition 6.15. Let $\mathcal{C}$ be a class of graphs. A graph G is in the class $\mathcal{C} \oplus kv$ if and only if there is a set $S \subseteq V(G)$ of size at most k such that $G \setminus S$ is in $\mathcal{C}$.

Observation 6.16. *If $\mathcal{C}$ is a minor closed class, then so is $\mathcal{C} \oplus kv$.*

It is easy to see that edgeless graphs, forests and planar graphs form minor-closed families of graphs. As a corollary of this fact together with the above observation we get that not only Vertex Cover, but also Feedback Vertex Set or Planar Deletion are in non-uniform FPT (Here the question is whether it is possible to delete at most k vertices to obtain an edge-less graph, a forest, and a planar graph, respectively).

The most powerful usage of the theory of minors is in the combination with the graph width measures for the problems restricted to planar graphs, graphs with bounded genus or graphs excluding a fixed graph as a minor [DFHT05]. We present only the planar case. The other cases, although similar in the basic ideas, are more complicated in details.

Theorem 6.17 (Robertson, Seymour, and Thomas [RST94])**.** *Let $l \geq 1$ be an integer. Every planar graph of treewidth at least $6l - 4$ contains an $(l \times l)$-grid as a minor.*

If we forbid ourselves edge deletions and instead of each vertex deletion we contract the vertex to be deleted with some of its neighbors, we arrive at the following corollary.

Corollary 6.18. *Let $l \geq 1$ be an integer. Every connected planar graph of treewidth at least $6l - 4$ can be transformed into a partially triangulated $(l \times l)$-grid using only edge contractions.*

Using the above theorem we can derive a fairly general result for problems that are bidimensional. Intuitively, a parameterized graph problem is bidimensional if it is closed under (at least some) minor operations and if the parameter for grids grows linearly with the number of vertices of the grid. More formally:

Definition 6.19 (Demaine et al. [DFHT05])**.** A parameter P assigning an integer to each graph is *minor bidimensional* with density δ if

(1) $P(H) \leq P(G)$ whenever a graph H is a minor of a graph G, and

(2) for the $(l \times l)$-grid R, $P(R) = (\delta l)^2 + o(l^2)$.

The parameter P is called *contraction bidimensional* with density δ if

(1) contracting an edge in a graph G cannot increase $P(G)$,

(2) $P(C)$ is at most $P(G)$ whenever C is a connected component of G,

(3) for any partially triangulated $(l \times l)$-grid R, $P(R) \geq (\delta l)^2 + o(l^2)$, and

(4) δ is the smallest real number for which this inequality holds.

The parameter P is called *bidimensional* if it is either minor or contraction bidimensional.

It is usually very easy to prove that some parameter is bidimensional. We have already shown that the vertex cover number satisfies the condition (1) for the minor bidimensionality. If we notice, that an $(l \times l)$-grid contains l disjoint parallel paths each of length l and, thus, a matching of size $l \cdot \lfloor l/2 \rfloor$ we arrive at the following observation.

Observation 6.20. *Vertex cover number $vc(G)$ is minor bidimensional with density $1/\sqrt{2}$.*

Similarly one can see that the minimum size of a dominating set in a graph $ds(G)$ is closed under contractions of edges and taking connected components. Furthermore, a partially triangulated $(l \times l)$-grid contains $(l-2)^2$ inner vertices, among which no vertex can dominate more than 9 (including itself). Hence $ds(G)$ is contraction bidimensional.

The following theorem is a simple corollary of Theorem 6.17.

Theorem 6.21 (Demaine et al. [DFHT05]). *Let P be a bidimensional parameter. Then for any planar graph G, $tw(G) = O(\sqrt{P(G)})$.*

To use the above theorem we need to able to determine the treewidth of a graph very quickly, the algorithm of Bodlaender [Bod96] is not fast enough for our purpose. Fortunately, treewidth can be $\frac{3}{2}$ approximated in planar graphs.

Theorem 6.22 (Seymour and Thomas [ST94]; Gu and Tamaki [GT05]). *There is an algorithm that given a planar graph G outputs in a cubic time a tree decomposition of width w with the guarantee, that the treewidth $tw(G)$ is at least $2w/3$ (the algorithm actually computes an optimal so-called branch decomposition of the graph G, from which such a tree decomposition can be derived).*

The last important ingredient in a subexponential algorithm for computing a bidimensional parameter is a fast algorithm for the computation of the parameter on graphs of bounded treewidth. We need an algorithm with running time $2^{O(tw(G))} \cdot poly(n)$ or at least $2^{o(tw(G)^2)} \cdot poly(n)$. Such an algorithm for the vertex cover number can be derived from the algorithm for Independent Set in Section 6.2 and for (Annotated) Dominating Set an algorithm can be devised similarly.

Theorem 6.23 (Demaine et al. [DFHT05]). *If P is a bidimensional parameter and there is an $2^{O(tw(G))} \cdot poly(|V(G)|)$-time algorithm for its computation, then it is possible to decide in time $2^{O(\sqrt{k})} \cdot poly(|V(G)|)$ for a planar graph G whether it has $P(G) \leq k$.*

Proof. We prove the theorem for a parameter P that is minor bidimensional, the case of contraction bidimensionality is similar. We assume that $P(R) = (\delta l)^2 + o(l^2)$ for every $(l \times l)$-grid R. Hence there is some l_0 and $0 < \delta_0 \leq \delta$ such that for every $l \geq l_0$ and every $(l \times l)$-grid R we have $P(R) \geq (\delta_0 l)^2$.

We use the algorithm from Theorem 6.22 to obtain a tree-decomposition of width w. Due to Theorems 6.22 and 6.17 we know that the graph contains a grid of side at least $(2w/3)/6 = \frac{w}{9}$. If $\frac{1}{9}w \geq l_0$ and $(\frac{1}{9}\delta_0 \cdot w)^2 > k$ then the answer is no, as P is greater than k already on the $\frac{w}{9} \times \frac{w}{9}$-grid contained in G. Otherwise, we have a tree-decomposition of width at most $\max\{9l_0, 9\delta_0^{-1}\sqrt{k}\}$ and, hence, the problem can be solved in time $2^{O(\max\{9l_0, 9\delta_0^{-1}\sqrt{k}\})} \cdot poly(|V(G)|) = 2^{O(\sqrt{k})} \cdot poly(|V(G)|)$. □

The running times can be sometimes further improved, if the dynamic programming itself is designed more carefully using the properties of planar graphs. The results further generalize in a certain way to a classes of graphs of higher genus and classes excluding some fixed graph as a minor. See [DFHT05] for details of such generalizations.

Recent results [FLST10] also show that if the problem satisfies some further conditions, then one can even obtain a polynomial kernel for it, by replacing large

parts of the graph with low treewidth and small boundary by equivalent smaller parts.

