INFEASIBILITY OF POLYNOMIAL KERNELIZATION

A thesis submitted in partial fulfillment of
the requirements for the award of the degree of
Masters of Science

Neeldhara Misra
Junior Research Fellow
Institute of Mathematical Sciences

HOMI BHABHA NATIONAL INSTITUTE

Certificate

Certified that the work contained in the thesis entitled On the Infeasibility of Obtaining Polynomial Kernels, by Neeldhara Misra, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Venkatesh Raman
Theoretical Computer Science Group,
Institute of Mathematical Sciences, Chennai

Abstract

In approaching a problem that is deemed to be computationally hard, preprocessing steps are often employed as a first attempt at problem simplification. The notion of *problem kernelization* is a formal framework in which algorithms that simplify problems (instead of solving them) can be analyzed for efficiency. A kernelization algorithm performs data reduction with provable performance bounds. Many NP-complete problems are known to admit kernelization algorithms. However, there are also NP-complete problems for which we do not expect such algorithms (indeed, under reasonable complexity theoretic assumptions, it is known that such algorithms are not possible). This immediately gives us a finer classification of problems that are only understood as intractable in the classical context, in particular, this is one way to discriminate between NP-complete problems.

In the context of parameterized complexity, it is well known that kernelization corresponds exactly to efficient parameterized computation, or *fixed parameter tractability*. However, not all problems that belong here are "equally efficient", and a more thorough classification may be obtained by examining the efficiency of the kernelization procedures that are known for these problems. Here, a natural measure of efficiency would correspond to the kernel size. We survey methods for establishing that some problems cannot have polynomial–size kernels, under the assumption that $\mathsf{PH} \neq \Sigma_3^{\mathsf{P}}$. These methods involve devising "composition algorithms" which are demonstrated for many problems. We suggest a standardization under which the similarity of the compositional techniques used for different problems becomes evident.

We also illustrate a problem which, while unlikely to have a polynomial kernel, has multiple (π, to be precise) polynomial sized kernels.

We conclude with pointers to the many directions for future work that have opened up because of these studies.

Acknowledgements

I am indebted to my advisor, Prof. Venkatesh Raman. His guidance during the preparation of this thesis has been invaluable. His enthusiasm, for research and teaching alike, is contagious. I have been fortunate enough to be in the same classroom as him, and this has had more benefits than I can count!

I am grateful to all the faculty at the Department of Theoretical Computer Science. Everything (if anything) that I understand about Computer Science is due to their patient, creative, and inspiring teaching.

Thanks are due to Dr. Saket Saurabh, for suggesting the theme of the thesis, and being available for every odd question, irrespective of his current timezone. I would also like to thank him for trusting me with ideas in parameterized complexity before I knew any complexity.

I owe one (each) to Somnath Sikdar, Geevarghese Philip, Anand Avati and Michael Dom, for numerous helpful and fruitful discussions. Special thanks to M. Praveen for pointing me to pgf/tikz, and a lot of subsequent help in and out of context. Somdeb Ghose, thank you, for keeping my enthusiasm with illegible fonts and BabyPink under control, and (wiki-)answers to every question I threw at you.

I owe the pretty fonts to S. P. Suresh, who was also kind enough to be available for compiling my documents whenever I was unable to (even when it involved installing sixty-eight dependencies at his end). Thanks also for the pointer to XeLaTeX!

To every author who has contributed to making TeX, related macros and packages the wonderful typesetting system that it is - I owe you all a painless experience in authoring this document - *merci!*

For handling much abuse without complaint - many thanks to Aleph Naught, my laptop.

My mother has been responsible for ensuring that the thesis is completed in the same semester that I started it. I thank her for all the sleepless nights she spent trying to wake me up. Further, I owe any signs of partial sanity, and various phone bills, to my father.

Contents

List of Figures

List of Tables

0. Introduction

There are two ways of meeting difficulties:
you alter the difficulties or you alter yourself to meet them.
Phyllis Bottome

In attacking computationally hard problems, it is common (especially in practice) to attempt "reducing" the input instance using efficient pre-processing. This generally involves obtaining, from the given instance, an equivalent one that is simpler. To make the notion to be precise, we will need well-defined measures of efficiency and simplicity. In our discussions, for the former, we use the conventional benchmark of *polynomial time computability*. As for simplicity, we restrict ourselves to considerations of *size*. Thus our attempts will be concentrated on making the instance size as small as possible, where the size of an instance is defined according to the problem under consideration.

Many input instances have the property that they consist of some parts that are relatively easy to handle, and other parts that form the "really hard" core of the problem. The data reduction paradigm aims to efficiently "cut away easy parts" of the given problem instance and to produce a new and size-reduced instance where exhaustive search methods and other cost-intensive algorithms can be applied.

Preprocessing, as an algorithmic technique, is important and underrated in roughly equal proportions. In recent times, ideas from parameterized complexity have offered a natural but formal framework for the study of data reduction.

What is a natural notion of *efficient* data reduction?

We work with the assumption that any reasonable data reduction routine must be efficiently implementable. This is only natural - considering that a pre-processing routine that leaves us with half of the problem after taking the same amount of time that a brute force algorithm would (to *solve* the problem) is not very interesting.

Since we expect all reduction routines to be efficiently executable, to distinguish between them further, we measure efficiency by the extent to which a problem "shrinks"

with respect to the data reduction.

In recent times, much work has been done to examine questions related to the feasibility of efficient data reduction. It turns out that a certain hardness creeps up even among problems which are known to be otherwise tractable within the context of parameterized complexity.

In this document, we introduce what is usually meant by efficient data reduction, and survey the literature that gives us insight into when not to expect efficient data reductions.

0.1 Notation and Conventions

The set of natural numbers (that is, nonnegative integers) is denoted by $\mathbb{N}$. For a natural number n let $[n] := \{1, \ldots, n\}$. By $\log n$ we mean $\lceil (\log n) \rceil$ if an integer is expected. For $n = 0$, the term $\log n$ is undefined.

We denote a parameterized problem as a pair (Q, κ) consisting of a classical problem $Q \subseteq \{0,1\}^*$ and a parameterization $\kappa : \{0,1\}^* \to \mathbb{N}$, where κ is required to be polynomial time computable even if the result is encoded in unary.

Generally speaking, we use letters $G, H, \ldots$ to denote graphs, $u, v, \ldots$ to denote vertices of graphs, and $U, V, \ldots$, to denote subsets of vertices of graphs. Further, $\alpha, \beta, \ldots$ denote formulas, and $x, y, \ldots$ denote variables. L and Q will be used to denote problems, or languages, and k or κ (and, l, if the need arises) is reserved for the all-important parameter. We will also confusingly use $x_1, \ldots, x_t$ to denote problem instances. We will be explicit when we run out of letters and begin to stray away from this convention.

0.2 Graphs: Elementary Definitions

In this section we collect some elementary definitions of the terminology that we will frequently use.

For a finite set V, a pair $G = (V, E)$ such that $E \subseteq V^2$ is a graph on V. The elements of V are called *vertices*, while pairs of vertices (u, v) such that $(u, v) \in E$ are called *edges*.

In the following, let $G = (V, E)$ and $G' = (V', E')$ be graphs, and $U \subseteq V$ some subset of vertices of G.

The union of graphs G and G' is defined as $G \cup G' = (V \cup V', E \cup E')$, and their intersection is defined as $G \cap G' = (V \cap V', E \cap E')$.

U is said to be a *vertex cover* of G if every edge in G is adjacent to at least one vertex in U.

U is said to be an *independent set* in G if no two elements of U are adjacent to each other. The *independence number* of G is the number of vertices in a largest independent set in G.

U is said to be a *clique* in G if every pair of elements of U is adjacent to each other. The *clique number* of G is the number of vertices in a largest clique in G. A clique on three vertices is also called a *triangle*.

U is said to be a *dominating set* in G if every vertex in $V \setminus U$ is adjacent to some vertex in U. The *domination number* of G is the number of vertices in a smallest dominating set in G.

Let G' be a subgraph of G. If E' contains all the edges $\{u, v\} \in E$ with $u, v \in V'$, then G' is an *induced subgraph* of G, *induced by* V'. For any set of vertices $U \subseteq V$, $G[U]$ denotes the subgraph of G induced by U.

The *distance* between vertices u, v of G is the length of a shortest path from u to v in G; if no such path exists, the distance is defined to be ∞. The *diameter* of G is the greatest distance between any two vertices in G.

Graph G is said to be *connected* if there is a path in G from every vertex of G to every other vertex of G. If $U \subseteq V$ and $G[U]$ is connected, then U itself is said to be connected in G.

Removing a vertex v from a graph G involves deleting v from the vertex set, and removing all edges incident on v.

0.3 Parameterized Complexity

If a problem is NP-hard, then it is not likely that there is an algorithm that solves the problem exactly in polynomial time. Fortunately, much has indeed been done about many NP-hard problems. Randomized algorithms([MR96]), approximation algorithms([ACK+00, Hoc97]), and exact exponential algorithms([Woe03]) are the more popular modes of attack, and have been studied quite extensively. The notion of parameterized complexity is yet another way of tackling NP-hardness. Ideas from the parameterized approach may be used standalone, or in combination with one or more of the aforementioned techniques, to yield better algorithms. Although relatively recent, it is a standard tool of the trade by now. We limit ourselves to providing only some of the elementary definitions in this section.

0.3.1 Fixed-parameter tractable (FPT) algorithms

Algorithms that may take up to exponential time in the size of their input to arrive at a solution are called exact exponential algorithms. For example, a trivial algorithm for finding the smallest Vertex Cover would be to check every subset S of vertices of G to see if S is a vertex cover of G, and to pick a vertex cover of the minimum size. This algorithm yields a minimum-size vertex cover, and takes time $\Omega(2^n)$ where n is the number of vertices of G. In general, we would like to do better than this; we look for an algorithm that is significantly faster than the trivial algorithm, though it may still be exponential in n. These algorithms have running times of the form $O(c^n)$ where c is strictly smaller than 2, and in general, the race for improving the current-smallest value of c is both thrilling and unending.

Fixed-parameter tractable algorithms are a class of exact algorithms where the exponential blowup in the running time is restricted to a small parameter associated with the input size. That is, the running time of such an algorithm on an input of size n is of the form $O(f(k)\, n^c)$, where k is a parameter that is typically small compared to n, $f(k)$ is a (typically super-polynomial) function of k that does not involve n, and c is a constant. Formally,

Definition 1. A *parameterized* (decision) *problem* is a pair (Q, κ), where $Q \subseteq \Sigma^*$ is a language, and $\kappa : \Sigma^* \to \mathbb{N}$ is a function which is computable efficiently even when its output is to be specified in unary. The image of a string under κ is called the parameter of the problem.

Definition 2. A parameterized problem L is *fixed-parameter tractable* if there exists an algorithm that decides in $f(k) \cdot n^{O(1)}$ time whether $(x, k) \in L$, where $n := |x|$, $k := \kappa(x)$, and f is a computable function that does not depend on n. The algorithm is called a *fixed parameter algorithm* for the problem. The complexity class containing all fixed parameter tractable problems is called FPT.

Strictly speaking, the parameter can be anything that can be extracted out of the strings of a language, and can be written down efficiently (even in unary). Thus the value of the parameter $\kappa(x)$ is limited only by $|x|$. However, the notion of a parameter arises in many natural ways while considering a problem. Many graph problems, for example, are parameterized by structural aspects of graphs, like their maximum degree, average degree, girth, optimal vertex cover size, treewidth, etc. In most problems that involve combinatorial optimization, we are required to find an optimal (which may mean the largest or the smallest, depending on context) subset of the problem that is required to satisfy certain given properties. The size of a valid "solution" (which refers to any subset of the desired kind) is a natural option for a parameter. For instance, given a graph, we may be looking for the smallest set of vertices whose removal makes a graph acyclic - but notice that the following "parameterization" is quite natural: given a graph along with an integer k, we would like to know if there is a subset of k vertices (or less) whose removal makes the graph acyclic.

As noted in [Nie06]:

> In general it is not always straightforward to find the "right" parameterization for a problem - as a matter of fact, several reasonable, equally valid parameters to choose may exist and it often depends on the application behind or additional knowledge about the problem which parameterization is to be preferred.

As another illustration of parameterizing by solution size, consider the minimum vertex cover problem - we shift our focus to the slightly different issue of finding if the input graph G has a vertex cover of size l or less, and finding such a vertex cover if it does exist, for a *fixed* number l that is independent of the number of vertices n in the input graph. A trivial way to do this is to check, for each l-subset U of the vertex set V of G, whether U is a vertex cover of G. There are $\binom{n}{l} = \Theta\left(n^l\right)$ l-subsets of V, and for a given set U of l vertices, it takes $\Theta(n)$ time in the worst case to check if U is a vertex cover of G. Thus this algorithm runs in $\Theta\left(n^{l+1}\right)$ time in the worst case.

However, we now describe a simple algorithm $\mathcal{A}$ that checks if G has a vertex cover of size l, and finds such a vertex cover if it exists, in $O\left(2^l n\right)$ time (see Section 1.4 of [Nie06]). $\mathcal{A}$ thus runs in *linear* time in the size of its input. For moderately large values of l, $\mathcal{A}$ is thus feasible for *any* value of n that could arise in practice; compare this with the naive algorithm described above that takes $\Theta\left(n^{l+1}\right)$ time in the worst case, which becomes impractical for even moderately large n for comparatively smaller values of l.

There is in fact a faster algorithm, call it algorithm $\mathcal{B}$, that solves this problem: it finds if the input graph G on n vertices has a vertex cover of size l or less, and finds such a vertex cover if it does exist, in $O\left(1.2745^l l^4 + ln\right)$ time ([CG05]). Algorithm $\mathcal{B}$ is thus feasible for a much larger range of l than algorithm $\mathcal{A}$. Note also that by repeating $\mathcal{A}$ (resp. $\mathcal{B}$) for $l = 1, 2, \ldots$ till it finds a vertex cover, we can solve the Minimum Vertex Cover problem exactly in $O\left(2^k n\right)$ (resp. $O\left(1.2745^k k^4 + kn\right)$) time, where k is the size of the smallest vertex cover in the graph.

Note that the algorithms described above ($\mathcal{A}$ and $\mathcal{B}$) are both fixed-parameter tractable algorithms, when the problem is parameterized by "solution size".

Other parameterized complexity classes may be defined based on the kind of f that we are able to achieve algorithmically. (FPT corresponds to the most general definition, with all the allowance - limited only by computability.) If f can be chosen such that $f(k) = 2^{(k^{O(1)})}$, then (Q, κ) is in EXPT. If f can be chosen such that $f(k) = 2^{O(k)}$, then (Q, κ) is in EPT. If f can be chosen such that $f(k) = 2^{o(k)}$, then (Q, κ) is in SUBEPT.

The theory of parameterized complexity is now considered a standard approach to

solving NP-hard problems, often coupled with other mainstream techniques (such as approximation, exact and randomized algorithms) to obtain efficient algorithms for hard problems. The techniques used in parameterized complexity include, but are not limited to, the following:

1. **Depth-bounded search trees:** Briefly, the technique involves finding a "small subset" (in polynomial time) of the input instance such that at least one element of this subset is part of an optimal solution to the problem. For instance, in case of Vertex Cover this "small subset" is a two–element set consisting of the two endpoints of an edge–one of these two vertices has to be part of the vertex cover. This leads to a search tree of size 2^k for Vertex Cover, where the parameter k denotes the size of the vertex cover. The reader may refer to [Nie06] for more intricate variations of the basic idea.

2. **Iterative compression:** A result on finding odd–cycle traversals in [RSV04] led to what is now a standard paradigm for designing FPT algorithms. This setup works when the problems are parameterized by the size of the solution. We assume that a solution of size $(k+1)$ is "given", and attempt to compress it to a solution of size k. The usual method is to begin with a subgraph that consists of $(k+1)$ vertices, this generally admits a (trivial) k–sized solution. Then we expand the subgraph under consdieration one vertex at a time, and use the compressed solution (for the old graph) along with the new vertex as a starting point as the current solution. The solution is compressed every time the underlying graph is expanded, so the compression routine is run iteratively (hence the name). We stop when we either get to a solution of size k for the entire graph, or if an intermediate instance turns out to be incompressible. If the compression can be carried out in $f(k)$ time, the overall algorithm would take $(n-k)\cdot f(k)$, i.e., FPT time. For an example of the utility of this technique in designing exact exponential algorithms, see [FGK+08]. For an example of the algorithm being applied to the Cluster Vertex Deletion problem (where one is required to check if there exist k vertices whose removal makes the graph a disjoint union of cliques), see [HKMN08].

3. **Color coding:** Here we begin by designing a randomized algorithm, which is

eventually made deterministic by the use of hash families. Although the original intention was to aid the design of FPT algorithms, it has nearly become as much a tool for algorithm design as it is a pretext for the discovery of new kinds of hash families. The details of the technique are beyond the scope of this discussion, however, the interested reader will find an engaging exposition in [AYZ95].

4. **Dynamic programming over tree decompositions:** Often, many problems that are NP–complete on general graphs turn out to be polynomial time solvable on trees. Some intuition may be drawn from the fact that the at any node in a tree, all the subtrees rooted at it are disjoint from each other, and thus information collected at them can be "merged" quite easily by any dynamic programming routine that's moving upwards from the leaves. This is good enough motivation to study the possibility of writing down an arbitrary graph in way that it resembles the "separation" properties so prevalent in trees. The quantity *treewidth*, which is developed on top of the notion of balanced separators in graphs, formalizes the notion of the "tree-likeness" of a graph, and has been heavily investigated, especially as a parameter. Again, while it is not feasible for us to go into the exciting details of this approach, we note that it is surveyed in [Bod87].

For an introduction to the parameterized complexity approach, we point the reader to the books [Nie06, DF99, FG06], and the surveys [Gro02, Ram97, Mar08, HNW08].

0.4 Logic and Formulas

The problem of satisfiability in propositional logic is one of the most well investigated problems in both classical and parameterized complexity. It is the earliest problem that was shown NP-complete. Intriguing enough is the fact that most implementations of the brute force algorithm that runs in exponential time turn out to be efficient on a large selection of inputs in practice. Such evidence motivates the question of pinning down the aspects of the problem that make it hard to solve in the worst case, and identifying those aspects that give it a reasonable generic complexity (which informally refers to the fact that a large fraction of inputs are well-behaved). A natural place to

look for the answers to these questions is to examine the parameterized complexity of the problem. As it turns out, the problem offers numerous natural parameterizations (and many more less natural ones). We come back to this problem time and again; and we spend this section on its definition.

Let P be an arbitrary set, whose elements we shall refer to as *variables*. It will be convenient to assume that P is a countably infinite set. The set of formulas over P is inductively defined to be the smallest set of expressions such that:

- Each variable in the set P is a formula
- $(\neg\alpha)$ is a formula whenever α is, and
- $(\alpha \square \beta)$ is a formula whenever α and β are formulas and $\square$ is one of the binary connectives $\wedge, \vee$.

We denote by $\mathcal{F}(P)$ the set of all formulas over P.

A valuation of P is a function $\nu : P \to \{0, 1\}$, which may be extended to a function $\bar{\nu} : \mathcal{F}(P) \to \{0, 1\}$, as follows:

- For each variable x in the set P, $\bar{\nu}(x) = \nu(x)$.
- $\bar{\nu}(\neg\alpha) = 1 - \bar{\nu}(\alpha)$,
- $\bar{\nu}(\alpha \wedge \beta) = \min\{\bar{\nu}(\alpha), \bar{\nu}(\beta)\}$
- $\bar{\nu}(\alpha \vee \beta) = \max\{\bar{\nu}(\alpha), \bar{\nu}(\beta)\}$

A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, where a clause is a disjunction of literals. Every propositional formula can be converted into an equivalent formula that is in CNF.

The question of *satisfiability* is whether, given a formula α, there exists a valuation ν such that $\nu(\alpha) = 1$. This is one of the most well-studied NP-complete problems. Of course, it continues to be NP-complete if the formula is offered in CNF, (in fact, it remains NP-complete even when every clause has no more than three variables).

The following is one parameterized version of the problem that will be of interest to us later:

p-SAT	
Instance:	A propositional formula α in conjunctive normal form.
Parameter:	Number of variables of α.
Question:	Is α satisfiable?

However, there are many ways of parameterizing the problem of satisfiability. For a given formula α, some examples of the possibilities for $\kappa(\alpha)$ are:

1. Number of clauses of α.
2. The length of the formula α.
3. The size of the longest clause in α.
4. If instances x of satisfiability are specified as pairs (α, k), where $k \in \mathbb{N}$, then a possible parameter is simply $\kappa(x) = k$. We may ask if there exists a satisfying assignment that has at most k 1's, exactly k 1's, and at least k 1's.

0.5 Organization

In Chapter 1, we will examine established notions of data reduction, and observe that the notion of "reducibility" with respect to one or more parameters corresponds to "tractability" in the parameterized version (using the same parameters) of a problem's complexity. We examine the different kinds of reducibility in the interest of obtaining a finer classification of the set of fixed-parameter tractable problems. The most conspicuous sub-class is the set of problems that admit what we know as *polynomial kernels*, which will be used as our notion of efficient data reduction.

We show next, in Chapter 2, that it is not possible for every fixed-parameter tractable problem to admit efficient data reduction, under reasonable complexity-theoretic assumptions. We build a general framework that allows us to look at a parameterized language and prove the hardness of efficient reduction in particular. In particular, we

rule out the possibility of efficient data reductions for problems that admit *composition algorithms*.

There has been much work in recent times towards showing composition algorithms for many specific problems. We study some of them in Chapter 3, and observe that there seems to be a striking similarity among these algorithms.

However, showing composition algorithms for problems is not the only way to achieve conclusive lower bounds. The paradigm of reductions[1] among problems is one of the most heavily exploited in computer science, and in Chapter 4 we describe certain restricted transformations that give us hardness of polynomial kernelization without having to design compositions.

Finally we spend some time "dealing" with the hardness of polynomial kernelization in Chapter 5. It can be shown that although polynomial kernels are unlikely, there is nothing that prevents us from finding multiple kernels - in particular, we study a problem for which there are as many as n polynomial kernels. To begin with, it is not even clear that the problem is going to shrink because of this, but we discuss why the notion is interesting nevertheless.

The technique for showing the non-existence of polynomial kernels is well developed, but we do not know how to argue finer hardness results. For instance, a procedure for showing that problem is unlikely to admit a linear kernel is not known. In Chapter 6, we finally take stock of what techniques would make the picture of lower bounds on efficient kernelization more complete.

[1] Not to be confused with the reduction in "data reduction"!

1. On Polynomial Kernels

Eliminate all other factors,
and the one which remains must be the truth.
Sir Arthur Conan Doyle

Evil is whatever distracts.
Franz Kafka

In [Felo6], Michael Fellows points out the following:

> Pre-processing is a humble strategy for coping with hard problems, almost universally employed. It has become clear, however, that far from being trivial and uninteresting, that pre-processing has unexpected practical power for realworld input distributions, and is mathematically a much deeper subject than has generally been understood. It is almost impossible to talk about pre-processing in the classical complexity framework in any sensible and interesting way, and the historical relative neglect of this vital subject by theoretical computer science may be related to this fact.

Kernelization is pre-processing formalized. It even turns out that the notion of a problem being *kernelizable* corresponds exactly to the notion of its being fixed-parameter tractable. Although it is almost trivial to establish this - the semantics of the statement are less than trivial. In this chapter, we define kernelization on parameterized languages and variations of the default notion, before we can begin looking at conditional lower bounds on efficient kernelization.

1.1 Kernels and Parameterized Tractability

A kernelization algorithm for a parameterized language (Q, κ) is a polynomial time procedure that converts an instance x into y, such that $|y| = f(k)$ and $x \in Q$ if and

only if $y \in Q$. A kernelization procedure gives us a reduced instance whose size can be described as a function of k alone, and this is usually called the kernel of the problem.

Definition 3. Let (Q, κ) be a parameterized problem over $\{0,1\}^*$. A polynomial time computable function $K : \{0,1\}^* \to \{0,1\}^*$ is a kernelization of (Q, κ) if there is a computable function $h : \mathbb{N} \to \mathbb{N}$ such that for all $x \in \{0,1\}^*$ we have

$$(x \in Q \iff K(x) \in Q)$$

and

$$|K(x)| \leqslant h(\kappa(x)).$$

If K is a kernelization of (Q, κ), then for every instance x of Q the image $K(x)$ is called the kernel of x (under K).

Observe that a kernelization is a polynomial time many-one reduction of a problem to itself with the additional property that the image is bounded in terms of the parameter of the argument. We now establish fixed-parameter tractability and kernelization are identical notions. Although easy to see, the theorem is at the heart of our understanding of both kernelization and parameterized tractability. Having realized that FPT *is* the class of kernelizable problems, we also now have another way of getting deeper into this class (using, for example, of the quality of kernels as a basis for classification). We follow the presentation in [FG06], although the theorem can be found in [DF99] and [Nie06] as well.

Theorem 1. *For every parameterized problem (Q, κ), the following are equivalent:*

(1) $(Q, \kappa) \in \mathsf{FPT}$.

(2) Q *is decidable, and* (Q, κ) *has a kernelization.*

Proof. Let $\{0,1\}^*$ be the alphabet of (Q, κ).

$(2) \Rightarrow (1)$: Let K be a kernelization of (Q, κ). The following algorithm decides Q: Given $x \in \{0,1\}^*$, it computes $K(x)$ (in polynomial time) and uses a decision algorithm for Q to decide if $K(x) \in Q$. Since $|K(x)| \leqslant h(\kappa(x))$, the running time of the decision algorithm is effectively bounded in terms of the parameter $\kappa(x)$.

(1) $\Rightarrow$ (2) : Let $\mathbb{A}$ be an algorithm solving (Q, κ) in time $f(k) \cdot p(n)$ for some computable function f and polynomial $p(x)$. Without loss of generality we assume that $p(n) \geqslant n$ for all $n \in \mathbb{N}$. If $Q = \phi$ or $Q = \{0,1\}^*$, then (Q, κ) has the trivial kernelization that maps every instance $x \in \{0,1\}^*$ to the empty string ϵ. Otherwise, we fix $x_0 \in Q$ and $x_1 \in \{0,1\}^* \setminus Q$. The following algorithm $\mathbb{A}'$ computes a kernelization K for (Q, κ): Given $x \in \{0,1\}^*$ with $n := |x|$ and $k := \kappa(x)$, the algorithm $\mathbb{A}'$ simulates $p(n) \cdot p(n)$ steps of $\mathbb{A}$. If $\mathbb{A}$ stops and accepts (rejects), then $\mathbb{A}'$ outputs x_0 (x_1, respectively). If $\mathbb{A}$ does not stop in $\leqslant p(n) \cdot p(n)$ steps, and hence $n \leqslant p(n) \leqslant f(k)$, then $\mathbb{A}'$ outputs x. Clearly, K can be computed in polynomial time, $|K(x)| \leqslant |x0| + |x1| + f(k)$, and $(x \in Q \iff K(x) \in Q)$. □

Note that in the proof of Theorem 1, the size of the kernel is $f(k)$, which is typically exponential in k. A kernelization is rarely described as the automatic algorithm that can be derived from any given algorithm that solves the problem in FPT time.

A kernelization procedure is usually described as a collection of reduction rules, which are designed to transform the instances preserving equivalence. Each of these rules are applied to the instance recursively. Each application causes the instance to shrink in some way, and one hopes to be able to *prove* that if the rules have been applied till they are not applicable any more, then the resulting instance must be a kernel.

A striking example of the extent to which data reduction is effective was given by Weihe [Wei98, AFN02], when dealing with the NP-complete Red/Blue Dominating Set problem appearing in context of the European railroad network. In a preprocessing phase, two simple data reduction rules were applied again and again until no further application was possible. The impressive result of this empirical study was that each of these real-world instances were broken into very small pieces such that for each of these a simple brute-force approach was sufficient to solve the hard problems quite efficiently.

We make note of what is pointed out in [Nie06] in this context:

> Observe the "universal importance" of data reduction by preprocessing. It is not only an ubiquitous topic for the design of efficient fixed-

parameter algorithms, but it is of same importance for basically any method (such as approximation or purely heuristic algorithms) that tries to cope with hard problems. Refer [CDLS02, Lib01] to for related considerations concerning "off-line preprocessing" of intractable problems (with a special emphasis on computational problems from artificial intelligence and related fields).

Naturally, we would like the kernel produced by the kernelization algorithm to be as "small" as possible. This motivates the notion of polynomial kernels, which we see next, and we also describe some rules to reduce instances of the Vertex Cover problem in then. Before that, however, we describe a simple kernel p-SAT, the satisfiability problem for propositional logic parameterized by the number of variables.

p-SAT	
Instance:	A propositional formula α in conjunctive normal form.
Parameter:	Number of variables of α.
Question:	Is α satisfiable?

The following simple algorithm([FG06]) computes a kernelization for p-SAT: Given a propositional formula α with k variables, it first checks if $|\alpha| \leqslant 2k$. If this is the case, the algorithm returns α. Otherwise, it transforms α into an equivalent formula α' in disjunctive normal form such that $|\alpha'| \leqslant O(2^k)$. Note that the size of the parameter does not change in this transformation. The number of variables in the kernel is the same as the number of variables in the original instance.

1.2 Polynomial Kernels

A polynomial kernel is a kernel whose size is polynomial in the original parameter. They capture the essence of what we want to mean by *efficient kernelization*. Notice that in the context of kernelization, the adjective of efficiency has nothing to do with the runtime of the kernelization procedure (as every kernelization is required to run in polynomial time by definition). The measure of efficiency is the size of the kernel - the smaller the better.

Definition 4. Let (Q, κ) be a parameterized problem over $\{0,1\}^*$. A polynomial time computable function $K : \{0,1\}^* \to \{0,1\}^*$ is a polynomial kernelization of (Q, κ) if there is a polynomial $h : \mathbb{N} \to \mathbb{N}$ such that for all $x \in \{0,1\}^*$ we have

$$(x \in Q \iff K(x) \in Q)$$

and

$$|K(x)| \leqslant h(\kappa(x)).$$

We give an informal description of a simple polynomial kernel for the vertex cover problem:

Vertex Cover
Input: A graph $G = (V, E)$ and a positive integer $k \leqslant \|V\|$.
Question: Does G have a vertex cover of size k or less?

We describe some valid reduction rules for the Vertex Cover problem:

Rule 1: An isolated vertex u (vertex of degree 0) can not be in a vertex cover of optimal size. Since there are no edges associated with such a vertex, there is no benefit of including it in any vertex cover, so we may delete all the isolated vertices from a graph.

Rule 2: In the case of a pendant vertex u (a vertex of degree 1), there is a vertex cover of optimal size that does not contain the pendant vertex but does contain its unique neighbor v. Thus, we may delete both u and v and the edges incident on v.

Rule 3: Suppose the degree of a vertex u is more than k. Observe that u is forced to be a part of any vertex cover of size at most k. (Any k-sized vertex cover cannot afford to exclude u, because if u does not belong to the vertex cover, then to cover the edges incident on u, all its neighbors must belong to the vertex cover, but this would imply a vertex cover of size $k + 1$.) Thus, we may delete any vertex u such that $d(u) > k$, and for every such u, we decrease the parameter by 1.

Rule 4: Note that in any vertex cover, each vertex may cover at most k edges (since Rule 3 has been applied on all vertices of the graph). Thus, if the number of edges in the graph is in excess of k^2, we may safely rule out the possibility of its having a vertex cover of size k. Therefore, we terminate and say NO, if the graph has more than k^2 edges after the application of Rule 3.

The first step towards kernelizing a graph is to 'apply' these rules recursively. We begin with $U = \phi$ - this initializes the vertex cover to the empty set. Then, we delete isolated and pendant vertices, for all v such that $d(v) > k$, we have $U := U \cup \{v\}$ and $k := k - 1$. We repeat these applications till none of the rules are valid, in other words, we stop when we are left with a graph where the degree of every vertex in the graph is at most k. We refer to such a graph as a *reduced* instance.

We claim that the number of edges on any reduced graph that is a yes-instance is at most k^2, because of Rule 4. Thus, we have a kernel (as a bound on the number of edges also implies that the number of vertices is bounded - it's useful, in this context, to know that there are no isolated vertices).

There are many examples of kernels that are obtained using clever reduction rules, the vertex cover problem alone is known to admit a handful of different kinds of kernelizations.

Polynomial kernels have been discovered for a large number of problems. Vertex Cover is known to admit a vertex–linear kernel via either the use of linear programming (where the problem kernel has $2k$ vertices), or the so–called *crown reduction* rules (using which the kernel has at most $3k$ vertices). Details for both may be found in the book, [Nie06]. For work on a more general variant of Vertex Cover, see [CC08]. The Feedback Vertex Set problem (for which we are required to check if there exist k vertices whose removal makes the graph acyclic), for instance, has a quadratic kernel [Tho09]. For the problem of editing at most k edges (that is, adding non-existent edges and deleting given edges) such that the resultant graph is a disjoint union of cliques - called Cluster Editing, a quadratic kernel is known. For an example of work on Cluster Editing, see [GGHN03]. All these problems occur in numerous practical situations, and the kernelizations known have a very high utility quotient in the real world.

It is quite impossible to provide a fair glimpse into the class of problems with polynomial kernels, so we only refer to [GN07] for a survey on them.

1.3 Strong Kernels

If a kernelization K for a problem (Q, κ) has the additional property that

$$\kappa(K(x)) \leqslant \kappa(x),$$

then the kernelization is said to be *strong*. Thus we have the following definition:

Definition 5. Let (Q, κ) be a parameterized problem over $\{0,1\}^*$. A polynomial time computable function $K : \{0,1\}^* \to \{0,1\}^*$ is a strong kernelization of (Q, κ) if there is a computable function $h : \mathbb{N} \to \mathbb{N}$ such that for all $x \in \{0,1\}^*$ we have

$$(x \in Q \iff K(x) \in Q),$$

$$|K(x)| \leqslant h(\kappa(x)),$$

and

$$\kappa(K(x)) \leqslant \kappa(x).$$

They are also known as proper kernels in the literature ([AKF06]). We will observe later that any strong polynomial kernelization cannot have the property that

$$\kappa(K(x)) < \kappa(x)$$

unless $P \neq NP$, that is, a strong polynomial kernel that is strictly parameter-decreasing is highly implausible.

It is easy to see, from the proof of Theorem 1 that every problem that admits a kernel also admits a strong kernel. However, this does not apply within the domain of polynomial kernels. In particular, there exist problems that, on the one hand, have polynomial kernels, but have no strong polynomial kernels.

Theorem 2 ([CFM07])**.** *There is a parameterized problem* (Q, κ) *that has a polynomial kernelization but no strong polynomial kernelization.*

Proof. Let Q be a classical problem that is not solvable in time $2^{O(|x|)}$. We define a parameterized problem (P, κ) with $P \subseteq \{0,1\}^* \times 1^*$ and with $\kappa((x, 1^k)) = k$. By 1^k we denote the string consisting of k many 1s. For each $k \in \mathbb{N}$ we define the k-projection $P[k] := \{x \mid (x, 1^k) \in P\}$ of P by:

1. If $k = 2l + 1$, then $P[k] := Q_{=l}$, where $Q_{=l} := \{x \in Q \mid |x| = l\}$.

   Hence, all elements in $P[k]$ have length l.

2. If $k = 2l$, then $P[k] := \{x1^{2^l} | x \in Q_{=l}\}$, where $x1^{2^l}$ is the concatenation of x with the string 1^{2^l}.

   Hence, all elements in $P[k]$ have length $l + 2^l$.

Intuitively, an element in the $2l$-projection is an element in the $(2l+1)$-projection padded with 2^l many 1s. The following is a linear kernelization for P. Given (x, k), the kernel is computed as follows:

1. If k is odd: Let $k = 2l+1$. If $|x| = \frac{k-1}{2}$, then return (x, k), else return a trivial NO–instance.

2. If k is even: Let $k = 2l$. If x does not end with 2^l 1's, then return a trivial NO–instance. Else, let y denote the string x with the last 2^l 1's removed. If $|y| = k/2$, we return $(y, k + 1)$, else return a trivial NO–instance.

The correctness of the kernel is immediate when k is odd. When k is even, note that $(x, k) \in (P, \kappa)$ if and only if $(y \in Q_l)$. This is a linear kernel - in the first instance, the length of the kernel is k and in the second, it is $k/2$. Note that the parameter increases on the even-length projections.

We claim that P has no strong polynomial kernelization. Assume K is such a kernelization and $c \in \mathbb{N}$ such that

$$|K((z, 1^m))| \leqslant m^c.$$

We use K to solve $x \in Q$ in time $2^{O(|x|)}$:

Let x be an instance of Q and let $l := |x|$. We may assume that

$$(2l)^c < 2^l$$

(note that there are only finitely many x not satisfying this inequality). We compute (in time $2^{O(l)}$) $(u, k) := K((x1^{2^l}, 2l))$.

We know that $k \leqslant 2l$ and $|u| \leqslant (2l)^c < 2^l$. If u does not have the length of the strings in $P[k]$, then $(u, k) \notin P$ and therefore $x \notin Q$. In particular, this is the case if $k = 2l$ (as $|u| < 2l$). If u has the length of the strings in $P[k]$ and hence $k < 2l$, then it is easy to read off from u an instance y with $|y| < |x|$ and $(y \in Q \iff x \in Q)$. We then apply the same procedure to y.

□

1.4 Polynomial Pseudo-Kernels

Sometimes, while it may be hard to obtain a polynomial kernel, we may get close: the following is the notion of a polynomial pseudo-kernel, where the kernel size is required to be a polynomial in k and is allowed as a multiplicative factor, a function that is pseudo-linear in n, i.e., the size of the kernel is not purely a function of the parameter k.

Definition 6. Let (Q, κ) be a parameterized problem over $\{0,1\}^*$. A polynomial time computable function $K : \{0,1\}^* \to \{0,1\}^*$ is a polynomial pseudo–kernelization of (Q, κ) if there is a polynomial $h : \mathbb{N} \to \mathbb{N}$ such that for all $x \in \{0,1\}^*$, and some $\varepsilon > 0$ we have

$$(x \in Q \iff K(x) \in Q)$$

and

$$|K(x)| \leqslant h(\kappa(x)) \cdot n^{1-\varepsilon}.$$

This definition is prompted by the methods described in [CFM07] that allow us to conditionally rule out the existence of such kernels. We will describe these methods

presently. The notion of a polynomial pseudo-kernel is weaker than that of a polynomial kernel, and a lower bound here would therefore be stronger than the lower bounds on polynomial kernels.

2. No Polynomial Kernels

The irrationality of a thing is no argument against its existence, rather a condition of it.

Friedrich Wilhelm Nietzsche

We are now ready to examine the circumstances under which we do not expect a polynomial kernel for parameterized problem. We spend this chapter describing a characterization of problems that do not have polynomial-sized kernels[1] using a certain property. This turns out to be useful - showing that a problem is unlikely to admit a polynomial kernel boils down to demonstrating this property for the problem. A different, but similar property leads to a technique for proving stronger hardness results - in particular, we will also see a characterization of problems that are unlikely to have even pseudo-polynomial kernels.

2.1 Historical Remarks

One of the earliest signs of lower bounds on kernels came from the domain of approximation algorithms. For Vertex Cover, a kernel of size $2k$ corresponds to a 2–approximation to the question of finding a vertex cover of size k. The following is an informal argument to justify this. We run the kernelization, and include any vertices that are "forced into the solution" by the reduction rules. Then we may output the entire kernel along with these vertices as the vertex cover. For the general optimization question, we try various values of k. We may, in the worst case, have to examine all the possibilities for k, namely, $1, 2, 3, \ldots n$. For the smallest value of k for which we get a kernel that is not a trivial NO–instance, we return the kernel as an approximate solution.

By the same token, a $(2-\varepsilon)$–sized kernel could correspond to a ratio $(2-\varepsilon)$ approximation, provided that the kernelization is also an approximation–preserving reduc-

[1] Under reasonable complexity theoretic assumptions, specifically under the assumption that PH $\neq \Sigma_3^p$.

tion. However, improving the approximation ratio for Vertex Cover to anything smaller than 2 has been a long-standing open problem in approximation algorithm theory, and it is even unexpected [KR08]. If what is known as the *unique games conjecture* ([Kho02]) is true, then 2 is the best approximation ratio one can get for Vertex Cover.

Further, linear kernels with constants smaller than 1.36 would imply $P = NP$, again because of results from approximation theory. The following is known about Vertex Cover on Planar Graphs.

Theorem 3 ([CFKX05])**.** *For any* $\varepsilon > 0$, Vertex Cover on Planar Graphs *has no problem kernel formed by a planar graph with* $(4/3 - \varepsilon) \cdot k$ *vertices, unless* $P = NP$.

The flavor of the arguments used here can be used to rule out linear kernels with certain constant factors for other problems as well. However, it is not known if they generalize to showing non-linear lowerbounds. In [Nie06], it is remarked that "much remains to be done in the field for lower bounds for problem kernels". Indeed, there have been some developments since, and we turn to them now.

2.2 The More Recent Story

The promised characterization of problems that do not have polynomial kernels relies on the known infeasibility of "compressing" NP-complete problems. This is a recurring theme in this chapter, so we begin by providing some context to the definitions that we will soon need.

Let L be a language in NP. The notion of compressibility of L roughly corresponds to the existence of a polynomial time algorithm C that preserves instances ($C(x) \in L$ if and only if $x \in L$), and "compresses' them (the length of $C(x)$ is polynomial in the *length of the witness of* x). This features in the work of Harnik and Naor, [HN06], which was motivated by cryptographic applications - one of the earliest contexts in which the notion of *instance compression* turned out to be useful.

In [FS08], Fortnow and Santhanam re-formulated this definition in a way that separated the quantity in which the size of the compressed instance is required to be

small, instead of always tying it down to the size of the witness. A *parametric problem* is defined to be one where an instance of the problem is supplied with an additional parameter. The compression algorithm is required to behave as before, but now with the requirement that the length of its output is polynomial in this parameter. The new definition gives us some flexibility - for instance, it enables us to examine parameters that have nothing to do with the size of the witness.

They go on to establish that the general problem of satisfiability with small witnesses is *incompressible*, to the extent that a compression routine would imply $PH = \Sigma_3^p$. That is to say, given a propositional formula ϕ that uses n variables, we do not expect to (in polynomial time) get to an equivalent formula ψ whose length is polynomial in n. This allows us to infer that the search for a polynomial kernel for the problem of satisfiability parameterized by the number of variables is already doomed.

This is quite exciting, notice that we have already encountered hardness of polynomial kernelization! And since the problem is FPT, it *does* admit a kernel - so the hardness is indeed news to us[2]. In [BDFH08], Bodlaender et al re-define some of the notions that go into showing the infeasibility of classical problems so that they can be applied directly on problems that are parameterized in the sense of parameterized complexity (as opposed to the "parametric problems" introduced by Fortnow and Santhanam, which is the same notion, but isn't quite used in the same way in the rest of their work).