Chapter 7

Intractability

As with the classical complexity, there is no way known to show unconditionally the non-existence of an fpt-algorithm for a certain problem (such a result would imply P≠NP). Instead we use to show that the fixed-parameter tractability of the problem considered would mean the same result for a wide class of other problems. For that purpose we first need a notion of parameterized reduction.

7.1 Reductions, Classes

Definition 7.1. A *parameterized reduction* (fpt-reduction) from a parameterized problem $\mathcal{P}$ to a parameterized problem $\mathcal{Q}$ is an algorithm that on an instance[1] $(x, k) \in \Sigma^* \times \mathbb{N}$ of $\mathcal{P}$ produces in time $f(k) \cdot |x|^{O(1)}$ an instance $(x', k') \in \Sigma^* \times \mathbb{N}$ of $\mathcal{Q}$ such that

- $(x, k) \in \mathcal{P}$ if and only if $(x', k') \in \mathcal{Q}$, and
- $k' \leq g(k)$,

where the functions f and g depend only on k. A parameterized problem $\mathcal{P}$ is *fpt-reducible* to a parameterized problem $\mathcal{Q}$ if there is a parameterized reduction from $\mathcal{P}$ to $\mathcal{Q}$.

Remark 7.2. We use the term reduction for both the classical polynomial time many:one reductions and the parameterized ones. The meaning should be clear from the context.

The essential property of parameterized reductions is that whenever $\mathcal{P}$ reduces to $\mathcal{Q}$ (by an fpt-reduction) and $\mathcal{Q}$ is in FPT, then $\mathcal{P}$ is FPT as well. Note that the second condition of Definition 7.1 is necessary for that purpose. It also worth noticing, that kernelization (Definition 5.1) and polynomial parameter transformation (Definition 5.9) are both special cases of parameterized reduction.

[1] We give the definition of the parameterized reduction only for the case when parameter is a single integer, but in the sense of Remark 3.5 it generalizes also to all other cases.

Unlike the classical complexity, in parameterized complexity the classes of intractability are primarily determined by their canonical complete problem, not by a model of computation. To this end, we first need the following classes of Boolean formulae.

Definition 7.3. A Boolean formula φ is 1-*normalized* if it is in the form of a conjunction of disjunctions of two literals (it is an instance of 2-SAT).
For $t \in \mathbb{N}, t > 1$, we say that a formula φ is t-*normalized* if it is in the form of a conjunction of disjunctions of conjunctions of.... of literals, where the conjunctions and the disjunctions alternate $t-1$ times (a formula in conjunctive normal form is 2-normalized).
A formula is called *monotone*, if it contains no negations and *antimonotone* if every literal in it is a negation of a variable.
A *weight* of a Boolean assignment $a : \{x_1, \ldots, x_n\} \to \{true, false\}$ is the number of variables set to *true*.

Fundamental problems of parameterized intractability are the following:

Weighted t-Normalized Satisfiability
Input: A t-normalized Boolean formula φ and $k \in N$.
Question: Is there a satisfying assignment for φ of weight exactly k?

Weighted Satisfiability
Input: A Boolean formula φ and $k \in N$.
Question: Is there a satisfying assignment for φ of weight exactly k?

Weighted Circuit Satisfiability
Input: A Boolean decision circuit C and $k \in N$.
Question: Is there a satisfying assignment for C of weight exactly k?

In all cases we use the weight of the sought solution k as a parameter. We will further mention Weighted Monotone t-Normalized Satisfiability and Weighted Antimonotone t-Normalized Satisfiability, the definitions of these problems should be clear.

Now we are ready to introduce the basic classes of parameterized intractability:

Definition 7.4. For every $t \in \mathbb{N}$ the class W[t] consists of all parameterized problems that are fpt-reducible to Weighted t-Normalized Satisfiability.
The classes of parameterized problems reducible to Weighted Satisfiability and Weighted Circuit Satisfiability are called W[Sat] and W[P], respectively.
We say that a parameterized problem $\mathcal{P}$ is W[t]-hard if every problem in W[t] can be FPT-reduced to $\mathcal{P}$ and W[t]-complete if it is W[t]-hard and in W[t]. Similarly for W[Sat] and W[P].

Immediately from the definitions one can get the following hierarchy of the parameterized complexity classes.

$$\text{FPT} \subseteq \text{W[1]} \subseteq \text{W[2]} \subseteq \ldots \subseteq \text{W[t]} \subseteq \ldots \subseteq \text{W[Sat]} \subseteq \text{W[P]} \subseteq \text{XP}$$

The reduction of an FPT problem to Weighted 1-Normalized Satisfiability can be done by first solving the problem (by its fpt-algorithm) and outputting a constant-size instance with the same answer. For the last inequality it suffices to use a trivial algorithm for Weighted Circuit Satisfiability that tries all weight k assignments and checks whether any of them is satisfying.

All of the above inequalities are supposed to be strict but so far we only know that FPT $\subsetneq$ XP [DF98]. It is also interesting that it is not known, whether an eventual collapse of two classes would propagate, either upwards or downwards.

As the W-classes serve to show that some problem is presumably not in FPT, the following class can be used to show that some problem is not even in the class XP.

Definition 7.5. A parameterized problem $\mathcal{P} \subseteq \Sigma^* \times \Sigma^*$ is para-NP-complete if there is a string $k_0 \in \Sigma^*$ such that the k_0-*th slice* of $\mathcal{P}$, that is the set $\{(x, k_0) \mid (x, k_0 \in \mathcal{P}\}$, is NP-complete.

A classical example of a para-NP-complete problem is Graph Coloring parameterized by the number of colors to be used, as it is known to be NP-complete even for 3 colors [GJ79].

It is not hard to see, that if there is a para-NP-complete problem in XP, then P=NP.

7.2 Monotone/Antimonotone Collapse

The following well known theorem gives an overview what happens if we require all literals of the formula to be positive or all of them to be negative.

Theorem 7.6 (Downey, Fellows [DF95a, DF95b]; Monotone/Antimonotone Collapse)**.** *Let $t \in \mathbb{N}$. Then* Weighted Monotone t-Normalized Satisfiability *is*

- *W[t]-complete if t is even,*
- *W[$t-1$]-complete if t is odd and $t > 1$, and*
- *FPT for $t = 1$.*

Conversely Weighted Antimonotone t-Normalized Satisfiability *is*

- *W[$t-1$]-complete if t is even, and*
- *W[t]-complete if t is odd.*

It can be easily seen that every instance of Weighted Antimonotone 1-Normalized Satisfiability can be viewed as an instance of k-Independent Set (or k-Clique equivalently) and vice versa and, hence, k-Clique is W[1]-complete. It is only slightly harder to show, that Dominating Set (also with standard parameterization) is W[2]-complete. Hence these two problems form a graph problem basis for W[1] and W[2], the two complexity classes vast majority of intractable natural problems fall in. There are also some quite natural problems known to be W[t] for all $t \in \mathbb{N}$, W[Sat] or W[P] complete, but the other classes are mainly of theoretical interest.