Flum, Chen and Muller also execute a similar translation of results so that they can be stated in the language of parameterized problems and kernelization (rather than parametric problems and compression). In their work ([CFM07]), they extend the machinery further to show more serious implications on the hardness of kernelization, namely the non-existence of pseudo-polynomial kernels. All these implications eventually can be traced back to the notion of distillation algorithms, and the observation that NP-complete problems are unlikely to have them.

[2] If the problem was higher up in the *W*-hierarchy, we would not be so interested, since we already have good reasons to expect that the problem does not have a kernel of *any kind*, much less a polynomial kernel.

2.3 Distillation Algorithms

Fortnow and Santhanam first showed the infeasibility of compressing what they called OR-SAT. Let $\phi_i, 1 \leqslant i \leqslant t$, be instances of SAT, which is the standard language of propositional boolean formulas. OR-SAT is the following language:

$$\textit{OR-SAT} = \{< \phi_i, 1^n > | 1 \leqslant i \leqslant t \text{ at least one of the } \phi_i \text{ is satisfiable, and each } \phi_i \text{ is of length at most } n, \text{ i.e., uses at most } n \text{ variables.}\}$$

Any compression routine for *OR-SAT* can be thought of as an algorithm that accepts multiple instances of SAT and returns a small formula equivalent to the "or" of the input formulas. This motivates our first definition, introduced in [BDFH08].

A distillation algorithm for a given problem is designed to act as a "Boolean OR of problem-instances" - it receives as input a sequence of instances, and produces a yes-instance if and only if at least one of the instances in the sequences is also a yes-instance. Although the algorithm is allowed to run in time polynomial in the total length of the sequence, its output is required to be an instance whose size is polynomially bounded by the size of the maximum-size instance in its input sequence. Formally, we have the following:

Definition 7. Let $Q, Q' \subseteq \{0,1\}^*$ be classical problems. A *distillation from* Q *in* Q' is a polynomial time algorithm $\mathcal{D}$ that receives as inputs finite sequences $\overline{x} = (x_1, \ldots, x_t)$ with $x_i \in \{0,1\}^*$ for $i \in [t]$ and outputs a string $\mathcal{D}(\overline{x}) \in \{0,1\}^*$ such that

1. $|\mathcal{D}(\overline{x})| = (max_{i \in [t]} |x_i|)^{O(1)}$
2. $\mathcal{D}(\overline{x}) \in Q'$ if and only if for some $i \in [t] : x_i \in Q$.

If $Q' = Q$ we speak of a self-distillation. We say that Q has a distillation if there is a distillation from Q in Q' for some Q'.

So, given a sequence of t instances of Q, a distillation algorithm gives an output that is equivalent to the sequence of instances, in the sense that a collection with at

least one yes-instance (i.e. instance belonging to Q') is mapped to a yes-instance, and a collection with only no-instances is mapped to a no-instance. Such a task is trivial (as an OR of the formulas would suffice) but for the condition that the length of the output is bounded by the size of the maximum-size instance in its input sequence. Indeed, it is easy to check that a self-distillation algorithm without this constraint on $|\mathcal{D}(\overline{x})|$[3] exists for every NP-complete problem. In contrast, we have:

Theorem 4 ([FS08]). *If any NP-complete problem has a distillation algorithm then* $\mathsf{PH} = \Sigma_3^p$.

Let Q be an NP-complete problem with a distillation algorithm $\mathbb{A}$, and let $\overline{Q}$ denote the complement of Q. It can be shown that using $\mathbb{A}$, we can design a non-deterministic Turing machine (NDTM) that, with the help of polynomial advice, can decide Q in polynomial-time. This will show that $\mathsf{coNP} \subseteq \mathsf{NP/poly}$, and combined with Yap's theorem [Yap83] ($\mathsf{coNP} \subseteq \mathsf{NP/poly} \Rightarrow \mathsf{PH} \subseteq \Sigma_3^p$), this will prove the statement of the theorem. (For a detailed proof, cf. Appendix E or [BDFH08]).

A more general definition of distillation is the following:

Definition 8. Let $Q, Q' \subseteq \{0,1\}^*$ be classical problems, and let $f : \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ be a function. A *linear f-distillation from* Q *in* Q' is a polynomial time algorithm $\mathcal{D}$ that receives as inputs finite sequences $\overline{x} = (x_1, \ldots, x_t)$ with $x_i \in \{0,1\}^*$ for $i \in [t]$ and outputs a string $\mathcal{D}(\overline{x}) \in \{0,1\}^*$ such that

1. $|\mathcal{D}(\overline{x})| = f(t) \cdot (\max_{i \in [t]} |x_i|)^{O(1)}$
2. $\mathcal{D}(\overline{x}) \in Q'$ if and only if for some $i \in [t] : x_i \in Q$.

Towards the end of proving stronger lower bounds using the more general notion of distillation, we would need pseudo-linear functions, which are defined as below:

Definition 9. A function $f : \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ is pseudo-linear if there is some $c \in \mathbb{N}$ and some $\varepsilon \in \mathbb{R}$ such that for all $t \in \mathbb{N}$,

$$f(t) \leqslant c \cdot t^{1-\varepsilon}.$$

[3] Such algorithms are called OR^ω, the interested reader is referred to [CK95].

For a pseudo-linear function f, it turns out that the existence of linear f-distillations for NP-complete problems also implies $\mathsf{PH} = \Sigma_3^p$.

Theorem 5 ([CFM07]). *Let* $f : \mathbb{N} \to \mathbb{R}_{\geqslant 0}$ *be pseudo-linear. No* NP*-hard problem has a linear* f*-distillation unless* $\mathsf{PH} = \Sigma_3$.

We prove this along the lines of the proof of Theorem 4. We would like to exploit Theorems 4 and 5 in the context of problem kernelization. In the next section, we introduce some definitions that make the application almost immediate.

2.4 Composition Algorithms

We would like to speak of distillations for parameterized problems. These algorithms are similar to their classical counterparts in spirit, except that we may now want to specify additional constraints on the size of the parameter of the output instance. Indeed, we have various options, and the following will be of interest to us:

1. Require the new parameter to be polynomially-bounded by the maximum of all parameters in the sequence, and have no requirement on the length of the output instance.

2. Require the new parameter to be polynomially-bounded by the maximum of all parameters in the sequence, and have the length of the output bounded by the product of the number of instances and a polynomial in the size of the maximum-size instance in its input sequence.

3. Require all the parameters in the sequence to be the same, and have the new parameter polynomially-bounded in k, and have no requirement on the length of the output instance.

4. Require all the parameters in the sequence to be the same, and have the new parameter polynomially-bounded in the size of the maximum-size instance in its input sequence, and have no requirement on the length of the output instance.

The first variation is called an *OR*, the second is a *linear–OR*, and the last two are referred to as a *composition* and λ*-constant-OR*, respectively. The formal definitions are introduced in the sections where we use these algorithms. In the meantime, Table 2.1 is a summary of the different kinds of compositions, assuming the instances are $(x_1, x_2, \ldots, x_t)$ from a parameterized problem (Q, κ), and

$$l := \max_i \kappa(x_i).$$

In case $\kappa(x_i) = \kappa(x_j)$ for all pairs (i, j), then denote the common parameter by k. Let n denote the length of the longest instance in the input.

Table 2.1: Various compositions

	Parameters Same?	Output Parameter	Length of Output
OR	No	$l^{O(1)}$	—
Linear–OR	No	$l^{O(1)}$	$t \cdot n^{O(1)}$
Composition	Yes	$k^{O(1)}$	—
λ-constant OR	Yes	$n^{O(1)}$	—

Of interest to us is the fact that the existence of a composition algorithm for a parameterized problem along with a polynomial kernel implies a distillation algorithm for the corresponding classical problem. (Likewise, the existence of a Linear OR for a parameterized problem along with a polynomial kernel implies a linear f-distillation algorithm for the corresponding classical problem.) We understand the latter to be highly implausible, and thus it suffices to demonstrate such algorithms to rule out the existence of polynomial kernels for parameterized versions of NP-complete problems.

2.5 Ruling Out Polynomial Kernels

We first show that if the parameterized version of a NP-complete problem admits both a composition and a polynomial kernel, then it also has a distillation. This will

immediately imply that if a parameterized problem has a composition algorithm, then it has no polynomial kernel unless $PH = \Sigma_3^p$, due to Theorem 4.

Recall that a composition is a distillation on parameterized problems, which requires that all the input instances have the same value of the parameter and there is no restriction on the length of the output. The precise definition is the following:

Definition 10. Let (P, κ) be a parameterized problem. A *composition of P* is a polynomial time algorithm $\mathbb{A}$ that receives as inputs finite sequences $\overline{x} = (x_1, \ldots, x_t)$ with $x_i \in \{0,1\}^*$ for $i \in [t]$, such that $\kappa(x_1) = \kappa(x_2) = \cdots = \kappa(x_t)$. Let $k := \kappa(x_1)$. The algorithm is required to output a string $\mathbb{A}(\overline{x}) \in \{0,1\}^*$ such that

1. $\kappa(\mathbb{A}(\overline{x})) = k^{O(1)}$
2. $\mathbb{A}(\overline{x}) \in P$ if and only if for some $i \in [t] : x_i \in P$.

Theorem 6. *Let* (P, k) *be a compositional parameterized problem such that* P *is* NP*-complete. If* P *has a polynomial kernel, then* P *also has a distillation algorithm.*

Proof. Let $x_1, \ldots, x_t$ be instances of P. P is equipped with a polynomial kernelization $\mathbb{K}$, a composition $\mathbb{A}$, and NP-completeness reductions to and from SAT, the problem of deciding the satisfiability of a formula in propositional logic. Let

$$\phi : \{0,1\}^* \to SAT$$

be the reduction to SAT, and let

$$\psi : SAT \to \{0,1\}^*$$

be the reduction from SAT. Let n be the longest instance in the input sequence, i.e., let

$$n := max_{i \in [t]} |x_i|$$

The goal is to get a distillation $\mathcal{D}$ for P. This is obtained in a number of natural steps:

1. First, group together instances that have the same parameter and applying to composition on them, so that the total number of instances is bounded by the largest parameter in the sequence, which in turn is bounded by n.[4]
2. Then apply the kernelization routine on each of the composed instance, to reduce the size of each instance to at most a polynomial in n.
3. Now apply ϕ to convert all instances into boolean formulas.
4. Consider the disjunction of these boolean formulas.
5. Apply ψ to obtain an equivalent instance of P.

For a detailed and formal proof, we refer the reader to [BDFH08]. However, even from the outline above, it is easy to deduce that the composition of the steps above is a distillation, i.e., that it can be performed in polynomial time and that the length of the output is also polynomially bounded in n.

The process is illustrated in Figure 2.1.

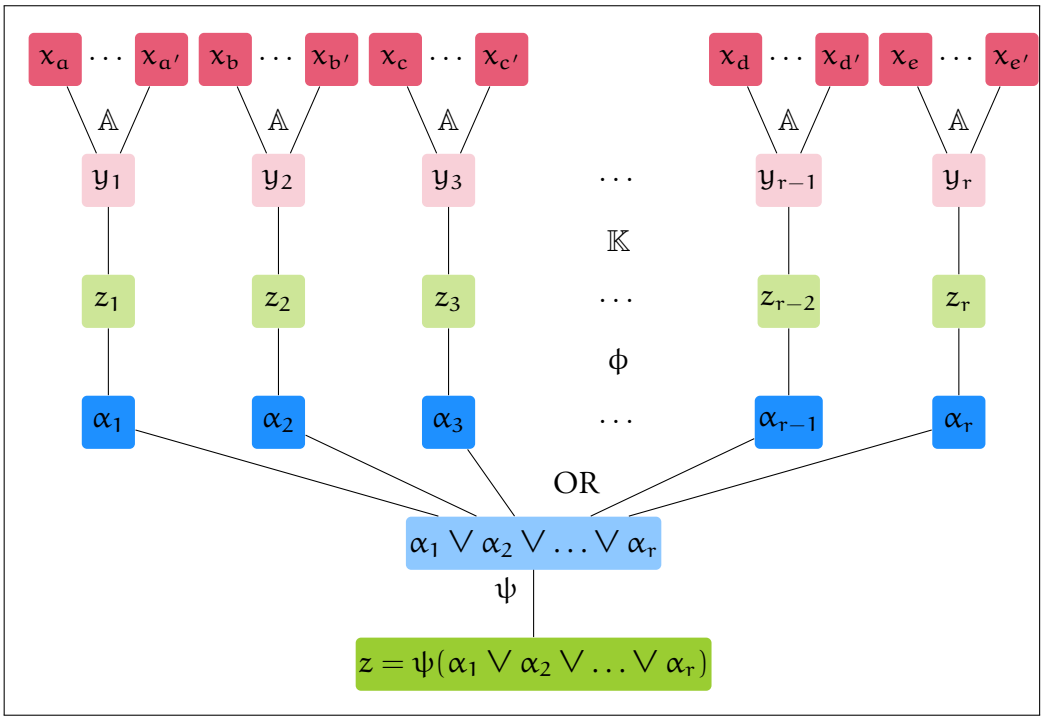


Figure 2.1: Obtaining a distillation for a NP-complete problem using composition and kernelization.

[4] Since we assume that the parameter is specified in unary.

□

2.6 Ruling Out Polynomial Pseudo-Kernels

The stronger result that rules out polynomial pseudo-kernels is actually easier to prove, because we have, in the hypothesis, a linear OR, so we don't have to worry about the instances having different parameters any more.

Recall that a Linear OR is a distillation on parameterized problems with the property that:

The parameter of the output is polynomial in the maximum of all parameters in the sequence, and the length of the output is bounded by the product of a polynomial in the size of the longest instance in its input sequence and the number of instances.

The precise definition is the following:

Definition 11. Let (P, κ) be a parameterized problem. A *linear OR of P* is a polynomial time algorithm $\mathbb{O}$ that receives as inputs finite sequences $\overline{x} = (x_1, \ldots, x_t)$ with $x_i \in \{0,1\}^*$ for $i \in [t]$ and outputs a string $\mathbb{O}(\overline{x}) \in \{0,1\}^*$ such that

1. $\kappa(\mathbb{O}(\overline{x})) = max_{1 \leqslant i \leqslant t} \kappa(x_i)^{O(1)}$
2. $|\mathbb{O}(\overline{x})| = t \cdot (max_{i \in [t]} |x_i|)^{O(1)}$
3. $\mathbb{O}(\overline{x}) \in P$ if and only if for some $i \in [t] : x_i \in P$.

Theorem 7. *Let* (P, k) *be a parameterized problem such that* P *is* NP*-complete. If* P *has a polynomial pseudo-kernel, and a linear OR, then* P *also has a linear* f*-distillation.*

Proof. Let $x_1, \ldots, x_t$ be instances of P. P is equipped with a polynomial pseudo-kernelization, and a linear OR $\mathbb{O}$. Clearly, if P has a pseudo-kernel of size $\kappa(x)^c \times f(n)$, where f is a pseudo-linear function, then it has a linear kernelization $\mathbb{K}$ with respect to the parameter $\kappa^c \times f$. Consider the following distillation algorithm:

$$\mathcal{D}(\overline{x}) = \mathbb{K}(\mathbb{O}(\overline{x}))$$

It is easy to see that

$$|\mathcal{D}(\overline{x})| = O\left(\kappa(\mathbb{O}(\overline{x}))^c \cdot f(|\mathbb{O}(\overline{x})|)\right)$$

But f is pseudo-linear and $\mathbb{O}(\overline{x})$ is only polynomial in the length of the longest instance. Denoting the length of longest instance by n, note that the parameter of an instance that is polynomial in n is at most the entire length of the instance, and hence

$$\kappa(\mathbb{O}(\overline{x}))^c = n^{O(1)}$$

and thus $|\mathcal{D}(\overline{x})|$ is

$$n^{O(1)} \cdot |\mathbb{O}(\overline{x})|^{1-\varepsilon}.$$

It follows that $\mathcal{D}$ is a linear f-distillation. □

2.7 Ruling Out Strong Polynomial Kernels

In this section we provide a mechanism for showing the hardness of obtaining strong polynomial kernels. The flavor of the arguments in this section is somewhat different from what we have seen so far. For every problem with a self-reduction that is parameter decreasing, we would not have strong polynomial kernels - unless $P = NP$. We describe two examples of problems that are of this kind, although it turns out that these problems would not even have polynomial kernels (under the stronger assumption that $PH = \Sigma_3^p$), and are revisited in the next chapter.

Theorem 8. *Let* (Q, κ) *be a parameterized problem (where* Q *is NP-complete) with* $Q(0) := \{x \in Q | \kappa(x) = 0\} \in PTIME$. *Assume that there is a polynomial reduction* R *from* Q *to itself which is parameter decreasing, that is, for all* $x \in Q$,

$$\kappa(R(x)) < \kappa(x).$$

Then,

if (Q, κ) *has a strong polynomial kernelization, then* $Q \in \mathsf{PTIME}$.

Proof. We begin with an instance x. To obtain a polynomial time algorithm to decide if x belongs to Q, the overall strategy is to first apply the strong polynomial kernelization on x, and then apply the self-reduction on the kernel of x. Since the kerenelization is non-increasing, the parameter of the kernel is at most the parameter of the original problem, and it will necessarily decrease after the self reduction. Repeating this $\kappa(x)$ times, at most, should give us an instance in $Q(0)$ in polynomial time. But now we may apply the polynomial time algorithm available for $Q(0)$ to decide this instance, and hence we have a PTIME algorithm for deciding if $x \in Q$.

More precisely, let $\mathbb{K}$ be a strong polynomial kernelization of (Q, κ). Let $\mathcal{A}$ be the composed algorithm $R \circ \mathbb{K}$. Let $\mathcal{B}$ be the polynomial time algorithm for deciding $Q(0)$. Then the following algorithm, $\mathcal{C}$ decides Q in polynomial time. Given an instance x, $\mathcal{C}$ computes $\mathcal{A}(x), \mathcal{A}(\mathcal{A}(x)), \ldots$; since $\mathcal{A}$ is strictly parameter decreasing, after $\kappa(x)$ runs of $\mathcal{A}$, we get an instance y such that $\kappa(y) = 0$. Now $\mathcal{C}$ simulates B on y. The correctness of the algorithm is immediate. Note that $\mathcal{C}$ is polynomial time algorithm, as $\mathcal{A}$ runs in polynomial time on instances whose sizes are at most polynomial in $|x|$ (because of the kernelization that it does first), and $\mathcal{B}$ is given to be polynomial time. □

2.7.1 Examples

Consider the parameterized version of the satisfiability problem:

p-SAT	
Instance:	A propositional formula α in conjunctive normal form.
Parameter:	Number of variables of α.
Question:	Is α satisfiable?

To rule out the possibility of a strong polynomial kernel, we need to describe a routine that reduces an instance of SAT to an equivalent instance with fewer variables. Let α be an instance of SAT, and let x belong to the variables used in α. We simply create two versions of α, one (α_1) in which we substitute for x the constant 0, and the other (α_2) in which we substitute for x the constant 1. It is immediate that α is equivalent to the disjunction $\alpha_1 \vee \alpha_2$. Note that this disjunction has one variable less than α. Also, SAT(0), the collection of satisfiable formulas with no variables, is obviously in PTIME. Hence, by Theorem 8, we observe that SAT is unlikely to admit a strong polynomial kernel.

Further, consider parameterized Pointed Path problem, the parameter being the length of the path:

k-POINTED PATH	
Instance:	A graph $G = (V, E)$, a vertex $v \in V$ and a non-negative integer k.
Parameter:	k.
Question:	Does G have a path of length k starting at v?

We define a parameter-decreasing polynomial reduction R from p-POINTED-PATH to itself as follows: Let (G, v, k) be an instance of p-POINTED-PATH and assume $k \geqslant 3$.

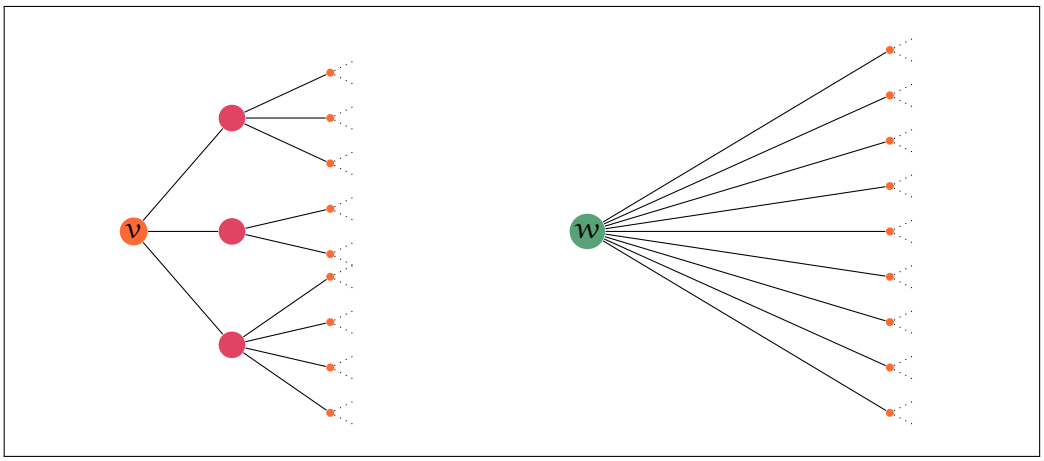


Figure 2.2: Self Reduction for Pointed Path

To obtain a parameter decreasing self-reduction, we delete v and all its neighbors, and connect the all vertices that were at distance 2 from v to a new vertex w. It is easy to see that every k path starting at v in G corresponds to a path of length $(k-1)$ in the reduced graph, starting at w. Therefore, the Pointed Path problem parameterized by the length of the path does not admit a strong polynomial kernel, unless $P = NP$.