To give a flavor of a parameterized reduction, we prove the theorem for Weighted Antimonotone 1-Normalized Satisfiability, which implies that Clique is W[1]-complete. Obviously it suffices to show the hardness.

Theorem 7.7 (Downey, Fellows [DF95b])**.** Weighted Antimonotone 1-Normalized Satisfiability *is W[1]-hard.*

The proof is a modification of the original proof from [DF95b], where it was stated in a much more general way.

Proof. We will reduce Weighted 1-Normalized Satisfiability, as expected. Let (φ, k) be an instance of this problem with variables $x_1, \dots x_n$. We will construct an equivalent instance ψ, k' of Weighted Antimonotone 1-Normalized Satisfiability. For simplicity we assume $2 \leq k \leq n$.

The variables of ψ form k blocks $A^l = \{a^l_1, \dots, a^l_n\}$ for $1 \leq l \leq k$ and $k-1$ blocks $B^l = \{b^l_{i,j} \mid 1 \leq i, j \leq n\}$ for $1 \leq l \leq k-1$. We set $k' = 2k-1$ as from each of the blocks one variable is to be *true*. True variables in A blocks represent the variables of φ set to *true* and the variables in B blocks represent gaps between them. In particular, if a^l_i is *true*, then the *true* variable in B^l must be $b^l_{i,j}$ for some j. Conversely if $b^l_{i,j}$ is *true*, then in A^{l+1} the variable a^{l+1}_{i+j} must be *true*.

These restrictions are enforced by clauses as follows:

- There is at most one variable *true* in each block: for every $1 \leq l \leq k$ and all $1 \leq i, j, i', j' \leq n$ add to ψ the clause $(\neg a^l_i \vee \neg a^l_{i'})$ if $i \neq i'$ and the clause $(\neg b^l_{i,j} \vee \neg b^l_{i',j'})$ if $l \leq k-1$ and $(i,j) \neq (i',j')$.
- The variable selected in A^l enforces the selection in B^l: for every $1 \leq l \leq k-1$ and every $1 \leq i, i', j \leq n$ add the clause $(\neg a^l_i \vee \neg b^l_{i',j})$ if $i' \neq i$
- The variable selected in B^l enforces the selection in A^{l+1}: for every $1 \leq l \leq k-1$ and every $1 \leq i, i', j \leq n$ add the clause $(\neg b^l_{i,j} \vee \neg a^{l+1}_{i'})$ if $i' \neq i+j$

Since the construction can only handle gaps of positive size, we know that if both a^l_i and $a^{l+1}_{i'}$ are set to *true*, then $i < i'$.

The most important thing to realize is that the fact that some variable of φ is set to *false* is represented by a certain variable set to *true* in our construction. Namely setting x_r to *false* can be represented by that

- some of the variables $a^1_{r+1}, \ldots, a^1_n$ is set to *true*, or
- some of the variables $b^l_{i,j}$ with $i < r < i + j$ and $1 \leq l \leq k-1$ is set to *true*, or
- some of the variables $a^k_1, \ldots, a^k_{r-1}$ is set to *true*.

Denote the set of the above mentioned variables by S_r.

Now consider a clause C of φ. If (for some $1 \leq p, r \leq n$) the clause C is of the form

- $(x_p \vee x_r)$ then we add into ψ the clauses $(\neg y \vee \neg z)$ for every $y \in S_p$ and $z \in S_r$
- $(\neg x_p \vee x_r)$ (or $(x_r \vee \neg x_p)$) then we add into ψ the clauses $(\neg a^l_p \vee \neg y)$ into ψ for every $1 \leq l \leq k$ and $y \in S_r$
- $(\neg x_p \vee \neg x_r)$ then we add into ψ the clauses $(\neg a^l_p \vee \neg a^{l'}_r)$ for every $1 \leq l, l' \leq k$.

We claim that if the clause C is not satisfied in the assignment examined, then at least one of the added clauses is not satisfied as well. For the first case, this holds since if neither x_p nor x_r is set to *true*, then at least one variable $y \in S_p$ is set to *true* and at least one variable $z \in S_r$ is set to *true* and $(\neg y \vee \neg z)$ is not satisfied. Conversely, if all the newly added clauses are satisfied, then either no variable of S_p or no variable of S_r is set to *true* and thus either x_p or x_r is set to *true* and C is also satisfied. For the other cases it can be seen similarly.

Hence, an assignment setting exactly the variables $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$ to *true*, where $i_1 \leq i_2 \leq \cdots \leq i_k$, satisfies φ if and only if the assignment setting exactly variables $a^l_{i_l}$, $1 \leq l \leq k$ and $b^l_{i_l, i_{l+1} - i_l}$, $1 \leq l \leq k-1$ to *true* is satisfying for ψ. This finishes the reduction, as the instance (ψ, k') can be clearly constructed in polynomial time. □

7.3 Characterization by Computational Models

Although we said that the parameterized hardness classes are not defined by a computational model, there is actually a way to characterize the classes in terms of Turing Machine computation [CF03]. We only show that for two most important classes W[1] and W[2]. For that purpose let us first formally define a Nondeterministic Turing Machine.

Definition 7.8. A *Nondeterministic Turing Machine* is a sextuple $(\Sigma, t, Q, s, A, \delta)$, where Σ is an alphabet, t is the number of tapes, Q is a set of internal states, s is the initial state, $A \subseteq Q$ is the set of accepting states and $\delta \subseteq (Q \times \Sigma^t \times Q \times \Sigma^t \times \{-1, 0, 1\}^t)$ is the set of transitions. A quintuple $(p, (r_1, \ldots, r_t), q, (w_1, \ldots, w_t), (m_1, \ldots, m_t))$ being in δ means that if the machine is in the state p reading the symbol r_i on a

tape i (for every $1 \leq i \leq t$) it can proceed to the state q, writing the symbol w_i on the tape i and moving the head on this tape by m_i (for every i).
Note that not only the set of transitions can contain several possible transitions for one particular state and a t-tuple of symbols read, but it is also not required to be total, that is it does not have to contain a transition for each combination of state and t symbols read. The maximum number of possible transitions from a particular state when particular symbols are read is called the *amount of nondeterminism*.

It was proven in [CI97] that the following natural parameterized analogue of the Halting Problem is W[1]-complete:

Short Nondeterministic Turing Machine Computation
Input: A single-tape nondeterministic Turing machine $M = (\Sigma, 1, Q, s, A, \delta)$; a positive integer k.
Question: Is there a computation of M (on the empty tape) that reaches an accepting state in at most k steps?
Parameter: The allowed length of a computation k.

Note that for the result it is crucial that no bound is given for the size of the alphabet, the number of states, nor to the amount of nondeterminism. The problem is FPT for any of the parameterizations obtained by combining the number of steps k with any of the aspects mentioned above [CI97].

Similarly, the following problem is complete for W[2] as shown in [Ces03]:

Short Multi-Tape Nondeterministic Turing Machine Computation
Input: A nondeterministic Turing machine $M = (\Sigma, t, Q, s, A, \delta)$; a positive integer k.
Question: Is there a computation of M on the empty input that reaches an accepting state in at most k steps?
Parameter: The allowed length of a computation k.

The result heavily depends on having unlimited number of tapes. Parameterized by k and t, the problem becomes equivalent to the previous one, that is W[1]-complete [CI97]. A characterization of other classes by means of short computations of alternating Turing Machines can be found in [CF03].

Although Turing Machine is a standard tool in complexity considerations, RAM (Random Access Machine) model seems to be used more often when talking about algorithms. We introduce the concept of non-determinism into this model in a slightly unnatural way as proposed by Chen et al. [CFG03].

In a nondeterministic random access machine (NRAM) model a single nondeterministic instruction "GUESS" is added to the standard deterministic random access machine (RAM) model. The semantics of this instruction is: *Guess a natural number less than or equal to the number stored in the accumulator and store it in the accumulator.* Acceptance of an input by an NRAM is defined as usually for nondeterministic machines, that is, the program accepts the particular

input if there is a computation on it that ends by an execution of the ACCEPT instruction. The steps of the computation of an NRAM that execute the GUESS instruction are called *nondeterministic steps*.

To stay within the class W[1], the following restrictions should by put on a program for NRAM.

Definition 7.9 (Chen, Flum, and Grohe [CFG03])**.** An NRAM program $\mathbb{P}$ is *tail-nondeterministic k-restricted* if there are computable functions f and g and a polynomial p such that on every run with input $(x, k) \in \Sigma^* \times \Sigma^*$ the program $\mathbb{P}$

- performs at most $f(k) \cdot p(|x|)$ steps;
- uses at most the first $f(k) \cdot p(|x|)$ registers;
- contains numbers $\leq f(k) \cdot p(|x|)$ in any register at any time;

and all nondeterministic steps are among the last $g(k)$ steps of the computation.

The following characterization due to Chen, Flum, and Grohe [CFG03] seems to be easier to apply then developing a reduction to a single-tape Turing Machine:

Theorem 7.10 (Chen, Flum, and Grohe [CFG03])**.** *A parameterized problem $\mathcal{P}$ is in W[1] if and only if there is a tail-nondeterministic k-restricted NRAM program deciding $\mathcal{P}$.*

The model can be further enriched by a nondeterministic instruction FORALL, to obtain alternating RAMs. By restricting the number of times the machine can switch from using the GUESS instruction to the FORALL instruction and vice versa, one can characterize the other classes of the W-hierarchy [CF03].

7.4 Multicolored Problems

In most reductions that directly reduce Weighted t-Normalized Satisfiability it is necessary to somehow represent, that some variable was not set to *true* in the solution considered. This is usually done by representing the gap between two neighboring (in a certain order) variables set to *true*. Such reductions are sometimes called *gap reductions*.

Another approach is to develop a problem in which the objects for a size-k solution are picked from k groups, each object from a different group. Then the case that an object is not picked into a solution is represented by picking another object from the same group. The groups are usually referred to as colors and we speak about multicolored problems. The result from Section 6.3 suggests, that the multicolored problem is usually no easier than the original one.

The problem most often used for hardness reductions is probably Multicolored Clique. It can be found in older literature under the name Partitioned Clique, but it is nowadays more known under the other name.

Multicolored Clique (MCC)
Input: A graph $G = (V, E)$, positive integer $k \in \mathbb{N}$ and a proper k-coloring $c : V \to \{1, \ldots, k\}$ of G.
Question: Is there a multicolored clique in G, that is, a clique taking exactly one vertex of each color?
Parameter: The number of colors k.

Note that any clique of size k in a graph properly colored by k colors must take exactly one vertex of each color, and there is no clique of size more than k.

The following easy theorem was in fact proved in [Pie03], and recently rediscovered by [FHRV09].

Theorem 7.11. Multicolored Clique *is W[1]-complete.*

Proof. We provide an almost trivial reduction from and to Clique which we have shown to be W[1]-complete in Section 7.2. Omitting the coloring from an instance of MCC one obtain an equivalent instance of Clique, which implies that MCC is in W[1]. To show the W[1]-hardness we let $G = (V, E), k$ be an instance of Clique. Consider an instance of MCC formed by $G' = (V', E')$, where $V' = V \times \{1, \ldots, k\}$ and $E' = \{\{(u, i), (v, j)\} \mid i \neq j \wedge \{u, v\} \in E\}$, the number k, and the coloring $c : V' \to \{1, \ldots, k\}$ assigning to each vertex (v, i) its second coordinate i.

If $\{v_1, \ldots, v_k\}$ is a k-Clique in G, then $\{(v_1, 1), \ldots, (v_k, k)\}$ is a multicolored clique in G'. On the other hand if $\{(v_1, 1), \ldots, (v_k, k)\}$ is a multicolored clique in G' then for every $i \neq j$ the set $\{v_i, v_j\}$ is an edge of G and, thus, $\{v_1, \ldots, v_k\}$. Hence (G, k) is a yes-instance of Clique if and only if (G', k, c) is a yes-instance of MCC, which finishes the proof, as G' can be constructed in polynomial time and the parameter is preserved. □

Note that the constructed instance of MCC has the same number of vertices of each color and also the number of edges with endpoints colored by a particular pair of colors is the same for each pair of colors selected. This property is also often used in the reductions.

We also mention, that in many reduction from MCC, not only there are gadgets for each color, that represent a selection of a vertex of this color, but there are often gadgets for each pair of different colors representing the selection of an edge between the vertices of the particular color. This simplifies the subsequent check, whether the select objects form a clique. We just check, whether the selected edges are incident with the selected vertices, which is often much simpler than checking whether the selected vertices are adjacent. This idea called *edge representation strategy* was first introduced by Fellows et al. in [FHRV09].

A very simple example of a reduction starting from Multicolored Clique (MCC) is to show that List Coloring is W[1]-hard parameterized by the vertex cover number. This was first observed in [FFL$^+$07] and so far constitutes one of a few, if not the only example of a problem hard with respect to the vertex cover number. We also mention that the paper is also the first one to show that some problem is W[1]-hard with respect to the treewidth for which it also uses a reduction from MCC together with the edge representation strategy.