2.8 Summary

The mechanism that has been developed so far for proving lower bounds on kernelization is summarized in Table 2.2.

Table 2.2: Various compositions

Size	Positive Evidence	Negative Evidence	Conjecture
$O(1)$	P-time Algorithm	NP–hardness	$P \neq NP$
$k^{O(1)}$	Polynomial Kernelization	Compositionality, Transformations	$PH \neq \Sigma_3^p$
$f(k)$	FPT Algorithm	$W[1]$–hardness	$FPT \neq W[1]$
$n^{1-\varepsilon} \cdot k^{O(1)}$	Pseudo Polynmial Kernelization	Linear OR	$PH \neq \Sigma_3^p$
$k^{O(1)}$, $\kappa(\mathbb{K}(x)) \leqslant \kappa(x)$	Strong Polynomial Kernelization	Parameter–Decreasing Self-Reduction	$P \neq NP$

3. Examples of Compositions

Come together, right now.

– John Lennon

So far, we have examined various notions of algorithms that "compose" problem instances according to some requirements. We also established the implications that some of these algorithms have for kernel lower bounds. In this chapter we prove the (conditional) hardness of obtaining polynomial kernels for a variety of problems, and suggest a standardized framework in which these algorithms may be understood.

3.1 The General Compositional Setup

Algorithms that compose multiple instances of a problem into one (respecting a number of properties) have been developed for a various problems ([BDFH08],[CFM07]). At first sight, they come across as procedures that are hand–crafted for the specific problem at hand, having little or nothing in common. However, more careful examination reveals a strategy that figures in all of these algorithms – either in a very obvious way, or in mild disguise. We begin by describing this "strategy" with the help of some minimal formalism. This will allow us to describe all the algorithms in way that makes the presence of the strategy more apparent.

In general, a compositional algorithm $\mathcal{A}$ for a parameterized language (Q, κ) is required to "merge" instances

$$x_1, x_2, \ldots, x_t,$$

into a single instance x in polynomial time (more precisely, if $n_i := |x_i|$, and $n = \sum n_i$, then the algorithm's runtime is a polynomial in n). The output of the algorithm belongs to $(Q, \kappa(x))$ if and only if there exists at least one $i \in [t]$ for which $x_i \in$

(Q, κ). The algorithm is required to respect constraints on the length of x, and the magnitude of $\kappa(x)$, which differ depending what kind of a composition the algorithm is performing. The two specifics we are interested in are composition algorithms and Linear ORs.

Recall that the length of the output of a *composition algorithm* is unrestrained, short of having to be $n^{O(1)}$ – which is understandable, we do not want the algorithm to run out of time before it can describe its output! A compositional algorithm is also equipped with the promise that $\kappa(x_i) = k$, for all $i \in [t]$. Recall that we require the parameter of the output to be a polynomial in the parameter of the inputs.

The length of the output of a *linear OR*, on the other hand, has to be $t \cdot (\max_{i \in [t]} |x_i|)^{O(1)}$, and the parameter of the output is also required to be polynomial in the largest of the parameters in the inputs.

We now describe some general observations that can be applied to specific problems during the design of such algorithms. We will use $\mathcal{A}$ to refer to a compositional algorithm of either kind (a composition or a linear OR).

The first observation is inspired by the fact that the time available to $\mathcal{A}$ for working things out is more when the number of instances fed to it is larger. We also know that a given instance can be solved completely in FPT time, so we attempt to exploit this. Suppose the membership query "$x \in (Q, \kappa)$?" can be solved in time

$$T_s = 2^{\kappa(x)^c} \cdot |x|^{O(1)}.$$

Notice that the time bound for $\mathcal{A}$ is

$$T_a = n^{O(1)} = (n_1 + \cdots + n_t)^{O(1)}.$$

Let k denote the largest parameter in the input. Observe that if $t > 2^{k^c}$, then we can afford to invest time T_s for each of the instances:

$$\begin{aligned} T &\leqslant 2^{k^c}(n_1^{O(1)} + \cdots + n_t^{O(1)}) \\ &\leqslant t \cdot (n_1 + \cdots + n_t)^{O(1)} \\ &\leqslant n^{O(1)} \end{aligned}$$

Let's take advantage of this: whenever the number of instances is larger than 2^{k^c}, and a FPT algorithm using time $2^{k^c} \cdot n^{O(1)}$ is known, we iterate through the instances, solving each one completely. If we encounter a YES–instance on the way, we stop and return that instance as the output, otherwise, (all the instances are NO–instances) we choose to return any one of them. It is easy to verify that this is a compositional algorithm or a linear OR, as the case may be.

The objective of the exercise is to ensure that we may always be working, without loss of generality, with a bounded number of instances in the input to the compositional algorithm. The longer the time taken by the FPT algorithm used in the argument above, the weaker the bound. Our interests will be restricted[1] to bounds of the form $2^{k^{O(1)}}$, because of ulterior motives that will become clear presently. In particular, all the problems we discuss in this chapter admit FPT algorithms.

At this point, we recall that the parameter of the output is required to be a polynomial in the parameter of the input instances (or the maximum of the parameters in the input instances, in case they are different). This requirement can now be rephrased in a manner that will be useful to us. Note that

$$t \leqslant 2^{k^c} \Rightarrow k^c > \log t,$$

allowing us to say that $\kappa(x)$, the magnitude of the parameter of the output, is allowed to be $(\log t)^{O(1)}$.

It turns out that this is a good way of putting it, because the "$\log(t)$" allowance for new parameter generally is just enough to encode information about the individual instances in the composed instance. Usually, it turns out that $\log(t)$ objects that contribute to the parameter can make their presence felt in $2^{\log(t)}$ different ways in the composed instance, and since this is exactly equal to the number of input instances, we find them useful in finding traces of the original instances in the composed instance. This emerges as a concrete strategy when we work out specific examples.

We describe compositional algorithms for various problems. The description of these algorithms generally revolves around a single function, which we will call ρ :

[1] It is easy to check that the argument that allows us to restrict ourselves to a bounded number of instances to begin with is easily generalized.

$\{0,1\}^* \times \{0,1\}^* \to \{0,1\}^*$, which accepts two instances and returns one. We use ρ in a systematic way over the inputs $(x_1 \ldots x_t)$. It will generally be useful to have in mind the picture of a complete binary tree on t leaves.

We would like to visualize the input instances as plugged in at the leaves of a complete binary tree with t vertices. For convenience, we generally assume $t = 2^l$ ($l \leqslant k^c$). This makes the tree l levels deep. (Such an assumption can usually be easily justified.) We inductively compute the contents of a given node as being $\rho(a, b)$, where a and b are the instances obtained at the children of this node. The output of the composed algorithm is what is obtained at the root of this tree (as shown in the figure below). Given this general framework, let us now put it to work.

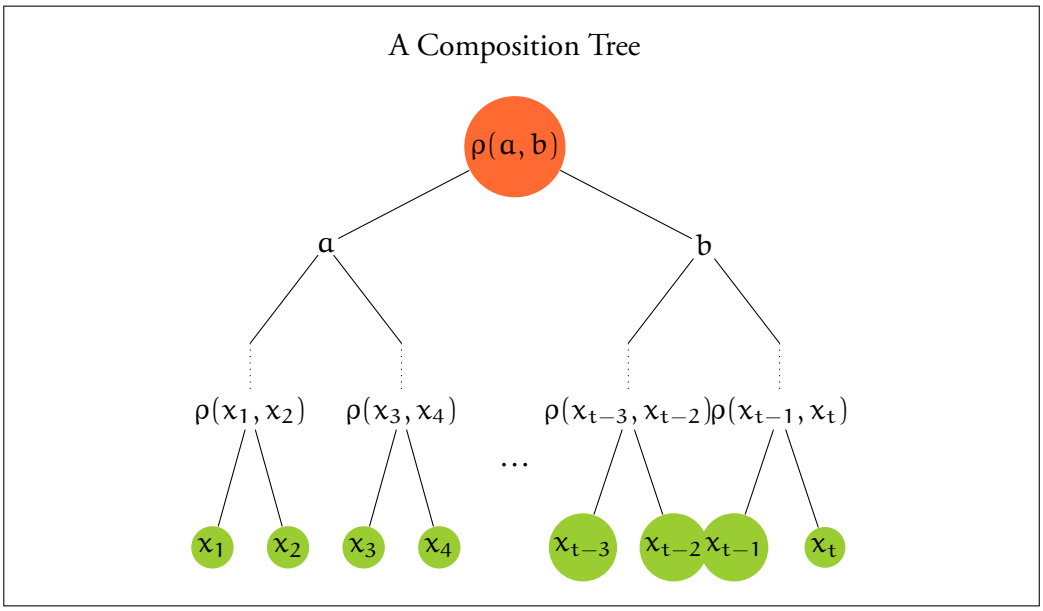


Figure 3.1: General Framework For Composition

3.2 k-path

We begin with a warm–up example; consider the following problem:

k-PATH	
Instance:	A graph G and a non-negative integer k.
Parameter:	k.
Question:	Does G have a path of length k?

This problem is shown to be in FPT via the technique of color coding. The algorithm randomly "colors" the vertex set with k colors (i.e., it picks, uniformly at random, a function $c : V \to [k]$), and hopes that the vertices of the path are colored distinctly (i.e., the function restricted to at least one witness k-length path is injective, these are the successful events). It then uses dynamic programming to actually identify the path from the colored graph. The algorithm is de–randomized using a family of hash functions, and the reader is referred to [AYZ95] for more details.

We present here a composition for k-PATH. Suppose the input instances are:

$$(G_1, k_1), \ldots, (G_1, k_t)$$

A composition algorithm for the case when all the k_i's are the same is trivial, we simply provide the disjoint union of all the graphs as the output.

In the more general case, we pre–process the graphs to ensure that all the parameters are equal – let k be the maximum among the k_i's. For all $i \in [t]$, we add a path of length $(k - k_i)$ to G_i, and make one of the endpoints adjacent to every $v \in V(G_i)$.

Although this is not essential, observe that the FPT algorithm known for k–path allows us to bound the number of instances. Now define $\rho(G, H) = G \uplus H$, and notice that the disjoint union of all graphs has a path of length k if and only if at least one of them does[2].

Notice that the composition takes linear time, and leaves the parameter unchanged.

[2] Since this works for any number of instances, so it was not really necessary to solve every instance completely when we had enough them.

3.3 Disjoint Factors

Let L_k be the alphabet consisting of the letters $\{1, 2, \ldots, k\}$. We denote by L_k^* the set of words on L_k . A factor of a word $w_1 \cdots w_r \in L_k^*$ is a substring $w_i \cdots w_j \in L_k^*$, with $1 \leqslant i < j \leqslant r$, which starts and ends with the same letter, i.e., the factor has length at least two and $w_i = w_j$. A word w has the *disjoint factor property* if one can find disjoint factors $F_1, \ldots, F_k$ in w such that the factor F_i starts and ends by the letter i.

For example, in the word 123235443513 has all the r–factors, $r \in [4]$ - but not as many disjoint factors. It has disjoint 2, 3, and 4 factors, but, for instance, the 5–factor overlaps with with the 4–factor, and the 1–factor overlaps with them all. Of course, other combinations of disjoint factors are attainable from this word, but it is clearly a No–instance of the Disjoint Factors problem.

Observe that the difficulty lies in the fact that the factors F_i do not necessarily appear in increasing order, otherwise detecting them would be computable in $O(n)$, where n is the length of W. The problem is known to be NP–complete ([BDFH08], cf. Appendix C).

We now introduce the parameterized problem Disjoint Factors.

p-Disjoint Factors	
Instance:	A word $w \in L_k^*$.
Parameter:	$k \geqslant 1$.
Question:	Does w have the Disjoint Factors property?

It is immediate that p–Disjoint Factors is FPT. Since the problem can be solved in time that is linear in the length of the word given the ordering of the factors, we simply iterate over all possible orderings of factors - this gives us an algorithm with runtime $k! \cdot n$. However, less obvious is a $2^k \cdot p(n)$ algorithm for the problem – however, it is known (via the technique of Dynamic Programming, see [BDFH08] and Appendix D).

Let $w_1, w_2, \ldots, w_t$ be words in L_k^*. Towards the composition algorithm $\mathcal{A}$, we may begin by assuming that the number of instances is bounded by 2^k, since when

we have more, we may use the FPT algorithm to solve each instance separately. As a matter of convention, we output the first w for which $\mathcal{A}$ succeeds in finding k disjoint factors, and in the event that none of the words have k disjoint factors, $\mathcal{A}$ returns w_t. Clearly, $\mathcal{A}$ is a composition.

When $t < 2^k$, we make the reasonable assumption that $t = 2^l$ for some l. Notice that $l \leqslant k$, and therefore, we observe that we may use at most l^d new letters in the composed instance, for some constant d. It turns out that we will need exactly l new letters, $b_1, \ldots, b_l$. We begin by plugging in the words at the leaves along with b_0 appended to either end:

$$\rho(\mathbb{T}_0(i)) = b_0 w_i b_0.$$

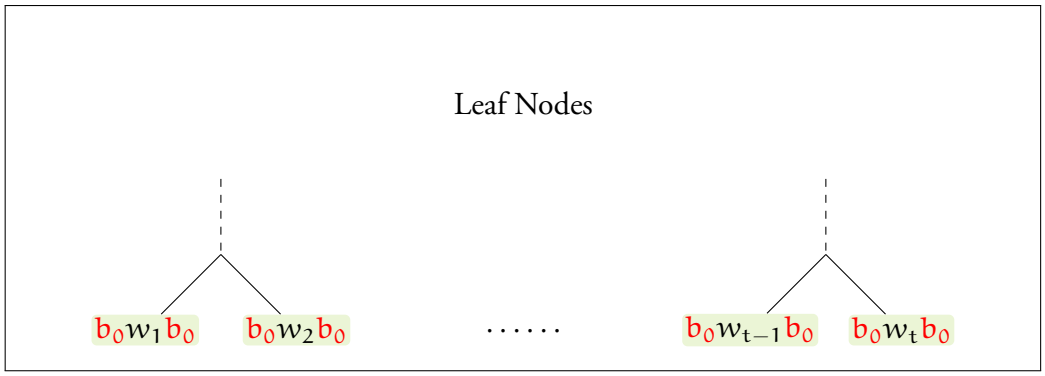


Figure 3.2: Composition for Disjoint Factors: Leaves

Let $\lambda(xa, by)$, where a,b are letters, and x,y are words, denote the word $xaby$ when $a \neq b$ and the word xay when $a = b$. Then we have, at the j^{th} level; $1 \leqslant j < l$:

$$\rho(\mathbb{T}_j(i)) = b_j \;\; \lambda(\rho(\mathbb{T}_{j-1}(2i-1)), \rho(\mathbb{T}_{j-1}(2i))) \;\; b_j.$$

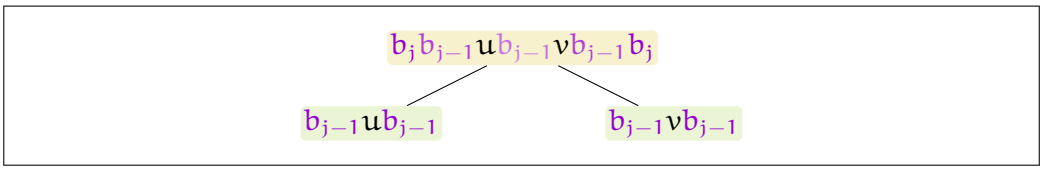
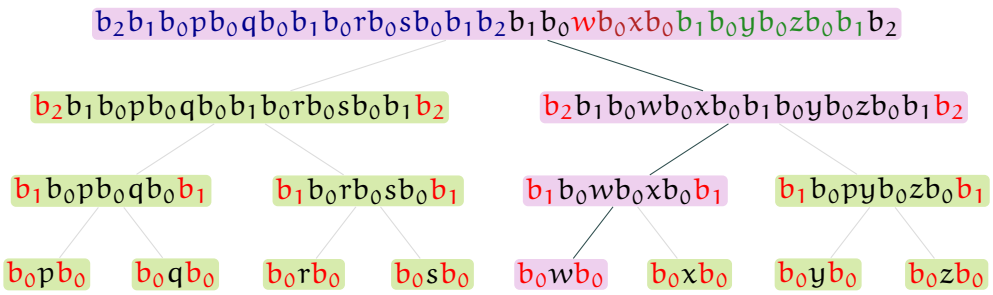


Figure 3.3: Disjoint Factors: ρ

At the root, we perform a simple concatenation:

$$\mathbb{T}_l(1) = \lambda(\mathbb{T}_{l-1}(1), \mathbb{T}_{l-1}(2)) = w$$

Evidently, if $\mathbb{T}_l(1))$ has the disjoint factors property, then we are forced to pull out the $1, 2, \ldots, k$ disjoint factors from exactly one of the original words.



An example with

$$\{p, q, r, s, w, x, y, z\}$$

as input words. Notice how a solution in the root can be traced to some unique solution in the leaves and conversely! (Warning: beware of confusing this w with the w we have been using to denote the composed instance – they are different.)

Figure 3.4: Disjoint Factors Composition: An Example

We argue this in the forward direction first: suppose one of the w_i's has the disjoint factors property. We want to show that the composed instance also has the disjoint factors property (over the extended alphabet $[k] \cup \{b_0, \ldots b_{l-1}\}$, observe that:

The $1, 2, \ldots, k$ disjoint factors may be obtained from w_i.

Note that w_i is a substring of one of the b_{l-1} factors. For b_{l-1}, therefore, we choose that b_{l-1} factor that does not overlap with w_i.

Further, there are exactly three b_{l-2} factors that do not overlap with w_i, and two of these overlap with the chosen b_{l-1} factor, so there is one b_{l-2} factor that does not overlap with any of the factors chosen thus far, and we use this as the b_{l-2} factor.

This process can be continued till all the b_i factors are obtained, indeed, we will always have one for every $0 \leqslant i \leqslant l - 1$ by construction.

On the other hand, if the composed instance has the disjoint factors property, then we would like to derive that all the $1, 2, \ldots, k$ factors are substrings of w_i for some i. It is easy to observe that if we delete all the b_i factors from the word w, then the remaining word contains exactly one of the w_i's as a substring. For example, notice that the b_{l-1} factor overlaps with half of the w_i's, and the b_{l-2} factor overlaps with half of the remaining w_i's, and so on. Thus, once the b_i factors have been accounted for, only one possible source remains for the $1, 2, \ldots k$ factors – necessarily one of the w_i's. But then w_i must exhibit (by definition) the disjoint factors property over the restricted alphabet $[k]$, and hence we are done.

Note that the parameter of the composed instance is $k + l$, where $l \leqslant k$. Thus the size of the alphabet of the composed instance is at most twice the size of the original alphabet. It is easy to see that the algorithm runs in polynomial time, as the operations at every node would consume only constant time.

3.4 p–SAT

We use p–SAT to denote the following parameterized version of SAT:

p-SAT	
Instance:	A propositional formula α in conjunctive normal form.
Parameter:	Number of variables of α.
Question:	Is α satisfiable?

The problem is evidently in FPT, as an algorithm merely has to iterate over all possible assignments of the variables, there being 2^n of them. The runtime is $2^n \cdot m$, where m is the length of the formula.

We now describe a Linear OR $\mathcal{A}$ for p–SAT. The algorithm is along the lines of [CFM07], although we deviate from their presentation slightly, in the interest of placing the algorithm in the present context. Let $\alpha_1, \alpha_2, \ldots, \alpha_t$ be CNF formulas. Further, let n_i denote the number of variables in α_i, and let

$$n := \max_{i \in [n]} n_i \quad \text{and} \quad m := \max_{i \in [t]} |\alpha_i|.$$

The Linear OR is required to construct a formula γ which is satisfiable if and only if there exists at least one $i \in [t]$ such that α_i is satisfiable, such that the length of γ is $t.n^c$ for some constant c, and the number of variables in γ is bounded by n^d for some constant d.

As before, observe that $\mathcal{A}$ can handle 2^n instances (or more) by examining all possible assignments for every formula. We assume the convention that the output is the first α_i for which $\mathcal{A}$ discovers a satisfying assignment, and in the event that none of the formulas are satisfiable, then $\mathcal{A}$ returns α_t. Clearly, $\mathcal{A}$ is a Linear OR.

When $t < 2^n$, we make the reasonable[3] assumption that $t = 2^l$ for some l. Notice that $l \leqslant n$, and therefore, we may translate the constraint on the number of variables in γ to l^d. Let $\mathbb{T}$ be the complete binary tree with t leaves. We use $\mathbb{T}_j(i)$ to refer to the i^{th} node at the j^{th} level of the tree, where the leaves are considered to be at level 0. We will need $b_1, \ldots, b_l$ —these are new variables that do not occur in any of the α_is.

The linear OR will append a sensible[4] disjunction of l literals picked from $b_1, \ldots, b_l$ and $\neg b_1, \ldots, \neg b_l$ to all the disjuncts of α_i, in a way that we associate an unique combination of literals with every α_i.

If $\pi = p_1 \wedge \ldots \wedge p_k$, then let $\mathcal{C}(\pi, i)$ denote p_i. We define ρ on the leaves as follows:

[3] If t is not an exact power of two, we let l be the smallest integer for which $t < 2^l$, and duplicate one of the given formulas $2^l - t$ times.

[4] This refers to the fact that b and $\neg b$ do not appear together as disjuncts.