In List Coloring we are given a graph G, a set of colors B and a mapping $L : V(G) \to \mathcal{P}(B)$ assigning to each vertex its list of available colors. The question is whether there is a proper coloring $c : V(G) \to B$ of G respecting the lists. This means that for every $v \in V(G)$ we have $c(v) \in L(v)$.

Theorem 7.12. List Coloring *is W[1]-hard parameterized by the vertex cover number.*

Proof. For the reduction, assume that we are given an instance of MCC with a graph G having n vertices of each color out of $\{1, \ldots, k\}$. The vertices of color i are denoted $v_{i,1}, \ldots, v_{i,n}$. We construct an instance of List Coloring formed by a graph G', a set of colors $B = V(G)$ and a mapping $L : V(G') \to B$. We start by introducing k vertices $a_1, \ldots, a_k$; the vertex a_i will have a list $L(a_i) = \{v_{i,1}, \ldots, v_{i,n}\}$. The colors chosen for these vertices should represent the selected vertices of particular colors.

To ensure that the selected vertices form a clique, we do the following. For each i and i', where $1 \leq i < i' \leq k$, if for some $1 \leq j, j' \leq n$ the vertices $v_{i,j}$ and $v_{i',j'}$ are not connected by an edge in G, we add a new vertex into G' which will be connected to a_i and $a_{i'}$ and will have a list $\{v_{i,j}, v_{i',j'}\}$. Hence, if the vertex a_i was assigned the color $v_{i,j}$ and the vertex $a_{i'}$ the color $v_{i',j'}$, then there would be no chance to color this new vertex. Otherwise there is always at least one color left.

It is easy to see, that this way $x_1, \ldots, x_k$ is a multicolored clique in G if and only if the partial coloring $f : \{a_1, \ldots a_k\} \to V(G)$, that assigns the color x_i to the vertex a_i for every i, can be extended to a proper list coloring of G' respecting the lists L. As the construction can be clearly done in polynomial time, it remains to note that the set $\{a_1, \ldots a_k\}$ forms a vertex cover of size k for the graph G'. Hence, the parameter of the new instance equals the parameter of the original instance and the reduction is indeed a parameterized one. □

7.5 Connections to the Exponential Time Hypothesis

Although it is hard to believe that there could an $f(k)n^{O(1)}$-time algorithm for Short Nondeterministic Turing Machine Computation (which would

be implied by W[1]=FPT) still for some people less familiar with parameterized complexity the following hypothesis is more plausible than that W[1]≠FPT.

Hypothesis 7.13 (Exponential Time Hypothesis (ETH))**.** *There is no algorithm that solves n-variable 3SAT in time* $2^{o(n)}$*.*

In fact, ETH is a stronger hypothesis — it implies W[1]≠FPT, while it is not known whether W[1]≠FPT implies ETH. In particular Abrahamson et al. [ADF95] have shown the following:

Theorem 7.14 (Abrahamson, Downey, and Fellows [ADF95])**.** *If there is an* $f(k) \cdot n^{O(1)}$ *time algorithm for* k-Clique*, then ETH fails.*

This result was later strengthened so that it also excludes algorithms with the exponent of the polynomial running time growing slower than linear in the parameter:

Theorem 7.15 (Chen, Huang, Kanj, and Xia [CHKX04])**.** *If there is an* $f(k) \cdot n^{o(k)}$ *time algorithm for* k-Clique*, then ETH fails.*

The result stated for Clique can be easily translated to other problems. Namely, if there was a parameterized reduction transforming an instance (G, k) of k-Clique to an instance (x', k') of a problem $\mathcal{P}$ with $k' = O(k^c)$ then an algorithm for $\mathcal{P}$ with running time $f(k)n^{o(k^{1/c})}$ would imply an $f'(k)n^{o(k)}$ algorithm for k-Clique and, thus, ETH would fail. As most of the parameterized reductions known have either $k' = O(k)$ or $k' = O(k^2)$ this provides a good lower bound for many problems. Note that this way the results also translate to many problems parameterized by a structural or other parameters.

Assuming ETH it is also possible to prove, that for certain problems the dependence of the exponent of the exponential part of the running time on the parameter is asymptotically optimal.

Theorem 7.16 (Cai and Juedes [CJ03])**.** *If* Vertex Cover *can be solved in time* $2^{o(k)} \cdot n^{O(1)}$*, then ETH fails.*

Similar results can be proved also for problems on planar graphs, but here the known (asymptotically optimal) algorithms are only exponential in the square root of the parameter; see Section 6.6 for such algorithms.

Theorem 7.17 (Cai and Juedes [CJ03])**.** *If* Vertex Cover*,* Independent Set*, or* Dominating Set *can be solved in time* $2^{o(\sqrt{k})} \cdot n^{O(1)}$ *for planar graphs, then ETH fails.*

The following is a further strengthening of ETH, which is definitely not so widely believed as ETH itself.

Hypothesis 7.18 (Strong Exponential Time Hypothesis (SETH))**.** *Let $\epsilon > 0$. There is no algorithm that solves n-variable SAT in time $(2 - \epsilon)^n$.*

Under this assumption, the lower bound for the running times of an algorithm for Dominating Set can be further improved as follows.

Theorem 7.19 (Patrascu and Williams [PW10])**.** *If for some $k \geq 3$ and $\epsilon > 0$, k-Dominating Set can be solved in $O(n^{k-\epsilon})$ time then SETH fails.*

This result can be again translated in a certain way to some W[2]-hard problems. By contrast, an $O(n^{0.793k})$ algorithm for k-Clique is known [NP85]. This suggests, that also some differences in the running times achievable for W[1]-complete and W[2]-complete problems are to be expected.

List of Considered Problems

d-Hitting Set
Input: A family $\mathcal{F}$ of sets, each with at most d elements, and $k \in \mathbb{N}$.
Question: Is there a set of at most k elements, that contains an element from (hits) every set in $\mathcal{F}$?
Parameterizations Considered: solution-size k
Considered on pages: 16, 19, 20, 23, 27

k-Leaf Out-Branching
Input: A directed graph and $k \in \mathbb{N}$.
Question: Does the directed graph contain a subgraph in which every vertex except for one has in-degree exactly 1 and k vertices has out-degree 0 (are leaves)?
Parameterizations Considered: number of leaves k
Considered on pages: 23, 24

k-Local Search for Traveling Salesperson
Input: A graph with positive weights on edges, a Hamiltonian cycle in it and $k \in \mathbb{N}$.
Question: Can we find a Hamiltonian cycle which uses at most k edges not used by the original cycle and its weight is smaller than the weight of the original one
Parameterizations Considered: locality k
Considered on pages: 13

k-Path
Input: A graph G and $k \in \mathbb{N}$.
Question: Does G contain a path of length k?
Parameterizations Considered: solution-size k
Considered on pages: 22, 32

Bounded-Degree Deletion
Input: A graph G, $d \in \mathbb{N}$ and $k \in \mathbb{N}$.
Question: Is there a set of at most k vertices such that its deletion turns G into a graph with maximum degree at most d?

Parameterizations Considered: solution-size k
Considered on pages: 23

Clique
Input: A graph and $k \in \mathbb{N}$.
Question: Does the graph have complete subgraph with at least k vertices?
Parameterizations Considered: solution-size k
Considered on pages: 44, 48, 50, 51

Conservative Coloring
Input: A graph which have all vertices except for one properly colored by k colors and $c \in \mathbb{N}$.
Question: a proper coloring of that graph with k colors which differ from the original one on at most c places?
Parameterizations Considered: conservativeness c
Considered on pages: 13

Dominating Set
Input: A graph G and $k \in \mathbb{N}$.
Question: Is there a set of at most k vertices, such that every vertex of G is either a part of it or has a neighbor in it?
Parameterizations Considered: solution-size k, treewidth $tw(G)$
Considered on pages: 24, 38, 39, 44, 50, 51

Feedback Vertex Set
Input: A graph G and $k \in \mathbb{N}$.
Question: Is there a set of at most k vertices such that its deletion turns G into a forest?
Parameterizations Considered: solution-size k
Considered on pages: 23, 37

Graph Coloring
Input: A graph G on n vertices and $k \in \mathbb{N}$.
Question: Can G be properly colored by at most k colors?
Parameterizations Considered: number of colors available k, dual parameterization $n - k$
Considered on pages: 20, 43

Independent Set
Input: A graph and $k \in \mathbb{N}$.
Question: Is there a subset of at least k vertices such that no two of the vertices are connected by an edge?
Parameterizations Considered: solution-size k, treewidth $tw(G)$

Considered on pages: 30, 32, 39, 44, 50

List Coloring
Input: A graph G, a set of colors B and a mapping $L : V(G) \to \mathcal{P}(B)$ assigning to each vertex its list of available colors
Question: Is there a proper coloring $c : V(G) \to B$ respecting the lists? This means that for every $v \in V(G)$ we have $c(v) \in L(v)$
Parameterizations Considered: vertex cover number $vc(G)$
Considered on pages: 48, 49

Max d-Sat
Input: A propositional formula φ in form of conjunction of clauses, each formed by exactly d literals and $k \in \mathbb{N}$.
Question: Is it possible to satisfy at least $m(1 - 2^{-d}) + k$ clauses?
Parameterizations Considered: param. above tight lower bounds k
Considered on pages: 12

Max Leaf
Input: A graph and $k \in \mathbb{N}$.
Question: Is there a spanning tree of the graph with at least k leaves?
Parameterizations Considered: solution-size k
Considered on pages: 25

Multicolored Clique (MCC)
Input: A graph $G = (V, E)$, positive integer $k \in \mathbb{N}$ and a proper k-coloring $c : V \to \{1, \ldots, k\}$ of G.
Question: Is there a multicolored clique in G, that is a clique taking exactly one vertex of each color?
Parameterizations Considered: number of colors k
Considered on pages: 47–49

Odd Cycle Transversal
Input: A graph G and $k \in \mathbb{N}$.
Question: Is there a set of at most k vertices such that its deletion turns G into a bipartite graph?
Parameterizations Considered: solution-size k
Considered on pages: 33

Packing 3-Sets
Input: A system $\mathcal{C}$ of three-element subsets of a finite set S and an integer $k \in \mathbb{N}$.
Question: Is there a subsystem $\mathcal{C}'$ of at least k mutually disjoint sets?
Parameterizations Considered: solution-size k
Considered on pages: 35

Planar Deletion
Input: A graph G and $k \in \mathbb{N}$.
Question: Is there a set of at most k vertices such that its deletion turns G into a planar graph?
Parameterizations Considered: solution-size k
Considered on pages: 23, 37

Short Multi-Tape Nondeterministic Turing Machine Computation
Input: A nondeterministic Turing machine $M = (\Sigma, t, Q, s, A, \Delta)$; a positive integer k.
Question: Is there a computation of M on empty input that reaches accepting state in at most k steps?
Parameterizations Considered: allowed length of a computation k, number of tapes
Considered on pages: 46

Short Nondeterministic Turing Machine Computation
Input: A single-tape nondeterministic Turing machine $M = (\Sigma, 1, Q, s, A, \Delta)$; a positive integer k.
Question: Is there a computation of M (on empty tape) that reaches the accepting state in at most k steps?
Parameterizations Considered: allowed length of a computation k, size of the alphabet $|\Sigma|$, number of states $|Q|$, amount of non-determinism, their combinations
Considered on pages: 46, 49

Steiner Tree
Input: A graph $G = (V, E)$ with integral weights on edges $w : E \to \mathbb{N}$, a set of terminals $T \subseteq V$ and an integer $p \in \mathbb{N}$.
Question: Is there a tree containing all the terminals of cost at most p?
Parameterizations Considered: maximum number of non-terminals in a solution (solution-size parameterization above tight lower bound) $p - |T| + 1$, number of terminals $|T|$
Considered on pages: 13, 28

Vertex Cover
Input: A graph G and $k \in \mathbb{N}$.
Question: Is there a set of at most k vertices, that contains at least one endpoint of each edge?
Parameterizations Considered: solution-size k
Considered on pages: 8, 10, 18, 19, 21, 27, 37, 50

Weighted t-Normalized Satisfiability
Input: A t-normalized Boolean formula φ and $k \in N$.
Question: Is there a satisfying assignment for φ of weight exactly k?
Parameterizations Considered: weight of the assignment k
Considered on pages: 42–44, 47

Weighted Antimonotone t-Normalized Satisfiability
Input: A t-normalized antimonotone Boolean formula φ and $k \in N$.
Question: Is there a satisfying assignment for φ of weight exactly k?
Parameterizations Considered: weight of the assignment k
Considered on pages: 42–44

Weighted Circuit Satisfiability
Input: A Boolean decision circuit C and $k \in N$.
Question: Is there a satisfying assignment for C of weight exactly k?
Parameterizations Considered: weight of the assignment k
Considered on pages: 42, 43

Weighted Monotone t-Normalized Satisfiability
Input: A t-normalized monotone Boolean formula φ and $k \in N$.
Question: Is there a satisfying assignment for φ of weight exactly k?
Parameterizations Considered: weight of the assignment k
Considered on pages: 42, 43

Weighted Satisfiability
Input: A Boolean formula φ and $k \in N$.
Question: Is there a satisfying assignment for φ of weight exactly k?
Parameterizations Considered: weight of the assignment k
Considered on pages: 42

Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. Cited on pages 3 and 4.