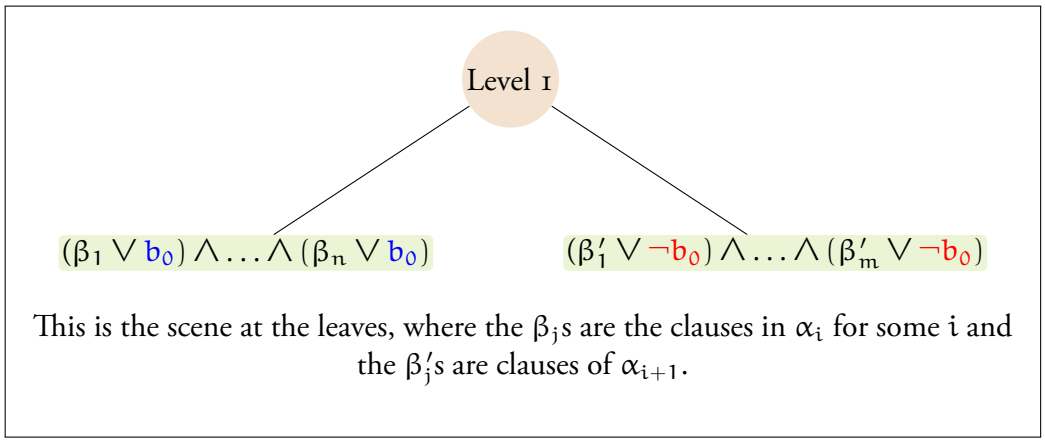


Figure 3.5: p–SAT Composition: Leaves

If i is even, then $\rho(\mathbb{T}_0(i)) = \bigwedge_k(\mathcal{C}(\alpha_i, k) \vee b_0)$.

If i is odd, then $\rho(\mathbb{T}_0(i)) = \bigwedge_k(\mathcal{C}(\alpha_i, k) \vee \neg b_0)$.

Further up at the j^{th} level, ρ is defined using the formulas at the $(j-1)^{th}$ level. Let $\beta_L(i)$ denote $\rho(\mathbb{T}_{j-1}(2i-1))$ and $\beta_R(i)$ denote $\rho(\mathbb{T}_{j-1}(2i))$

If i is even, then $\rho(\mathbb{T}_j(i)) = \bigwedge_k(\mathcal{C}(\beta_L(i), k) \vee b_j) \bigwedge \bigwedge_k(\mathcal{C}(\beta_R(i), k) \vee b_j)$.

If i is odd, then $\rho(\mathbb{T}_j(i)) = \bigwedge_k(\mathcal{C}(\beta_L(i), k) \vee \neg b_j) \bigwedge \bigwedge_k(\mathcal{C}(\beta_R(i), k) \vee \neg b_j)$.

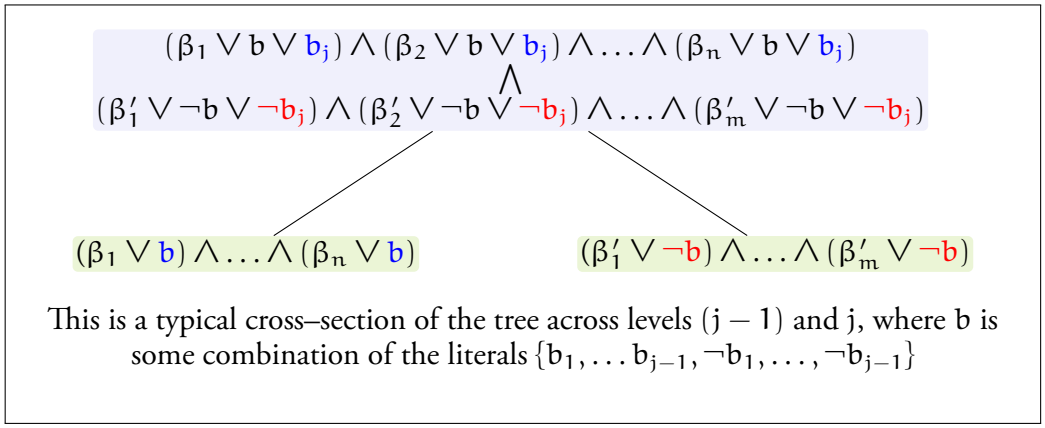


Figure 3.6: p–SAT: The ρ operation.

Let $\gamma := \mathbb{T}_l(1)$. We now argue why γ is satisfiable if and only if at least one of the α_i's has a satisfying assignment.

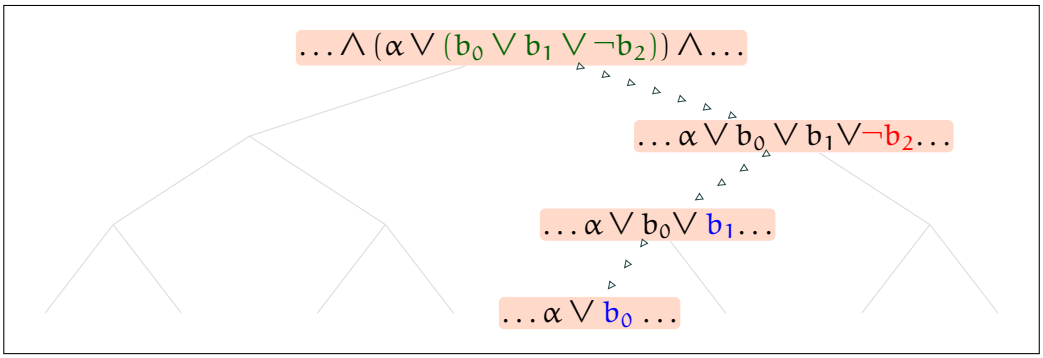


Figure 3.7: p–SAT composition: An Example

Let $(x_1, \ldots, x_N, y_1, \ldots, y_l)$ be an assignment on the variables of the α_i's, and b_j's that satisfies γ. We show that the assignment restriced to the variables of the α_i's satisfies at least one fo the α_i's.

Suppose not. This implies that the assignment of $(x_1, \ldots, x_N)$ to the variables of the α_i's fails to satisfy any of the α_i's. Note in the composed instance, all possible combinations of the b–literals need to be satisfied if none of the orignal formulas are satisfiable. Further observe that no assignment to the b_j's can have the property of satisfying all possible disjunctions over the b_js. Indeed, it is easy to construct a clause of all the b–literals that will remain unsatisfied for any given assignment ν – if $\nu(b_i) = 0$, then we add the literal $\neg b_i$ to the clause, and if $\nu(b_i) = 1$, then we add the literal b_i to the clause. Thus γ is not satisfiable, contradicting our assumption.

In the converse, let i be the smallest index for which α_i is satisfiable. Let $b(j)$ denote the value of the j^{th} bit in the usual binary representation of i. Then it's easy to check that the assignment of $(1 - b(i))$ to b_i is satisfies all the clauses in γ that do not involve clauses from α_i, and clauses involving α_i can be satisfied by using the assignment that satisfies α_i.

Note that we have increased the parameter by $l \leqslant k$, and the algorithm runs in polynomial time as the node-operations can be performed in constant time.

3.5 SAT Parameterized by the Weight of the Assignment

We use pw–SAT to denote the following parameterized version of SAT:

pw-SAT	
Instance:	A propositional formula α in conjunctive normal form, with clause length bounded by some constant c, and a non-negative integer k.
Parameter:	k.
Question:	Is there a satisfying assignment for α with weight at most k?

The *weight* of an assignment is the number of 1's in it. This version of SAT is FPT (as long as the lengths of the clauses are bounded). For example, when the clauses have at most 2 literals, a 2^k–sized search tree suffices: as long as there are clauses with only positive literals, take an arbitrary one of these clauses and branch into two cases, setting either of the variables true (at least one of the variables must be true). In each branch, we simplify the formula appropriately, after which, a formula remains where every clause has at least one negative literal and this instance is trivial to solve. This method generalizes to arbitrary constant sized clauses.

In the following, we describe an argument which shows the problem to be compositional.

Let the input instances be:

$$(\alpha_1, k), (\alpha_2, k), \ldots, (\alpha_t, k)$$

Let α be the formula obtained after performing the linear OR from the previous section (however, using the FPT algorithm for pw–SAT to bound the number of instances) on the instances $(\alpha_1, \alpha_2, \ldots, \alpha_t)$. Consider β, defined as follows:

$$\begin{gathered}\alpha \\ \wedge \\ (\neg c_0 \vee \neg b_0) \wedge (c_0 \vee b_0) \wedge \\ (\neg c_1 \vee \neg b_1) \wedge (c_1 \vee b_1) \wedge \\ \cdots \\ (\neg c_{l-1} \vee \neg b_{l-1}) \wedge (c_{l-1} \vee b_{l-1})\end{gathered}$$

This formula introduces $l \leqslant k$ new variables and essentially forces the following implications:

$$\begin{aligned} c_i &\Rightarrow \neg b_i \\ \neg c_i &\Rightarrow b_i \end{aligned}$$

for all $1 \leqslant i \leqslant (l-1)$. It is easy to see that the total weight of any satisfying assignment restricted to the b's and c's is exactly equal to k (note that b_i and c_i can never have the same value in any satisfying assignment).

Clearly, if any of the α_i's have a weight k satisfying assignment, then it can be extended to a weight $k+l$ satisfying assignment for α. In the converse, if α has a satisfying assignment of weight $k+l$, note that the assignment restricted to the variables $\{b_i \mid 0 \leqslant i \leqslant (l-1)\}$ and $\{c_i \mid 0 \leqslant i \leqslant (l-1)\}$ has weight exactly l and it follows that one of the α_i's must admit a weight–at–most k satisfying assignment. The parameter of the composed instance is $k+l$, and the algorithm runs in polynomial time for the same reasons as before.

Note that we may also obtain a Linear OR for pw–SAT along similar lines. To begin with, let the instances be:

$$(\alpha_1, k_1), (\alpha_2, k_2), \ldots, (\alpha_t, k_t)$$

We use k to denote the largest parameter among the inputs, that is,

$$k := max_{1 \leqslant i \leqslant t} k_i$$

Now consider the modified instance set:

$$(\alpha'_1, k), (\alpha'_2, k), \ldots, (\alpha'_t, k)$$

where α'_i is obtained by adding $k - k_i$ new variables as clauses:

$$\alpha'_i = \alpha_i \wedge z_1 \wedge z_2 \wedge \ldots \wedge z_{k-k_i}.$$

It is easy to check that the composition algorithm discussed in this section, along with the modifications described above, gives us a Linear OR.

3.6 Colored Red–Blue Dominating Set

The Red-Blue Dominating Set problem is the following:

Red-Blue Dominating Set (RBDS)	
Instance:	A bipartite graph $G = (T \cup N, E)$ and a non–negative integer k
Parameter:	$k + \lvert T \rvert$.
Question:	Does there exist a vertex set $N' \subseteq N$ of size at most k such that every vertex in T has at least one neighbor in N'?

An instance of RBDS comprises of a bipartite graph $G = (T \cup N, E)$ and an integer k. We ask whether there exists a vertex set $N' \subseteq N$ of size at most k such that every vertex in T has at least one neighbor in N'. The problem is parameterized by $k + |T|$.

In the literature, RBDS is usually known under the name "Red-Blue Dominating Set"and the sets T and N are called "blue vertices" and "red vertices", respectively. Here, we call the vertices "terminals" and "nonterminals" in order to avoid confusion with the colored version of the problem that we are going to introduce. RBDS is equivalent to Set Cover and Hitting Set and is, therefore, NP–complete [GJ79]. Our interest is in showing the hardness of polynomial kernelization for this problem - however, we do not know of a composition for it so far. However, we will eventually

provide a way reducing to this problem from a closely related variant. The notion of reductions will be the subject of the next chapter, we presently describe the variant and show that it is compositional ([DLS09]).

Consider the colored version of RBDS, denoted by Colored Red-Blue Dominating Set (Col-RBDS):

Colored Red-Blue Dominating Set (Col-RBDS)	
Instance:	A bipartite graph $G = (T \cup N, E)$ and a non–negative integer k and a function $col : N \to \{1, \ldots, k\}$.
Parameter:	$k + \lvert T \rvert$.
Question:	Does there exist a vertex set $N' \subseteq N$ of size at most k such that every vertex in T has at least one neighbor in N' and N' has exactly one vertex of each color?

Here, the vertices of N are colored with colors chosen from $\{1, \ldots, k\}$, that is, we are additionally given a function $col : N \to \{1, \ldots, k\}$, and N' is required to contain exactly one vertex of each color. The parameter is again $k + |T|$.

We describe a composition for on a sequence

$$(G_1 = (T_1 \cup N_1, E_1), k, col_1), \ldots, (G_t = (T_t \cup N_t, E_t), k, col_t)$$

of Col-RBDS instances with $|T_1| = |T_2| = \ldots = |T_t| = p$.

We assume that we are dealing with at most $2^{(p+k)}$ instances. This is reasonable because of a FPT algorithm for the problem ([DLS09]) that is described in the next chapter. Again, for ease of description, we let $t = 2^l$, using extra trivial no–instances whenever necessary to achieve this. Begin by plugging in the instances at the leaves of $\mathbb{T}$, which is the complete binary tree on t leaves. Recall that we use $\mathbb{T}_j(i)$ to refer to the i^{th} node at the j^{th} level of the tree, where the leaves are considered to be at level 0..

We use $\diamondsuit(u, v)$ to indicate the operation of identifying two vertices u and v in the graph, which intuitively amounts to merging them into a common vertex and using the union of their neighborhoods as the neighborhood of the common vertex.

We will need to expand the set of terminal vertices as we climb up the tree (just like we needed more variables to expand on the disjuncts in the formulas when we composed p–SAT). We will need $2l \cdot (k-1)$ new vertices:

$$\bigstar = \{u_1(r), u_2(r), \ldots u_l(r), v_1(r), v_2(r), \ldots v_l(r)\}$$

for all $1 \leqslant r < k$.

Denote by $\lambda(G, i)$ the graph that is obtained from G after adding vertices $\{u_i(r), v_i(r)\}$, $(1 \leqslant r < k)$ to the terminal vertex set of G, and the following edges to the edge set:

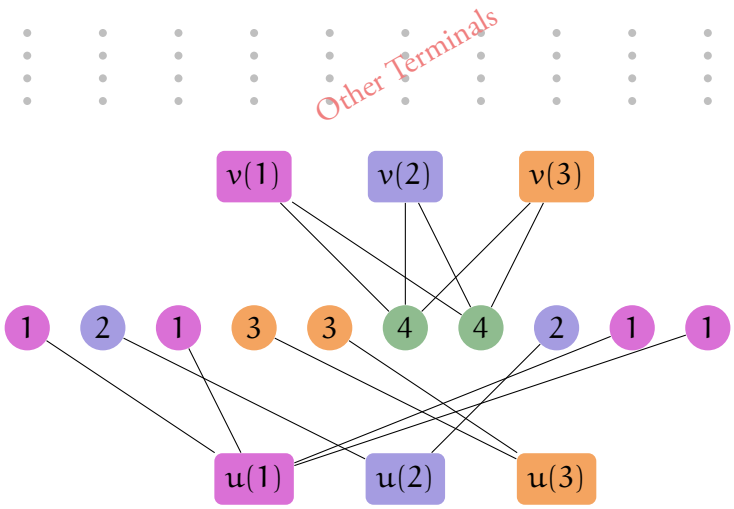
$$E_1(G, i) = \{(x, u_i(r)) \mid x \in V(G), col(x) = r\}, 1 \leqslant r < k$$
$$E_2(G, i) = \{(y, v_i(r)) \mid x \in V(G), col(y) = k\}, 1 \leqslant r < k$$

Note that all vertices that have color $1 \leqslant r < k$ are adjacent to $u_i(r)$ and all vertices with color k are adjacent to $v_i(r)$, for all r. We similarly define $\overline{\lambda}(G, i)$ – the vertex set is expanded by $\{u_i(r), v_i(r)\}$, $(1 \leqslant r < k)$, and the new edges are:

$$E_1(G, i) = \{(x, v_i(r)) \mid x \in V(G), col(x) = r\}, 1 \leqslant r < k$$
$$E_2(G, i) = \{(y, u_i(r)) \mid x \in V(G), col(y) = k\}, 1 \leqslant r < k$$

Here, all vertices that have color $1 \leqslant r < k$ are adjacent to $v_i(r)$ and all vertices with color k are adjacent to $u_i(r)$., for all r. If G is specified at $T \cup N$, then note that the new vertices added by either λ or $\overline{\lambda}$ belong to T (and hence do not need a coloring.)

Further, we will also need a merge function μ, which allows us to identify the terminal vertices. Let $G_1 = (T_1 \cup N_1, E_1)$ and $G_2 = (T_2 \cup N_2, E_2)$. We assume that the set of terminals which do not belong to $\bigstar$ are ordered in some fashion. We denote by t_i, the i^{th} terminal vertex of T_1, and by t'_i, the i^{th} terminal vertex of T_2. Similarly, let u_i, v_i denote vertices from $\bigstar$ in G_1, and let u'_i, v'_i denote vertices from $\bigstar$ in G_2. Let j be the largest index for which $\{u_i, v_i\} \subset G_1$, and j' the largest index for which $\{u'_i, v'_i\} \subset G_2$. Then μ is well–defined when $j = j'$:



An illustration of the application of λ to a graph. The subscripts on the u and v vertices are omitted for clarity, as are the edges between the terminals and the terminals and non-terminals. Note that a similar picture for $\overline{\lambda}$ would be identical with the u and v labels flipped.

Figure 3.8: Composition of col-RBDS: Application of λ

$$\mu(G_1, G_2) = H,$$

where $V(H)$ is the set:

$$\begin{aligned} &\{v \mid v \in N_1 \cup N_2\} \\ \cup\ &\{u \mid u = \diamondsuit(t_i, t_i'), 1 \leqslant i \leqslant p\} \\ \cup\ &\{w \mid w = \diamondsuit(u_i(r), u_i'(r)), 1 \leqslant i \leqslant j, 1 \leqslant r < k\} \\ \cup\ &\{w \mid w = \diamondsuit(v_i(r), v_i'(r)), 1 \leqslant i \leqslant j, 1 \leqslant r < k\} \end{aligned}$$

We are essentially merging all the original terminal vertices and the $\bigstar - -$vertices appropriately. The set of edges is naturally defined as the union of the edge sets of the graphs G_1 and G_2, modified suitably by the $\diamondsuit$ operation. This is illustrated below:

We are finally ready to describe ρ. At the leaves, we have:

If i is even, then: $\rho(\mathbb{T}_0(i)) = \lambda(G_i, 0)$

If i is odd, then: $\rho(\mathbb{T}_0(i)) = \overline{\lambda}(G_i, 0)$

At the j^{th} level, $1 \leqslant j \leqslant (l-1)$, we have:

If i is even, then $\rho(\mathbb{T}_j(i)) = \lambda(\mu(\mathbb{T}_{j-1}(2i-1), \mathbb{T}_{j-1}(2i)), j)$.

If i is odd, then $\rho(\mathbb{T}_j(i)) = \overline{\lambda}(\mu(\mathbb{T}_{j-1}(2i-1), \mathbb{T}_{j-1}(2i)), j)$.

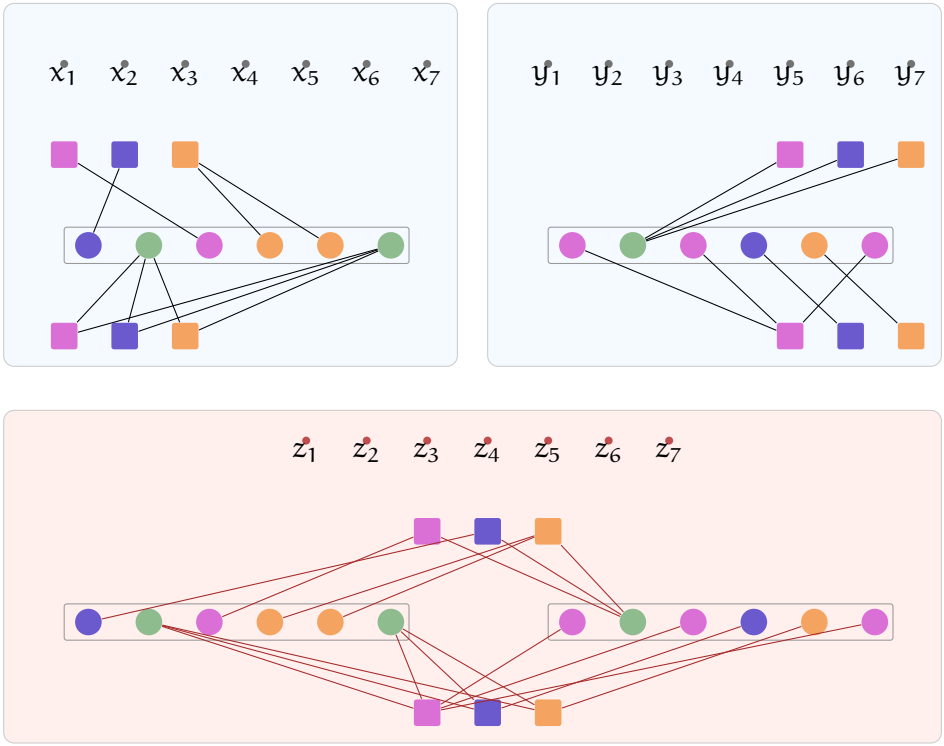
Finally, at the root, we simply merge the graphs at the two children:

$$\rho(\mathbb{T}_l(1)) = \mu(\mathbb{T}_{l-1}(1), \mathbb{T}_{l-1}(2)).$$

We claim that $\mathcal{A}$, given by:

$$\mathcal{A}(G_1, G_2, \ldots, G_t) = G = \rho(\mathbb{T}_l(1)),$$

is a composition for Col-RBDS. Clearly, if G_i is a Yes–instance of Col-RBDS, then all the terminals from $\bigstar$ are covered (to see this, we follow the path from the



Above is an example application of μ to two graphs. Note that the x_i's get identified with the y_i's (the merge of x_i and y_i is labelled z_i in the graph obtained), and the vertices from $\bigstar$ are also identified. For the non-terminals, we take a simple union.