[ACP87] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic and Discrete Methods*, 8(2):277–284, 1987. Cited on page 14.

[ADF95] Karl R. Abrahamson, Rodney G. Downey, and Michael R. Fellows. Fixed-parameter tractability and completeness IV: On completeness for W[P] and PSPACE analogues. *Ann. Pure Appl. Logic*, 73(3):235–276, 1995. Cited on page 50.

[AEFM89] Karl R. Abrahamson, John A. Ellis, Michael R. Fellows, and Manuel E. Mata. On the complexity of fixed parameter problems (extended abstract). In *FOCS*, pages 210–215. IEEE, 1989. Cited on page 1.

[AGK+10] Noga Alon, Gregory Gutin, Eun Jung Kim, Stefan Szeider, and Anders Yeo. Solving MAX-r-SAT above a tight lower bound. In Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 511–517, 2010. Cited on page 13.

[AYZ95] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995. Cited on pages 32 and 33.

[BDFH09] Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Danny Hermelin. On problems without polynomial kernels. *Journal of Computer and System Sciences*, 75(8):423–434, 2009. Cited on page 22.

[BGN08] Nadja Betzler, Jiong Guo, and Rolf Niedermeier. Parameterized computational complexity of Dodgson and Young elections. In Gudmundsson, editor, *SWAT*, volume 5124 of *LNCS*, pages 402–413. Springer, 2008. Cited on page 13.

[BK10] Hans L. Bodlaender and Arie M.C.A. Koster. Treewidth computations I. Upper bounds. *Information and Computation*, 208(3):259 – 275, 2010. Cited on page 14.

[Bod93] Hans L. Bodlaender. On linear time minor tests with depth-first search. *J. Algorithms*, 14(1):1–23, 1993. Cited on page 25.

[Bod96] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996. Cited on pages 14, 32, and 39.

[Bod09] Hans L. Bodlaender. Kernelization: New upper and lower bound techniques. In *Proc. 4th IWPEC*, volume 5917 of *LNCS*, pages 17–37. Springer, 2009. Cited on page 21.

[BTY08] Hans L. Bodlaender, Stéphan Thomassé, and Anders Yeo. Analysis of data reduction: Transformations give evidence for non-existence of polynomial kernels. Technical report, Department of Information and Computing Sciences Utrecht University, Utrecht, The Netherlands, 2008. Cited on page 23.

[Ces03] Marco Cesati. The Turing way to parameterized complexity. *J. Comput. Syst. Sci.*, 67(4):654–685, 2003. Cited on page 46.

[CF03] Yijia Chen and Jörg Flum. Machine characterization of the classes of the W-hierarchy. In Baaz and Makowsky, editors, *CSL*, volume 2803 of *LNCS*, pages 114–127. Springer, 2003. Cited on pages 45, 46, and 47.

[CFG03] Yijia Chen, Jörg Flum, and Martin Grohe. Bounded nondeterminism and alternation in parameterized complexity theory. In *IEEE Conference on Computational Complexity*, pages 13–29. IEEE Computer Society, 2003. Cited on pages 46 and 47.

[CFJ04] Benny Chor, Mike Fellows, and David W. Juedes. Linear kernels in linear time, or how to save k colors in $O(n^2)$ steps. In Hromkovič, Nagl, and Westfechtel, editors, *Graph-Theoretic Concepts in Computer Science, 30th International Workshop,WG 2004*, volume 3353 of *LNCS*, pages 257–269, 2004. Cited on page 20.

[CHKX04] Jianer Chen, Xiuzhen Huang, Iyad A. Kanj, and Ge Xia. Linear fpt reductions and computational lower bounds. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 212–221, New York, NY, USA, 2004. ACM. Cited on page 50.

[CI97] Marco Cesati and Miriam Di Ianni. Computation models for parameterized complexity. *Math. Log. Q.*, 43:179–202, 1997. Cited on page 46.

[CJ03] Liming Cai and David W. Juedes. On the existence of subexponential parameterized algorithms. *J. Comput. Syst. Sci.*, 67(4):789–807, 2003. Cited on page 50.

[Cou92] Bruno Courcelle. The monadic second-order logic of graphs III: Tree-decompositions, minor and complexity issues. *ITA*, 26:257–286, 1992. Cited on page 31.

[CR72] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 73–80, New York, NY, USA, 1972. ACM. Cited on page 4.

[DF92a] Rodney G. Downey and Michael R. Fellows. Fixed-parameter intractability. In *Structure in Complexity Theory Conference*, pages 36–49, 1992. Cited on page 1.

[DF92b] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness. *Congressus Numerantium*, 87:161–178, 1992. Cited on page 1.

[DF95a] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal on Computing*, 24(4):873–921, 1995. Cited on page 43.

[DF95b] 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. Cited on pages 43 and 44.

[DF98] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Springer, 1998. Cited on pages 1, 8, 10, 18, 24, and 43.

[DFHT05] Erik D. Demaine, Fedor V. Fomin, Mohammadtaghi Hajiaghayi, and Dimitrios M. Thilikos. Subexponential parameterized algorithms on bounded-genus graphs and H-minor-free graphs. *J. ACM*, 52(6):866–893, 2005. Cited on pages 37, 38, and 39.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390. Cited on page 29.

[DLS09] Michael Dom, Daniel Lokshtanov, and Saket Saurabh. Incompressibility through colors and IDs. In *ICALP '09: Proceedings of the 36th International Colloquium on Automata, Languages and Programming*, pages 378–389, Berlin, Heidelberg, 2009. Springer-Verlag. Cited on pages 22 and 23.

[DvM10] Holger Dell and Dieter van Melkebeek. Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. In Schulman, editor, *STOC*, pages 251–260. ACM, 2010. Cited on pages 23 and 24.

[DW72] Stuart E. Dreyfus and Robert A. Wagner. The Steiner problem in graphs. *Networks*, 1:195–207, 1972. Cited on page 28.

[DYW+07] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *Proc. 23rd ICDE*, pages 836–845. IEEE, 2007. Cited on page 28.

[Fel09] Michael R. Fellows. Towards fully multivariate algorithmics: Some new results and directions in parameter ecology. In *Proc. 20th IWOCA*, volume 5874 of *LNCS*, pages 2–10. Springer, 2009. Cited on pages 2 and 17.

[FF56] Lester R. Ford and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. Cited on page 35.

[FFL+07] Michael R. Fellows, Fedor V. Fomin, Daniel Lokshtanov, Frances A. Rosamond, Saket Saurabh, Stefan Szeider, and Carsten Thomassen. On the complexity of some colorful problems parameterized by treewidth. In Dress, Xu, and Zhu, editors, *COCOA*, volume 4616 of *LNCS*, pages 366–377. Springer, 2007. Cited on page 49.

[FFL+09] Henning Fernau, Fedor V. Fomin, Daniel Lokshtanov, Daniel Raible, Saket Saurabh, and Yngve Villanger. Kernel(s) for problems with no kernel: On out-trees with many leaves. In Albers and Marion, editors, *STACS*, volume 3 of *LIPIcs*, pages 421–432. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2009. Cited on pages 23 and 24.

[FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., 2006. Cited on pages 2, 8, 9, 10, and 20.

[FHRV09] Michael R. Fellows, Danny Hermelin, Frances A. Rosamond, and Stéphane Vialette. On the parameterized complexity of multiple-interval graph problems. *Theoretical Computer Science*, 410(1):53–61, 2009. Cited on page 48.

[FLST10] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Dimitrios M. Thilikos. Bidimensionality and kernels. In Charikar, editor, *SODA*, pages 503–510. SIAM, 2010. Cited on page 39.

[FS08] Lance Fortnow and Rahul Santhanam. Infeasibility of instance compression and succinct PCPs for NP. In Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing*, pages 133–142. ACM, 2008. Cited on page 22.

[FT87] Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, 1987. Cited on page 30.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. Cited on page 43.

[GMN09] Jiong Guo, Hannes Moser, and Rolf Niedermeier. Iterative compression for exactly solving NP-hard minimization problems. In Lerner, Wagner, and Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, pages 65–80. Springer, 2009. Cited on page 35.

[GN07] Jiong Guo and Rolf Niedermeier. Invitation to data reduction and problem kernelization. *ACM SIGACT News*, 38(1):31–45, 2007. Cited on page 21.

[GT05] Qian-Ping Gu and Hisao Tamaki. Optimal branch-decomposition of planar graphs in $O(n^3)$ time. In Caires, Italiano, Monteiro, Palamidessi, and Yung, editors, *ICALP*, volume 3580 of *LNCS*, pages 373–384. Springer, 2005. Cited on page 39.

[HN10] Sepp Hartung and Rolf Niedermeier. Incremental list coloring of graphs, parameterized by conservation. In Kratochvíl, Li, Fiala, and Kolman, editors, *Theory and Applications of Models of Computation, 7th Annual Conference, TAMC 2010*, volume 6108 of *LNCS*, pages 258–270. Springer, 2010. Cited on page 13.

[JZC04] Weijia Jia, Chuanlin Zhang, and Jianer Chen. An efficient parameterized algorithm for m-set packing. *J. Algorithms*, 50(1):106–117, 2004. Cited on page 35.

[Kna10] Christian Knauer. The complexity of geometric problems in high dimension. In Kratochvíl, Li, Fiala, and Kolman, editors, *Theory and Applications of Models of Computation, 7th Annual Conference, TAMC 2010*, volume 6108 of *LNCS*, pages 40–49. Springer, 2010. Cited on page 13.

[KT05] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. Cited on page 27.

[KW91] Daniel J. Kleitman and Douglas B. West. Spanning trees with many leaves. *SIAM J. Discret. Math.*, 4(1):99–106, 1991. Cited on page 15.

[KW10] Ken-ichi Kawarabayashi and Paul Wollan. A shorter proof of the graph minor algorithm: the unique linkage theorem. In Schulman, editor, *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010*, pages 687–694. ACM, 2010. Cited on page 36.

[LSS09] Daniel Lokshtanov, Saket Saurabh, and Somnath Sikdar. Simpler parameterized algorithm for OCT. In Fiala, Kratochvíl, and Miller, editors, *IWOCA*, volume 5874 of *LNCS*, pages 380–384. Springer, 2009. Cited on page 33.

[Mar08] Dániel Marx. Searching the k-change neighborhood for TSP is W[1]-hard. *Oper. Res. Lett.*, 36(1):31–36, 2008. Cited on page 13.

[MN09] Jiří Matoušek and Jaroslav Nešetřil. *Invitation to Discrete Mathematics, Second Edition*. Oxford University Press, 2009. Cited on page 4.

[Moh99] Bojan Mohar. A linear time algorithm for embedding graphs in an arbitrary surface. *SIAM J. Discrete Math.*, 12(1):6–26, 1999. Cited on page 14.

[Nie06] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006. Cited on pages 1 and 8.

[Nie10] Rolf Niedermeier. Reflections on multivariate algorithmics and problem parameterization. In *Proc. 27th STACS*, volume 5 of *LIPIcs*, pages 17–32. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010. Cited on pages 2 and 17.

[NP85] Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985. Cited on page 51.

[NR03] Rolf Niedermeier and Peter Rossmanith. An efficient fixed-parameter algorithm for 3-Hitting Set. *J. Discrete Algorithms*, 1(1):89–102, 2003. Cited on page 28.

[Pie03] Krzysztof Pietrzak. On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. *Journal of Computer and System Sciences*, 67(4):757 – 771, 2003. Parameterized Computation and Complexity 2003. Cited on page 48.

[PW10] Mihai Patrascu and Ryan Williams. On the possibility of faster SAT algorithms. In Charikar, editor, *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 1065–1075, 2010. Cited on page 51.

[Ree92] Bruce A. Reed. Finding approximate separators and computing tree width quickly. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 221–228, New York, NY, USA, 1992. ACM. Cited on page 14.

[RS83] Neil Robertson and Paul D. Seymour. Graph minors. I. Excluding a forest. *J. Comb. Theory, Ser. B*, 35(1):39–61, 1983. Cited on page 36.

[RS84] Neil Robertson and Paul D. Seymour. Graph minors. III. Planar treewidth. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984. Cited on pages 6 and 14.

[RS95] Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *J. Comb. Theory, Ser. B*, 63(1):65–110, 1995. Cited on page 36.

[RS04] Neil Robertson and Paul D. Seymour. Graph minors. XX. Wagner's conjecture. *J. Comb. Theory, Ser. B*, 92(2):325–357, 2004. Cited on page 36.

[RST94] Neil Robertson, Paul D. Seymour, and Robin Thomas. Quickly excluding a planar graph. *J. Comb. Theory, Ser. B*, 62(2):323–348, 1994. Cited on page 37.

[RSV04] Bruce A. Reed, Kaleigh Smith, and Adrian Vetta. Finding odd cycle transversals. *Oper. Res. Lett.*, 32(4):299–301, 2004. Cited on page 33.

[ST94] Paul D. Seymour and Robin Thomas. Call routing and the ratcatcher. *Combinatorica*, 14(2):217–241, 1994. Cited on page 39.

[Wes96] Douglas B. West. *Introduction to graph theory.* Prentice Hall Inc., Upper Saddle River, NJ, 1996. Cited on page 4.