Figure 3.9: Composing col-RBDS

root of the tree to the leaf that has $\lambda(G_i)$, and note that at every level, the vertices of $\bigstar$ introduced into the graph are adjacent to one of the vertices in the solution of G_i, that is, the set of k vertices that dominate all the original terminals of G_i).

In the converse, suppose G has k non–terminals which dominate all the terminals in G. We argue that all these must be k vertices from exactly one of the G_i's – and then it is immediate that they must dominate all the terminals of G_i (if not, they would not be able to dominate all the terminals of G). Let the set of vertices that dominate all the terminals of G be S, and let

$$S = d_1, d_2, \ldots, d_k$$

where d_i is the vertex for which $\mathtt{col}(d_i) = i$. Suppose the vertices of S did not belong to exactly one of the G_i's. Let us further assume (wlog) that the vertices of S come from two graphs, G_i and G_j. Let $d_k \in G_j$, and pick any $d_r \in G_i$. Consider the binary representation of i. Notice that d_r is adjacent to all the $u_p(r)$–vertices in $\bigstar$ whenever the p^{th} bit of the binary representation of i is 1, and it is adjacent to all the $v_q(r)$–vertices in $\bigstar$ whenever the q^{th} bit of the binary representation of i is 0. Therefore, the vertices

$$\{u_q(r) \mid q^{th} \text{ bit in the binary representation of } i \text{ is } 0.\}$$

and

$$\{u_q(r) \mid q^{th} \text{ bit in the binary representation of } i \text{ is } 0.\}$$

are not dominated by d_r, and the only vertices that are adjacent to them are vertices whose color is either r or k. Since we do not have more than one vertex colored r in S, we must turn to the vertex colored k. But the only vertices colored k that can dominate all the vertices above belong to G_i, and unfortunately the d_k in S is from G_j, $j \neq i$, and thus it is impossible to dominate all the terminals that come from $\bigstar$ if S is split across more than one graph.

Thus we have that Col-RBDS parameterized by $(|T|, k)$ is compositional.

3.7 Summary

In this chapter, we have shown compositions and linear ORs for a number of problems. We summarize them in the lemmas below, and their implications on the existance of polynomial kernels and pseudo–kernels in theorems thereafter.

Lemma 1. *The following problems admit Linear ORs:*

1. k–PATH.

   Given a graph G *and an integer* k*, does there exist a path of length* k*? The problem is parameterized by* k*, the length of the path.*

2. p–SAT

   Given a propositional formula α*, is it satisfiable? The problem is parameterized by the number of variables in* α*.*

Lemma 2. *The following problems admit composition algorithms:*

1. p–DISJOINT FACTORS

   Given a word w *over an alphabet of size* k*, does* w *have the Disjoint Factors property? The problem is parameterized by* k*, the size of the alphabet.*

2. pw–SAT

   Given a propositional formula α *and an integer* k*, does there exist a satisfying assignment of weight at most* k*? The problem is parameterized by* k*, the weight of the assignment.*

3. COL-RBDS

   Given a bipartite graph $G = (T \cup N, E)$*, an integer* k*, and a function col* $: N \to \{1, \ldots, k\}$*, does there exist a vertex set* $N' \subseteq N$ *of size at most* k *such that every vertex in* T *has at least one neighbor in* N' *such that* N' *has exactly one vertex of each color? The problem is parameterized by* $k + |T|$*, the size of the solution and the number of terminals.*

By Lemma 1, and Theorems 5 and 7, we now have the following:

Theorem 9. *The following problems do not admit polynomial pseudo–kernels unless* $\mathsf{PH} = \Sigma_3^p$.

1. k–PATH.

   Given a graph G *and an integer* k*, does there exist a path of length* k*? The problem is parameterized by* k*, the length of the path.*

2. p–SAT

   Given a propositional formula α*, is it satisfiable? The problem is parameterized by the number of variables in* α*.*

By Lemma 2, and Theorems 4 and 6, we now have the following:

Theorem 10. *The following problems do not admit polynomial kernels unless* $\mathsf{PH} = \Sigma_3^p$.

1. p–Disjoint Factors

   Given a word w *over an alphabet of size* k*, does* w *have the Disjoint Factors property? The problem is parameterized by* k*, the size of the alphabet.*

2. pw–SAT

   Given a propositional formula α *and an integer* k*, does there exist a satisfying assignment of weight at most* k*? The problem is parameterized by* k*, the weight of the assignment.*

3. Col-RBDS

   Given a bipartite graph $G = (T \cup N, E)$*, an integer* k*, and a function* $col : N \to \{1, \ldots, k\}$*, does there exist a vertex set* $N' \subseteq N$ *of size at most* k *such that every vertex in* T *has at least one neighbor in* N' *such that* N' *has exactly one vertex of each color? The problem is parameterized by* $k + |T|$*, the size of the solution and the number of terminals.*

When the only tool you own is a hammer,
every problem begins to resemble a nail.

Abraham Maslow

In this chapter, we introduce the notion of (polynomial time and parameter) transformations, which will allow us to prove results for problems that do not obviously have compositionality.

4.1 Philosophy and Definition

We begin by describing what we mean by a polynomial time and parameter reduction[1] ([BDFH08]).

Definition 12 (Polynomial time and parameter reduction)**.** Let (P, κ) and (Q, γ) be parameterized problems. We say that P is polynomial time and parameter reducible to Q, written $P \preccurlyeq_{\mathrm{ppt}} Q$, if there exists a polynomial time computable function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$, and a polynomial $p : \mathbb{N} \to \mathbb{N}$, and for all $x \in \{0,1\}^*$ and $k \in \mathbb{N}$ if $f(x) = y$, then the following hold:

1. $x \in P$, if and only if $y \in Q$, and
2. $\gamma(y) \leqslant p(\kappa(x))$

We call f a polynomial time and parameter transformation from P to Q.

This is reminiscent of the notion of fixed parameter reductions introduced by Downey and Fellows (see [DFst, DF95b, DF95a]). Recall that these are defined as follows:

Definition 13 (Fixed-parameter Tractable reductions)**.** Let (P, κ) and (Q, γ) be parameterized problems. An *fpt* reduction is a function $f : \{0,1\}^* \longrightarrow \{0,1\}^*$ such that:

[1] Note that the word *polynomial* is meant to be an adjective to both 'time' and 'parameter'.

1. $x \in P$, if and only if $f(x) \in Q$,

2. f is computable by an fpt-algorithm (with respect to κ), that is, there is a computable function g and a polynomial p such that f is computable in time $f(\kappa(x)) \cdot p(|x|)$, and

3. There is a computable function $h : \mathbb{N} \to \mathbb{N}$ such that $\gamma(f(x)) \leqslant h(\kappa(x))$.

The notion of reductions are generally popular as they are at the heart of most known completeness results. Just as the notion of Cook [Coo71] and Karp [Kar72] reductions are useful in showing the classical hardness of various problems, the analogous notion of fixed parameter reductions are used to show the hardness of parameterized problems.

Notice that not all fixed parameter reductions are polynomial time and parameter reductions, as the latter has the additional requirement that the parameter of the Q–instance obtained by the reduction has to be a polynomial in the parameter of the P–instance that we started out with. In fixed parameter reductions, however, there is no constrained imposed on the parameter.

Our motives for defining polynomial time and parameter reductions have a somewhat different flavor, as will become clear from the theorem below.

Theorem 11. *Let* (P, κ) *and* (Q, γ) *be parameterized problems such that* P *is* NP*-complete, and* $Q \in NP$. *Suppose that* f *is a polynomial time and parameter transformation from* P *to* Q. *Then, if* Q *has a polynomial kernel, then* P *has a polynomial kernel.*

Proof. Suppose that Q has a polynomial kernel. Now, consider the following algorithm, that gets as input a pair $(x, k) \in \{0, 1\}^*$, which is an input for P. First, we compute $f(x)$, say $f(x) = y$. Then, we apply the polynomial kernelization algorithm (say $\mathbb{K}$) for Q to y, suppose this gives $\mathbb{K}(y)$. As P is NP-complete, there is a polynomial time transformation from Q to P, say λ. Suppose $\lambda(\mathbb{K}(y)) = z$. We claim that the algorithm that transforms x to z is a polynomial kernel for P.

Note that $\gamma(y)$ is polynomially bounded in $\kappa(x)$, as f is a polynomial time and parameter transformation. Now, $\gamma(\mathbb{K}(y))$ and the size of $\mathbb{K}(y)$ are polynomially

bounded in $\gamma(y)$ as these are obtained by a polynomial kernelization algorithm, and thus $\gamma(\mathbb{K}(y))$ is polynomially bounded in $\kappa(x)$. As λ is a polynomial time transformation, the size of z is polynomially bounded in $\mathbb{K}(y)$, and hence is a polynomial in $\kappa(x)$.

It is easy to see that the algorithm uses polynomial time, and that

$$x \in P, \text{ if and only if } y \in Q$$

This is illustrated in Figure 6.2.

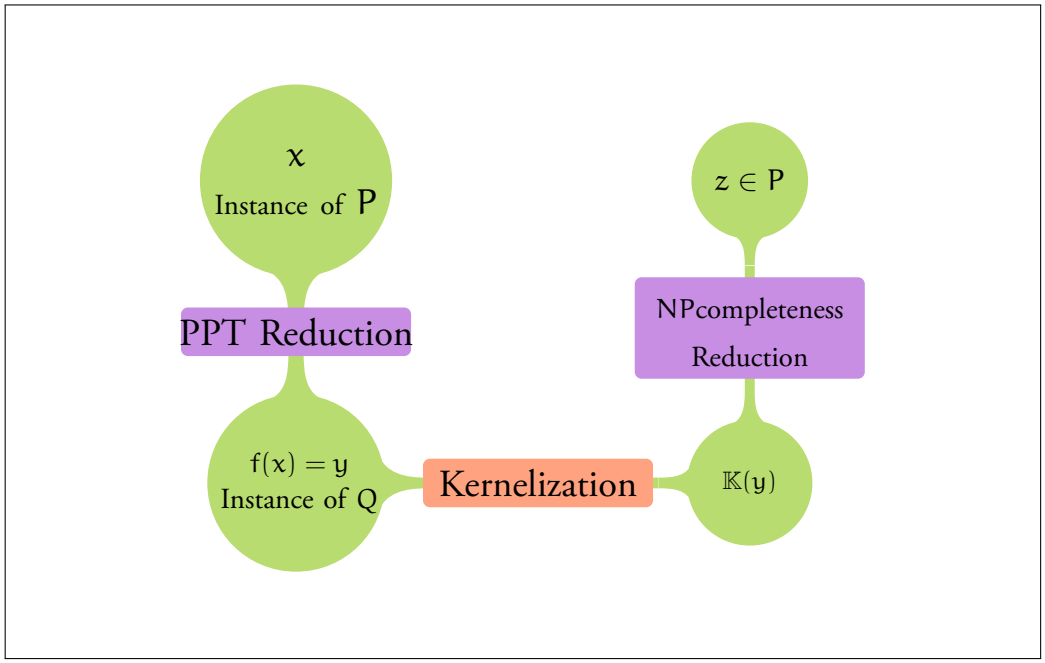


Figure 4.1: Illustrating the utility of PPT Reductions in Kernelization

□

As an easy corollary of Theorem 11, note that whenever (P, κ) and (Q, γ) are parameterized problems (such that P is NP-complete, and $Q \in NP$) and $P \preccurlyeq_{ppt} Q$, if P is compositional, then Q does not have a polynomial kernel unless $PH = \Sigma_p^3$. This turns out to be an extremely useful observation in the context of showing hardness of efficient kernelization. A natural strategy suggests itself: to prove that a problem P is

unlikely to admit a polynomial kernel, we reduce some NP–complete problem Q, for which we have a composition, to P. For many of the examples we deal with, there are no obvious (or less than obvious) composition algorithms known, so this is indeed a non-trivial alternative to the strategy of showing a composition algorithm for ruling out polynomial kernels.

4.2 (Vertex) Disjoint Cycles

Consider the following two parameterized problems.

Disjoint Cycles	
Instance:	Undirected graph $G = (V, E)$ and a non-negative integer k.
Parameter:	k.
Question:	Does G contain at least k vertex-disjoint cycles?

Cycle Packing	
Instance:	Undirected graph $G = (V, E)$ and a non-negative integer k.
Parameter:	k.
Question:	Does G contain at least k edge-disjoint cycles?

The problem of Disjoint Cycles is strongly related to the Feedback Vertex Set (FVS) problem, wherein the question is whether there exist k vertices whose deletion makes the graph acyclic (usually studied with k as the parameter). Clearly, if a graph has more than k vertex disjoint cycles, then it cannot have a FVS of size k or less, as any FVS has to pick at least one vertex from every cycle. If there are at most k vertex disjoint cycles, the implications are less immediate, but an upper bound of $O(k \log k)$ on the size of the optimal FVS is known, due to a result by Erdös and Posa. For the FVS problem, a kernel of size $O(k^2)$ ([Tho09]) by Thomassé, who improved upon a kernel of size $O(k^3)$ ([Bod07]). The Disjoint Cycle Packing has a polynomial kernel (cf. [BTY08], Appendix B).

In contrast, we do not expect the Disjoint Cycles problem to admit a polynomial kernel, and we establish this by showing a polynomial parameter and time transformation from Disjoint Factors.

Recall that the Disjoint Factors problem was the following:

p-Disjoint Factors	
Instance:	A word $w \in L_k^*$.
Parameter:	$k \geqslant 1$.
Question:	Does w have the Disjoint Factors property?

Given an input (W, k) of Disjoint Factors, with $W = w_1 \cdots w_n$, a word in $\{0,1\}^*$, we build a graph $G = (V, E)$ as follows. First, we take n vertices $v_1, \ldots, v_n$, and edges $\{v_i, v_{i+1}\}$ for $1 \leqslant i < n$, i.e., these vertices form a path of length n. Let P denote this subgraph of G. Then, for each $i \in L_k$, we add a vertex x_i, and make x_i incident to each vertex v_j with $w_j = i$, i.e., to each vertex representing the letter i.

G has k disjoint cycles, if and only if (W, k) has the requested k disjoint factors. Suppose G has k disjoint cycles $c_1, \ldots, c_k$. As P is a path, each of these cycles must contain at least one vertex not on P, i.e., of the form x_j, and hence each of these cycles contains exactly one vertex x_j. For $1 \leqslant j \leqslant k$, the cycle c_j thus consists of x_j and a subpath of P. This subpath must start and end with a vertex incident to x_j. These both represent letters in W equal to j. Let F_j be the factor of W corresponding to the vertices on P in c_j. Now, $F_1, \ldots, F_k$ are disjoint factors, each of length at least two (as the cycles have length at least three), and F_j starts and ends with j, for all j, $1 \leqslant j \leqslant k$.

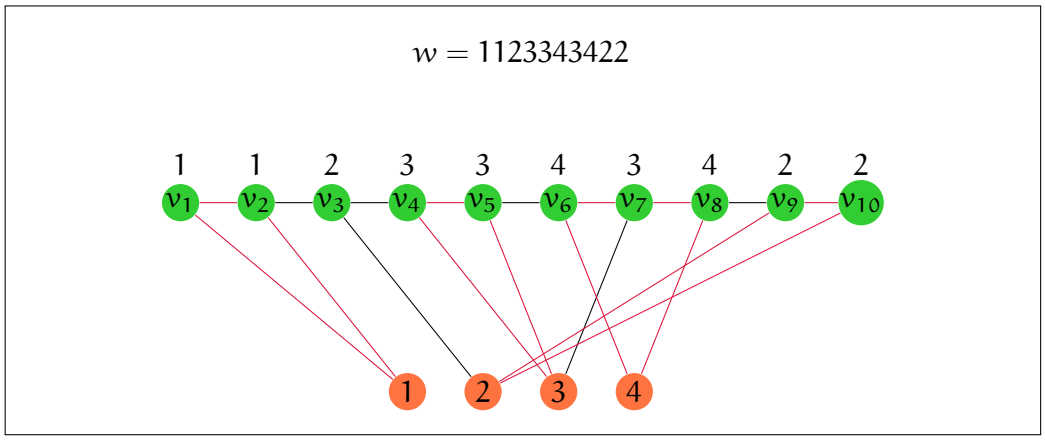


Figure 4.2: Disjoint Factors $\preccurlyeq_{ppt}$ Disjoint Cycles

Conversely, if we have disjoint factors $F_1, \ldots, F_k$ with the properties as in the Disjoint Factors problem, we build k vertex disjoint cycles as follows: for each j, $1 \leqslant j \leqslant k$, take the cycle consisting of x_j and the vertices corresponding to factor F_j. Thus we have shown:

Theorem 12 ([BDFH08]). Disjoint Factors *does not admit a polynomial kernel unless* $\mathsf{PH} = \Sigma_3^p$.

4.3 Red-Blue Dominating Set

Recall that in the previous chapter, we showed a composition for the colored variant of RBDS. Our real interest, however, is in showing the hardness of obtaining polynomial kernels for RBDS. We complete that argument here, by reducing RBDSfrom Col-RBDS.

Recall that in RBDS we are given a bipartite graph $G = (T \cup N, E)$ and an integer k and asked whether there exists a vertex set $N' \subseteq N$ of size at most k such that every vertex in T has at least one neighbor in N'. We also called the vertices "terminals" and "nonterminals" in order to avoid confusion with the colored version of the problem (Col-RBDS).

In the colored version that we showed was compositional, the vertices of N are colored with colors chosen from $\{1, \ldots, k\}$, that is, we are additionally given a function $col : N \to \{1, \ldots, k\}$, and N' is required to contain exactly one vertex of each color.

Theorem 13 ([DLS09]). (1) *The unparameterized version of* Col-RBDS *is* NP*-complete.*
(2) *There is a polynomial parameter transformation from* Col-RBDS *to* RBDS.
(3) Col-RBDS *is solvable in* $2^{|T|+k} \cdot |T \cup N|^{O(1)}$ *time.*

Proof. (1) It is easy to see that Col-RBDS is in NP. To prove its NP-hardness, we reduce the NP-complete problem RBDS to Col-RBDS: Given an instance $(G = (T \cup N, E), k)$ of RBDS, we construct an instance $(G' = (T \cup N', E'), k, col)$ of Col-RBDS where the vertex set N' consists of k copies $v^1, \ldots, v^k$ of every vertex $v \in V$,

one copy of each color. That is, $N' = \bigcup_{a\in\{1,\dots,k\}}\{v^a \mid v \in N\}$, and the color of every vertex $v^a \in N_a$ is $col(v^a) = a$. The edge set E' is given by

$$E' = \bigcup_{a\in\{1,\dots,k\}} \{\{u, v^a\} \mid u \in T \wedge a \in \{1,\dots,k\} \wedge \{u, v\} \in E\}.$$

The correctness of this construction is immediate.

(2) Given an instance $(G = (T \cup N, E), k, col)$ of Col-RBDS, we construct an instance $(G' = (T' \cup N, E'), k)$ of RBDS.

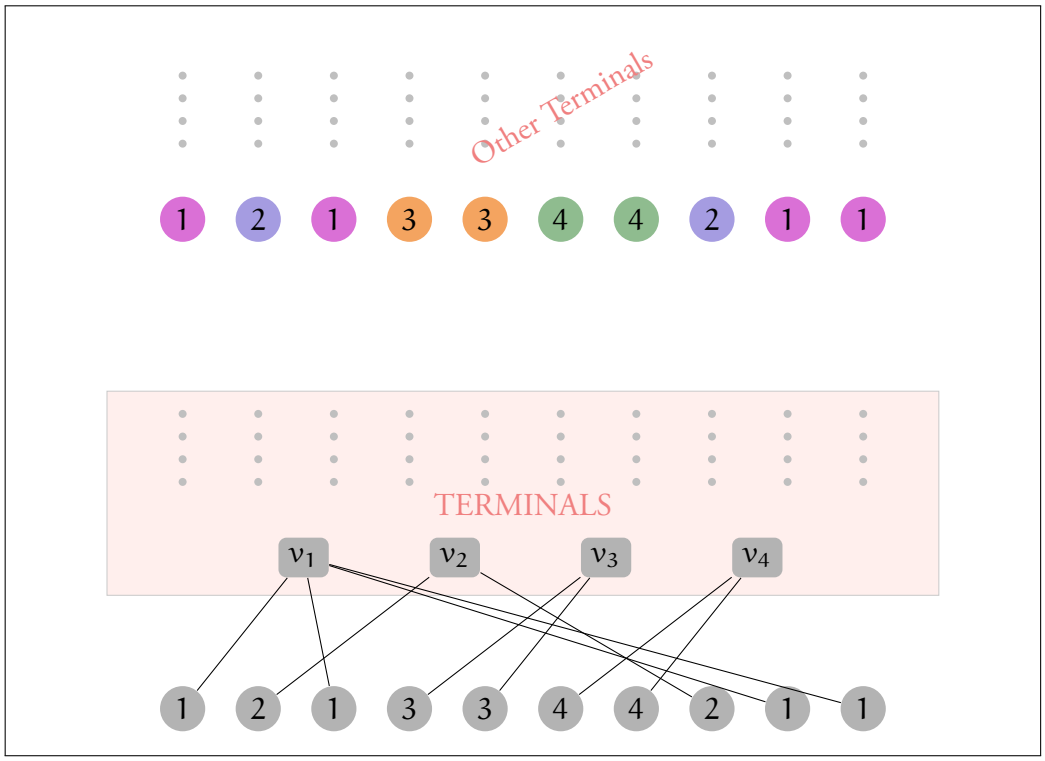


Figure 4.3: The polynomial parameter transformation from the colored version of RBDS to RBDS.

In G', the set T' consists of all vertices from T plus k additional vertices $z_1, \dots, z_k$. The edge set E' consists of all edges from E plus the edges

$$\{\{z_a, v\} \mid a \in \{1,\dots,k\} \wedge v \in N \wedge col(v) = a\}.$$

The proof of the correctness of this construction is immediate.

(3) To solve Col-RBDS in the claimed running time, we first use the reduction given in (2) from Col-RBDS to RBDS. The number $|T'|$ of terminals in the constructed instance of RBDS is $|T| + k$. Next, we transform the RBDS instance (G', k) into an instance $(\mathcal{F}, U, k)$ of Set Cover where the elements in U one-to-one correspond to the vertices in T' and the sets in $\mathcal{F}$ one-to-one correspond to the vertices in N. Since Set Cover can be solved in $O(2^{|U|} \cdot |U| \cdot |\mathcal{F}|)$ time [FKW04, Lemma 2], statement (3) follows.

□

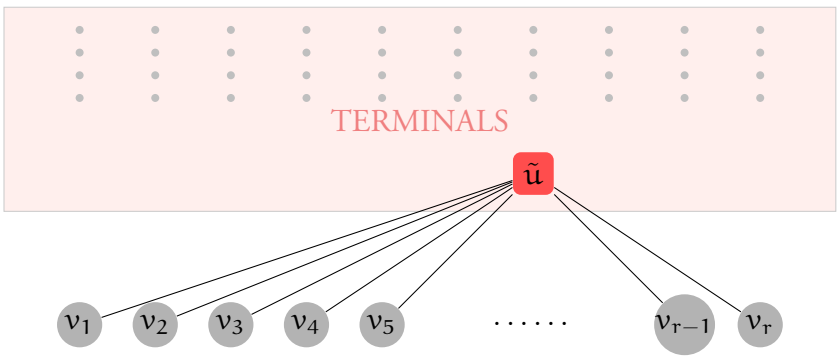
4.4 Reductions from RBDS

In this section, we use the fact that we have ruled out the possibility of polynomial kernels for RBDS to give hardness results for four other problems, all of which are known to be NP-complete (see [GJ79]). In this section, we rule out the possibility of their admitting polynomial kernels by polynomial parameter transformations from RBDS.

4.4.1 Steiner Tree

In Steiner Tree we are given a graph a graph $G = (T \cup N, E)$ and an integer k and asked for a vertex set $N' \subseteq N$ of size at most k such that $G[T \cup N']$ is connected. The problem is parameterized by $k + |T|$.

Let $(G = (T \cup N, E), k)$ be an instance of RBDS. To transform it into an instance $(G' = (T' \cup N, E'), k)$ of Steiner Tree, define $T' = T \cup \{\tilde{u}\}$ where $\tilde{u}$ is a new vertex and let $E' = E \cup \{\{\tilde{u}, v_i\} \mid v_i \in N\}$. It is easy to see that every solution for Steiner Tree on (G', k) one-to-one corresponds to a solution for RBDS on (G, k).



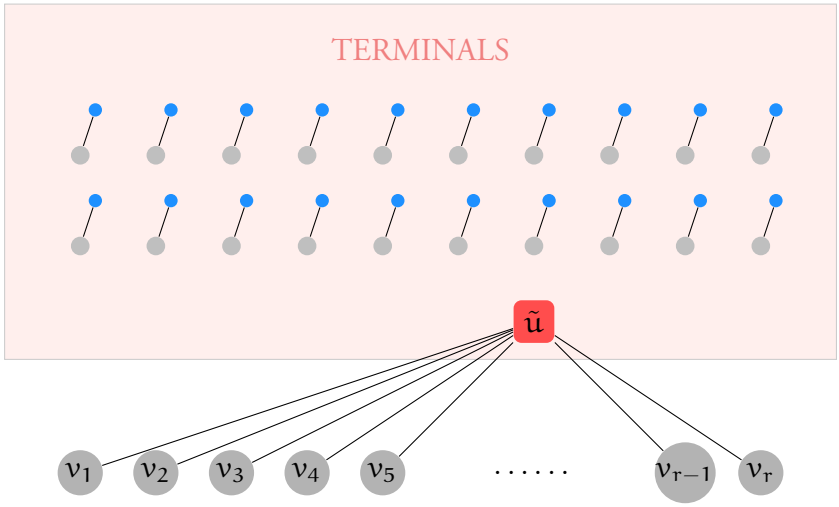
This is how we modify an instance of RBDS to convert it into an instance of Steiner Tree. Notice that solution to the Steiner Tree problem on this modified problem must dominate all the non-terminals (if it doesn't, then $\tilde{u}$ is not connected to the vertex that is not dominated).

Figure 4.4: The polynomial parameter transformation from RBDS to Steiner Tree.

4.4.2 Connected Vertex Cover

In ConVC we are given a graph $G = (V, E)$ and an integer k and asked for a vertex cover of size at most k that induces a connected subgraph in G. The parameter for the problem is the solution size, k.

To transform (G, k) into an instance $(G'' = (V'', E''), k'')$ of ConVC, first construct the graph $G' = (T' \cup N, E')$ as described above. The graph G'' is then obtained from G' by attaching a leaf to every vertex in T'. Now, G'' has a connected vertex cover of size $k'' = |T'| + k = |T| + 1 + k$ if and only if G' has a steiner tree containing k vertices from N if and only if all vertices from T can be dominated in G by k vertices from N.



There is an optimal solution for VertexCover that contains all the gray terminals, and the vertex $\tilde{u}$. To be a solution for ConVC, it must additionally connect all the vertex cover vertices, which are incidentally the set of all terminals for Steiner Tree, and hence it reduces precisely to the Steiner Tree problem (which in turn reduces to RBDS).

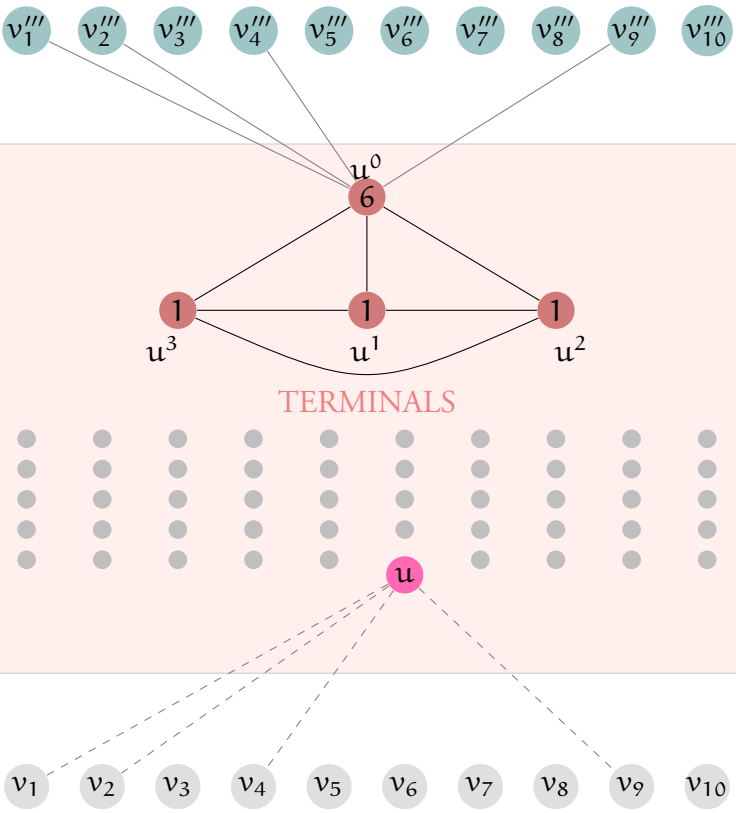
Figure 4.5: The polynomial parameter transformation from RBDS to ConVC.

4.4.3 Capacitated Vertex Cover

In CapVC we are asked to find a vertex cover on a graph where the vertices have capacities associated with them, and every vertex can cover at most as many edges as its capacity. The problem takes as input a graph $G = (V, E)$, a capacity function $cap : V \to \mathbb{N}^+$ and an integer k, and the task is to find a vertex cover C and a mapping from E to C in such a way that at most $cap(v)$ edges are mapped to every vertex $v \in C$. The parameter of this problem is k.

Next, we describe how to transform (G, k) into an instance $(G''' = (V''', E'''), cap, k''')$ of CapVC. First, for each vertex $u_i \in T$, add a clique to G''' that contains four vertices $u_i^0, u_i^1, u_i^2, u_i^3$. Second, for each vertex $v_i \in N$, add a vertex v_i''' to G'''. Finally, for each edge $\{u_i, v_j\} \in E$ with $u_i \in T$ and $v_j \in N$, add the edge $\{u_i^0, v_j'''\}$

to G'''. The capacities of the vertices are defined as follows: For each vertex $u_i \in T$, the vertices $u_i^1, u_i^2, u_i^3 \in V'''$ have capacity 1 and the vertex $u_i^0 \in V'''$ has capacity $\deg_{G'''}(u_i^0) - 1$. Each vertex v_i''' has capacity $\deg_{G'''}(v_i''')$. Clearly, in order to cover the edges of the size-4 cliques inserted for the vertices of T, every capacitated vertex cover for G''' must contain all vertices $u_i^0, u_i^1, u_i^2, u_i^3$. Moreover, since the capacity of each vertex u_i^0 is too small to cover all edges incident to u_i^0, at least one neighbor v_j''' of u_i^0 must be selected into every capacitated vertex cover for G'''. Therefore, it is not hard to see that G''' has a capacitated vertex cover of size $k''' = 4 \cdot |T| + k$ if and only if all vertices from T can be dominated in G by k vertices from N.



An illustration of the reduction from RBDS to CapVC.

Figure 4.6: The polynomial parameter transformation from RBDS to CapVC.

4.4.4 Bounded Rank Set Cover

Finally, an instance of Bounded Rank Set Cover consists of a set family $\mathcal{F}$ over a universe U where every set $S \in \mathcal{F}$ has size at most d, and a positive integer k. The task is to find a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ of size at most k such that $\cup_{S \in \mathcal{F}'} S = U$. The problem is parameterized by $(k + d)$.

To transform (G, k) into an instance $(\mathcal{F}, U, k)$ of Bounded Rank Set Cover, add one element e_i to U for every vertex $u_i \in T$. For every vertex $v_j \in N$, add one set $\{e_i \mid \{u_i, v_j\} \in E\}$ to $\mathcal{F}$. The correctness of the construction is obvious, and since $|U| = |T|$, every set in $\mathcal{F}$ contains at most $d = |T|$ elements.

4.4.5 Summary

Thus we have the following theorem:

Theorem 14 ([DLS09]). *The problem* Steiner Tree *parameterized by* $(|T|, k)$*, and the problems* Connected Vertex Cover *and* Capacitated Vertex Cover*, both parameterized by* k*, and the problem* Bounded Rank Set Cover *parameterized by* (k, d) *do not admit polynomial kernels unless* $PH = \Sigma_p^3$.

4.5 Dominating Set on Degenerate Graphs

To show that the Dominating Set problem in d-degenerate graphs does not have a kernel of size $\mathrm{poly}(k, d)$ (assuming $PH \neq \Sigma_3^P$), we appeal to the fact that the Small Universe Hitting Set is compositional (we refer the reader to [DLS09] for details). In this problem we are given a set family $\mathcal{F}$ over a universe U with $|U| \leqslant d$ together with a positive integer k. The question is whether there exists a subset S in U of size at most k such that every set in $\mathcal{F}$ has a non-empty intersection with S. It can be shown that the Small Universe Hitting Set problem parameterized by the solution size k and the size $d = |U|$ of the universe does not have a kernel of size polynomial in (k, d) unless $PH = \Sigma_p^3$:

Theorem 15 ([DLS09]). Small Universe Hitting Set *parameterized by solution size* k *and universe size* $|U| = d$ *does not have a polynomial kernel unless* $PH = \Sigma_p^3$.

The Dominating Set *problem parameterized by the solution size* k *and the size* c *of a minimum vertex cover of the input graph does not have a polynomial kernel.*

Theorem 15 has some interesting consequences. For instance, it implies that the Hitting Set problem parameterized by solution size k and the maximum size d of any set in $\mathcal{F}$ does not have a kernel of size $\mathrm{poly}(k, d)$ unless $\mathsf{PH} = \Sigma_p^3$. The second part of Theorem 15 implies that the Dominating Set problem in graphs excluding a fixed graph H as a minor parameterized by $(k, |H|)$ does not have a kernel of size $\mathrm{poly}(k, |H|)$ unless $\mathsf{PH} = \Sigma_p^3$. This follows from the well-known fact that every graph with a vertex cover of size c excludes the complete graph K_{c+2} as a minor. Similarly, since every graph with a vertex cover of size c is c-degenerate it follows that the Dominating Set problem in d-degenerate graphs does not have a kernel of size $\mathrm{poly}(k, d)$ unless $\mathsf{PH} = \Sigma_p^3$.

Theorem 16 ([DLS09])**.** *Unless* $\mathsf{PH} = \Sigma_p^3$ *the problems* Hitting Set *parameterized by solution size* k *and the maximum size* d *of any set in* $\mathcal{F}$*,* Dominating Set in H-Minor Free Graphs *parameterized by* $(k, |H|)$*, and* Dominating Set *parameterized by solution size* k *and degeneracy* d *of the input graph do not have a polynomial kernel.*

5. Kernels for Problems Without a Kernel

If I have a thousand ideas and only one turns out to be good,
I am satisfied.

Alfred Bernhard Nobel

How do we cope with the hardness of polynomial kernelization? Does a composition algorithm, or worse, a Linear OR , indicate the end of the road for polynomially small kernels? So long as the PH holds up, we understand the answer to be in the affirmative. However, while we should not expect a polynomial kernel, nothing prevents us from creating many of them. We are clearly better off with as few of the many as possible. It's interesting to wonder if we may expect polynomial-in-k number of poly-sized kernels. (At first sight, it would seem that the overall size would then be a polynomial in k, and hence unexpected. But we have only learnt to not expect poly-sized *kernel*, while this collection of kernels is not, strictly speaking, a kernel.)

The rest of this chapter is dedicated to describing the strategy of devising a large number of small kernels for problems that have none to begin with.

5.1 On the use of "Kernel" in the plural

Although a problem (Q, κ) may not necessarily admit a polynomial kernel, it may evidently admit many of them, with the property that the instance is in the language if and only if at least one of the kernels corresponds to an instance that is in the language. It is worth checking if the problem has a Linear OR, to confirm that it is incompressible in a stronger sense than not admitting a polynomial kernel, before attempting to get many poly-kernels. In the many poly-kernels approach, we would like as few kernels as possible (each one adds to the runtime of any algorithm that would use the kernels to solve the problem in question). In the example discussed, we obtain n kernels of size $O(k^3)$ each. Notice that the instance size has actually increased by a multiplicative factor of $O(k^3)$, so this is not really a compression routine. (Given our obsession with polynomials in k, we should anticipate a compromise somewhere!)

This is a bit suspicious - aren't we better off using kernels, which, even when much worse than polynomial in size, are at least guaranteed to not *increase* the size of the instance? Let us see how 2^k sized kernels compare to n linear kernels. When $n = 2^k$, for example, the kernelization does not have to do anything, and may leave the instance untouched. On the other hand, with the many-kernels approach, we are left with n kernels of size k each. The nk-sized collection of instances would always be larger than a single instance of size n, but since the former object is not a single instance, there may be situations where it is a time-saver. For instance, a FPT algorithm would take time

$$f(k) \cdot (2^k)^{O(1)}$$

with the kernel, while it'll take time

$$2^k \cdot f(k) \cdot k^{O(1)}$$

while working with the many kernels. The latter beats the former for any non-trivial constant in the exponent.

Multiple polynomial kernels are quite exciting in practice - in particular, it opens up the possibility of implementing algorithms that use parallel processing, since the kernels are independent of each other, and this potentially makes them very useful. In fact, it is perhaps worth studying multiple kernels in general, even when the problem admits a poly-sized kernel. Since here we are not a-priori bound to have a large number of poly-kernels - we may hope to have, for instance, k^2 linear kernels for a problem where the best known kernels are cubic in size.

5.2 Case Study: k-outbranching

The Maximum Leaf Spanning Tree problem on connected undirected graphs is to find a spanning tree with the maximum number of leaves in a given input graph G. An extension of Maximum Leaf Spanning Tree to directed graphs is defined as follows. We say that a subdigraph T of a digraph D is an *out-tree* if T is an oriented tree with only one vertex r of in-degree zero (called the *root*). The vertices of T of out-degree zero are called *leaves*. If T is a spanning out-tree, i.e., $V(T) = V(D)$, then T is called an *out-branching* of D. The Directed Maximum Leaf Out-Branching

problem is to find an out-branching in a given digraph with the maximum number of leaves. The parameterized version of the Directed Maximum Leaf Out-Branching problem is k-Leaf Out-Branching, where for a given digraph D and integer k, it is asked to decide whether D has an out-branching with at least k leaves. If we replace 'out-branching' with 'out-tree' in the definition of k-Leaf Out-Branching, we get a problem called k-Leaf Out-Tree.

Unlike the undirected counterpart, the study of k-Leaf Out-Branching has been investigated only recently. Alon et al. [AFG+07b, AFG+07a] proved that the problem is fixed parameter tractable (FPT) by providing an algorithm deciding in time $O(f(k)n)$ whether a strongly connected digraph has an out-branching with at least k leaves. Bonsma and Dorn [BD08] extended this result to connected digraphs, and improved the running time of the algorithm. Recently, Kneis et al. [KLR08] provided a parameterized algorithm solving the problem in time $4^k n^{O(1)}$. This result was further improved by Daligaut et al. [DGKY08]. In a related work, Drescher and Vetta [DV08] described an $\sqrt{OPT}$-approximation algorithm for the Directed Maximum Leaf Out-Branching problem.

Further, it is established ([FFL+09]) that Rooted k-Leaf Out-Branching, where for a given vertex r one asks for a k-leaf out-branching rooted at r, admits an $O(k^3)$ kernel. A similar result also holds for Rooted k-Leaf Out-Tree, where we are looking for a rooted (not necessary spanning) tree with k leaves.

Given these positive results, it is but natural to suspect that k-Leaf Out-Branching admits a polynomial kernel. However, this is not the case – k-Leaf Out-Branching and k-Leaf Out-Tree do not admit polynomial kernels unless $PH = \Sigma_p^3$. We describe the proof as described in [FFL+09], at whose heart is the framework built so far, albeit with some twists. We summarize the situation in Table 5.2.

Table 5.1: Kernel(s) for Out–Tree and Out–Branching problems

	k–Out–Tree	k–Out-Branching
Rooted	$O(k^3)$ kernel	$O(k^3)$ kernel
Unrooted	No poly(k) kernel n kernels of size $O(k^3)$	No poly(k) kernel n kernels of size $O(k^3)$

Evidently, we may already observe that the polynomial kernels for the rooted versions of our problems provide the many-poly-kernels which were promised for the poly-kernel-intractable k-Leaf Out-Branching and k-Leaf Out-Tree. Indeed, let D be a digraph on n vertices. By running the kernelization for the rooted version of the problem for every vertex of D as a root, we obtain n graphs where each of them has $O(k^3)$ vertices, such that at least one of them has a k-leaf out-branching if and only if D does.

5.2.1 A Composition for k-Leaf Out-Branching

We assume the following theorem, from [FFL+09]:

Theorem 17. Rooted k-Leaf Out-Branching *and* Rooted k-Leaf Out-Tree *admits a kernel of size* $O(k^3)$.

It turns out that the kernel is useful in designing the compositional algorithm, which is an unusual feature of this composition.

Theorem 18. k-Leaf Out-Tree *has no polynomial kernel unless* $\text{PH}=\Sigma_p^3$.

Proof. The problem is NP-complete [AFG+07b]. We prove that it is compositional, which will imply the statement of the theorem. A simple composition algorithm for this problem is as follows. On input $(D_1,k),(D_2,k),\ldots,(D_t,k)$ output the instance (D,k) where D is the disjoint union of $D_1,\ldots,D_t$. Since an out-tree must be completely contained in a connected component of the underlying undirected graph of D, (D,k) is a yes-instance to k-Leaf Out-Tree if and only if any out of (D_i,k), $1\leqslant i\leqslant t$, is. This concludes the proof. □

We now show that k-Leaf Out-Branching is not likely to admit a polynomial kernel. The argument here, however, is less direct. We assume, for the sake of contradiction, that k-Leaf Out-Branching has a polynomial kernel. We then contradict Theorem 18 by demonstrating a polynomial kernel for k-Leaf Out-Tree. We do this in a number of steps:

1. First, we "decompose" the instance of the Out-Tree problem into n instances of the rooted variant, such that the original instance is a YES–instance if and only if one at least one of the decomposed instances is a YES–instance.

2. We then kernelize each of the decomposed instances using Theorem 17.

3. After this we would like to compose these instances into one and hope to prove that what we get is a kernel for k-LEAF OUT-TREE.

   The composition is made easier by converting these graphs into ones with some special structure, and going over to the *Out–Branching* problem in this restricted graph class using a NP–completeness reduction.

4. We "poly" kernelize the composed instance (since we begin with the assumption that k–LEAF OUT–BRANCHING admits a polynomial kernel).

5. Finally, since what we get is an instance of Out-Branching, we again apply a NP–completeness reduction in order to get to a polynomial kernel for k-LEAF OUT-TREE.

The overall plan is illustrated in Figure 5.1.

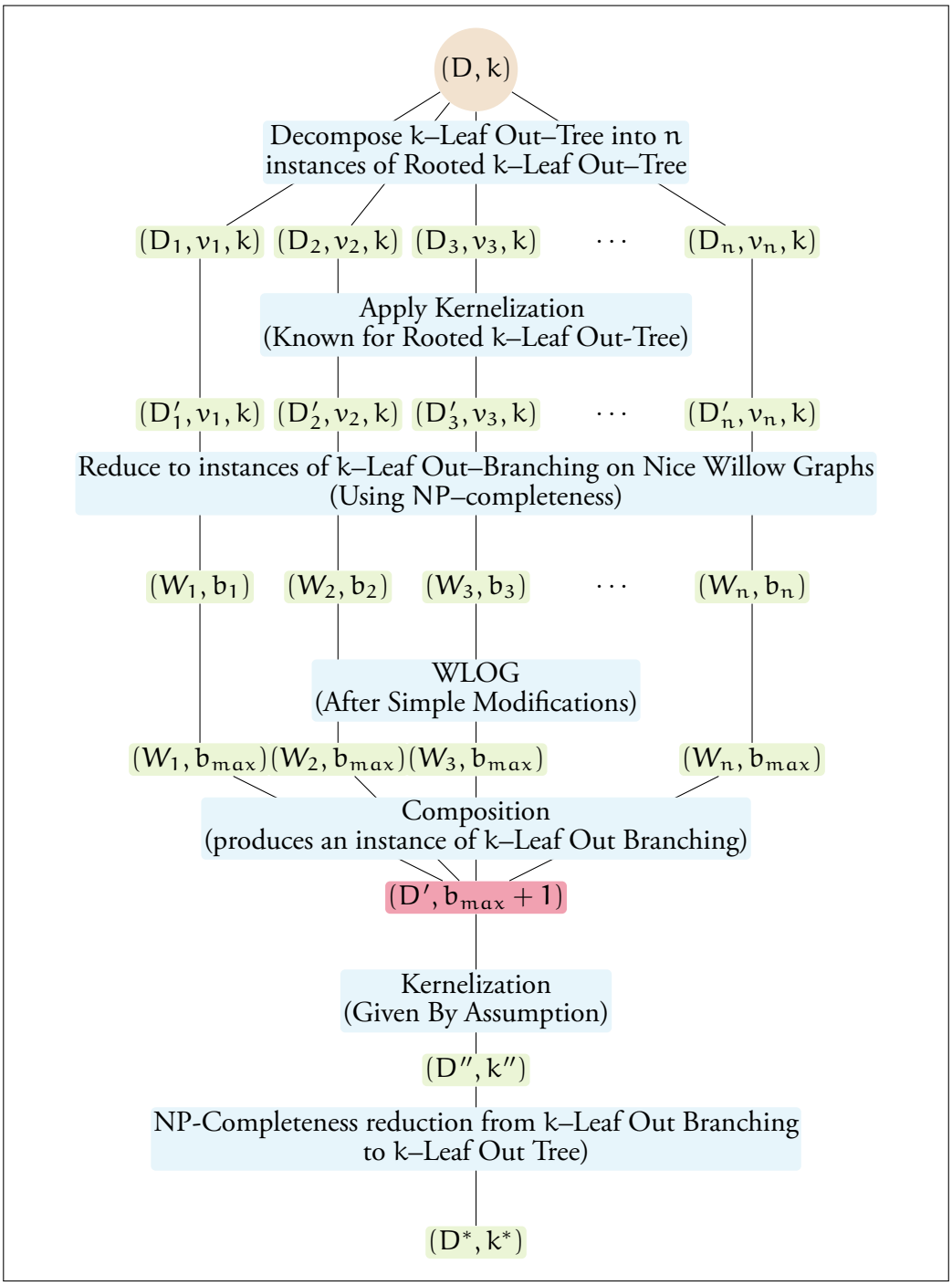


Figure 5.1: The No Polynomial Kernel Argument for k-Leaf Out-Branching.

We begin by introducing the special graph class that simplifies our composition step.

A *willow* graph [DV08] $D = (V, A_1 \cup A_2)$ is a directed graph such that $D' = (V, A_1)$ is a directed path $P = p_1 p_2 \dots p_n$ on all vertices of D and $D'' = (V, A_2)$ is a directed acyclic graph with one vertex r of in-degree 0, such that every arc of A_2 is a backwards arc of P. p_1 is called the *bottom* vertex of the willow, p_n is called the *top* of the willow and P is called the *stem*.

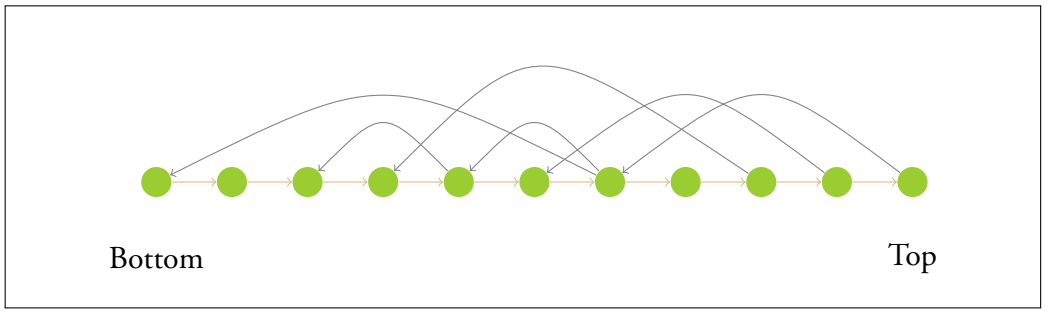


Figure 5.2: Example of a Willow Graph

A *nice willow* graph $D = (V, A_1 \cup A_2)$ is a willow graph where $p_n p_{n-1}$ and $p_n p_{n-2}$ are arcs of D, neither p_{n-1} nor p_{n-2} are incident to any other arcs of A_2 and $D'' = (V, A_2)$ has a p_n-out-branching.

Observation 1. *Let $D = (V, A_1 \cup A_2)$ be a nice willow graph. Every out-branching of D with the maximum number of leaves is rooted at the top vertex p_n.*

Proof. Let $P = p_1 p_2 \dots p_n$ be the stem of D and suppose for contradiction that there is an out-branching T with the maximum number of leaves rooted at p_i, $i < n$. Since D is a nice willow $D' = (V, A_2)$ has a p_n-out-branching T'. Since every arc of A_2 is a back arc of P, $T'[\{v_j : j \geqslant i\}]$ is an p_n-out-branching of $D[\{v_j : j \geqslant i\}]$. Then $T'' = (V, \{v_x v_y \in A(T') : y \geqslant i\} \cup \{v_x v_y \in A(T) : y < i\})$ is an out-branching of D. If $i = n - 1$ then p_n is not a leaf of T since the only arcs going out of the set $\{p_n, p_{n-1}\}$ start in p_n. Thus, in this case, all leaves of T are leaves of T'' and p_{n-1} is a leaf of T'' and not a leaf of T, contradicting the fact that T has the maximum number of leaves.

□

Lemma 3. k-Leaf Out-Tree *in nice willow graphs is NP-complete under Karp reductions.*

A proof is provided in Appendix 6.2.

Theorem 19. k-Leaf Out-Branching *has no polynomial kernel unless* PH=Σ_p^3.

Proof. We prove that if k-Leaf Out-Branching has a polynomial kernel then so does k-Leaf Out-Tree. Let (D, k) be an instance to k-Leaf Out-Tree. For every vertex $v \in V$ we make an instance (D, v, k) to Rooted k-Leaf Out-Tree. Clearly, (D, k) is a yes-instance for k-Leaf Out-Tree if and only if (D, v, k) is a yes-instance to Rooted k-Leaf Out-Tree for some $v \in V$. By Theorem 17 Rooted k-Leaf Out-Tree has a $O(k^3)$ kernel, so we can apply the kernelization algorithm for Rooted k-Leaf Out-Tree separately to each of the n instances of Rooted k-Leaf Out-Tree to get n instances (D_1, v_1, k), (D_2, v_2, k), ..., (D_n, v_n, k) with $|V(D_i)| = O(k^3)$ for each $i \leqslant n$. By Lemma 3, k-Leaf Out-Branching in nice willow graphs is NP-complete under Karp reductions, so we can reduce each instance (D_i, v_i, k) of Rooted k-Leaf Out-Tree to an instance (W_i, b_i) of k-Leaf Out-Branching in nice willow graphs in polynomial time in $|D_i|$, and hence in polynomial time in k. Thus, in each such instance, $b_i \leqslant (k+1)^c$ for some fixed constant c independent of both n and k. Let $b_{max} = \max_{i \leqslant n} b_i$. Without loss of generality, $b_i = b_{max}$ for every i. This assumption is safe because if it does not hold we can modify the instance (W_i, b_i) by replacing b_i with b_{max}, subdividing the last arc of the stem $b_{max} - b_i$ times and adding an edge from r_i to each subdivision vertex.

From the instances $(W_1, b_{max}), \ldots, (W_n, b_{max})$ we build an instance $(D', b_{max}+1)$ of k-Leaf Out-Branching. Let r_i and s_i be the top and bottom vertices of W_i, respectively. We build D' simply by taking the disjoint union of the willows graphs $W_1, W_2, \ldots, W_n$ and adding in an arc $r_i s_{i+1}$ for $i < n$ and the arc $r_n s_1$. Let C be the directed cycle in D obtained by taking the stem of D' and adding the arc $r_n s_1$.

If for any $i \leqslant n$, W_i has an out-branching with at least b_{max} leaves, then W_i has an out-branching rooted at r_i with at least b_{max} leaves. We can extend this to an out-branching of D' with at least $b_{max} + 1$ leaves by following C from r_i. In the other direction suppose D' has an out-branching T with at least $b_{max} + 1$ leaves. Let

i be the integer such that the root r of T is in $V(W_i)$. For any vertex v in $V(D')$ outside of $V(W_i)$, the only path from r to v in D' is the directed path from r to v in C. Hence, T has at most 1 leaf outside of $V(W_i)$. Thus, $T[V(W_1)]$ contains an out-tree with at least b_{max} leaves.

By assumption, k-Leaf Out-Branching has a polynomial kernel. Hence, we can apply a kernelization algorithm to get an instance (D'', k'') of k-Leaf Out-Branching with $|V(D'')| \leqslant (b_{max}+1)^{c_2}$ for a constant c_2 independent of n and b_{max} such that (D'', k'') is a yes-instance if and only if (D', b_{max}) is.

Finally, since k-Leaf Out-Tree is NP-complete, we can reduce (D'', k'') to an instance (D^*, k^*) of k-Leaf Out-Tree in polynomial time. Hence, $k^* \leqslant |V(D^*)| \leqslant (|V(D'')|+1)^{c_3} \leqslant (k+1)^{c_4}$ for some fixed constants c_3 and c_4. Hence, we conclude that if k-Leaf Out-Branching has a polynomial kernel then so does k-Leaf Out-Tree. Thus, Theorem 18 implies that k-Leaf Out-Branching has no polynomial kernel unless PH=Σ_p^3.

□

5.3 General Remarks

It appears that we may obtain many poly kernels for a problem (Q, κ) if we know of another language Q' for which we already have a polynomial kernel, and Q can be re-stated as the boolean OR of many instances of Q'. Although it is often not hard to find examples of Q and Q' which satisfy this criteria, we usually don't have the contrast of Q being hard to poly-kernelize *and* Q' having a poly-kernel.

For instance, consider k-path. We may fix an arbitrary vertex v and ask for a pointed $k-1$ path that begins at v. If we had a poly-sized kernel for pointed k-path, then we would have n polynomial kernels for k-path, as we simply have to iterate over all choices of v. Sadly, pointed k-path is just as hard as k-path in the polynomial kernel context, so this observation is not very useful.

We emphasize again that the idea of using multiple kernels may have applications that transcend the no-poly-kernel scene. It is an extremely recent technique, we expect to see it in action in more contexts than one.

6. The Missing Pieces

As the circle of light grows, so does the circumference of darkness around it.

– Albert Einstein

The scene of kernel lower bounds so far has been concentrated on NP-complete problems specified with one parameter. We hope to use the parameter for a refined analysis of the problem complexity. In the interest of pinning down the "hardness quotient" of a problem even further, a natural idea is to introduce one *more* parameter. This, presumably, gives us new insight into the complexity of the problem. An additional parameter may have the (desirable) effect of pushing the problem to a lower complexity class. Further, in the context of problem kernels, it is easy to ask hard questions. May we rule out kernels whose size is polynomial in both parameters? Or polynomial in at least one of the parameters?

For those who think of the notion of parameterized complexity as multivariate complexity', the use of multiple parameters is a natural generalization of the single parameter picture that we are accustomed to. If a problem is equipped naturally with a number of aspects that may be treated as parameters, then using these parameters to evaluate the problem's parameterized complexity is an exercise that suggests itself. As for problem kernels in the more general picture,5 the questions that can be asked evidently grows exponentially with the number of parameters. We restrict our attention to problems with two parameters in this chapter.

The machinery for establishing kernel lower bounds of various kinds is tantalizingly close to being comprehensive. We state the few questions that remain elusive. In the meantime, note that we already have examined a number of problems in which the theorems that we know turned out to be useful. We expect that the framework will find many more applications - indeed, it remains for them to be fully exploited. Although examples abound presently, it is not an overestimate to claim that this is only the beginning.

6.1 The Two-Parameter Context

Consider the problem of finding a dominating set with at most k vertices. On general graphs, the problem is well-known to be $W[2]$-hard when parameterized by the solution size. To "cope" with the hardness of the problem, we would like to restrict our attention to some subclass of graphs where we may hope to exploit a property of the subclass towards an efficient algorithm. Consider, for instance, graphs that have "small" vertex covers. These may be thought of as graphs that have vertex cover of size t (or smaller) for an arbitrary but fixed t. Any such t cuts out a slice from the set of all graphs - we now attempt to describe this "restriction" as a parameter that accompanies these graphs.

Consider the problem of finding a k-sized dominating set on a graph G which has a vertex cover of size t. We may regard both k and t as parameters to the problem. Noting that $k \leqslant t$ (any vertex cover is also a dominating set), our notion of a "FPT" algorithm is now something that spends

$$f(t) \cdot p(n)$$

time. Now equipped with the promise of a vertex cover of size t, do we get anywhere? The good news is that this variant is in FPT - however, it is unlikely to admit a kernel of size $(k+t)^{O(1)}$ ([DLS09]). The known FPT algorithm runs in time $2^t|V|^{O(1)}$.

Due to polynomial time and parameter preserving reductions, the hardness result implies that Dominating Set in H-Minor Free Graphs parameterized by solution size k and $|H|$ does not have a $O((k+|H|)^{O(1)})$ kernel, and that Dominating Set parameterized by k and degeneracy d of the input graph has no $O((k+d)^{O(1)})$ kernel (subject to the PH $\neq \Sigma_3^p$ assumption). On the positive side, the best kernel known so far for the problem of Dominating Set in Graphs With Small Vertex Cover is $O(2^t)$. We do not expect a $t^{O(1)}$ kernel here because of the hardness described. The more compelling situation arises with Dominating Set on d-degenerate graphs, where a $k^{O(d^2)}$ kernel is known ([PRS09]). It is not clear whether the lower or the upper bound will improve here; and while the latter involves the discovery of more powerful reduction rules, any improvement of the lower bound is likely to demand a new hardness-establishing technique.

6.1.1 Uniform Kernels

For problems with two parameters, say k and l, the size of the kernel may be a function of any one of the following kinds:

- $f(k, l)$: An arbitrary function of k and l alone.
- $p(k)^{f(l)}$: A polynomial in k with an arbitrary function of l sitting on its exponent.
- $p(k) \cdot f(l)$: A polynomial in k with the arbitrary function of l driven out of the exponent[1].
- $p(k)p(l)$: Effectively a pure polynomial in both parameters

Note that, as usual, the "arbitrary" function refers to a computable one. It seems a bit of a trend among FPT problems with two parameters - the "purely polynomial" kernels are shown to be unlikely, and is trivial to note that the $f(k, l)$-sized kernels exist (for problems in FPT with one parameter, we had an argument establishing that this implies a kernel - this is easily generalized to accommodate for more parameters). Usually we get as far as a $p(k)^{f(l)}$, and the crucial question is unanswered - may we have a kernel whose size is a polynomial in one of the parameters, with an arbitrary function of the other occurring as only a multiplicative factor? Unfortunately, we do not know of a "lower bound scheme" that will rule out such a possibility, and this is an important question. Without this, the only direction in which we may attack the two-parameter problems for which we are stuck at $p(k)^{f(l)}$-sized kernels is to attempt improvements in the size - but if such a goal is indeed unattainable, then this approach is doomed!

Another example that emphasizes this dilemma is the so-called "Small Universe Hitting Set" problem, which is the familiar Hitting Set problem except that the universe size is also a parameter:

[1] This should remind us of the difference between XP and FPT!

Small Universe Hitting Set	
Input:	A set family $\mathcal{F}$ over a universe U with $\|U\| = d$, and a positive integer k.
Question:	Is there a subset $H \subseteq U$ of size at most k such that for every set $S \in \mathcal{F}, H \cap S \neq \phi$?
Parameter(s):	k, d

The best algorithm known has a runtime of $O(2^d \cdot (|\mathcal{F}| + |U|)^{O(1)})$, and the best known kernel is of size $O(2^d)$ - no $O((k+d)^{O(1)})$ kernel is expected ([DLS09]).

The $p(k) \cdot f(l)$-sized kernels have even acquired a name by now - they are called uniformly polynomial kernels, and we emphasize that a lower bound framework for ruling out such kernels is would be intriguing - and useful.

6.2 Other Open Problems

The subclasses described by problems with polynomial, subexponential, and pseudo-polynomial sized kernels are strictly separated under reasonable complexity theoretic assumptions. The most pertinent question in this picture is to find ways of subdividing the "Polynomial Kernels" class further. Case(s) in point: Feedback Vertex Set, Cluster Editing, Edge Disjoint Cycles, and various other problems are known to admit quadratic kernels, with the linear kernel question being wide open. At this point, we only have tools that would help us resolve the "is there a linear kernel" question, but nothing that will help us prove the possibility that the existence of a linear kernel "is unlikely".

On the one hand, there are no known ways of saying that a problem is unlikely to have a kernel of size k^c for some specific c, because of which we are apparently stuck with "polynomial kernels" as a large class. On the other, recall our examination of the notions of strong kernelization - and note that they may give us another way of digging deeper into this set simply by not thinking of a problem from the point of view of its current-best-kernel size, but the *size of the new parameter in the kernel of the problem* instead.

Unless P = NP, the strict variant of the inequality

$$\kappa(\mathbb{K}(x)) \leqslant \kappa(x)$$

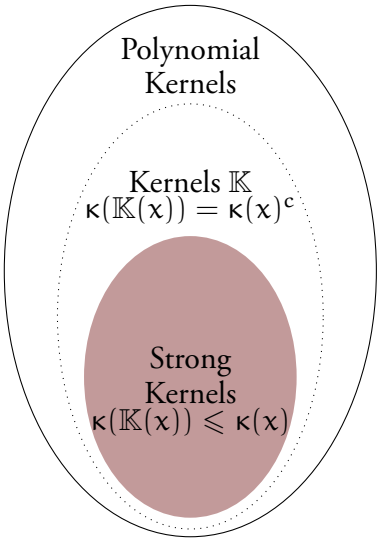
is never attained inside the innermost class of problems. In other words, a parameter-decreasing kernelization of polynomial size may be considered impossible. We know that the separation of classes that satisfy

$$\kappa(\mathbb{K}(x)) = \kappa(x)^c$$

and

$$\kappa(\mathbb{K}(x)) = \kappa(x)^{c+1}$$

is strict (unconditionally!). However, what is still missing is a technique that will allow us to place a problem in a precise class.



The other obvious set of questions to ask is whether the lower bound theorems can be improved to unconditional statements. This will have the effect of decreasing, by at least a constant factor, the number of times the phrase "under reasonable complexity theoretic assumptions" is used in the literature. A less ambitious goal would be to attempt proving the same results under some "well-believed" conjectures of parameterized complexity, for instance, FPT $\neq$ XP, or, FPT $\neq W[t]$ for some $t \in \mathbb{N}^+$.

Much of our focus has been on composition algorithms that perform some kind of a boolean "or" on their inputs. Their counterparts are algorithms that similarly perform an "and" of the inputs, however, the existence of such algorithms for NP-complete problems is not known to cause any specific disaster yet, and is therefore not useful for proving lower bounds. It is an interesting open problem to pursue the implications of the existence of "and" compositions for classical and parameterized languages.

The notion of kernelization is popular in practice - in many cases, it can be thought of as a precise way of stating all the heuristic-based preprocessing steps that have been popular and effective for a long time. In theory, the notion is important for more than one reason - there is an increasingly popular feeling that kernelization is *the*

way of understanding fixed parameter tractability. The theorem that establishes the equivalence of these notions is more than a syntactic equality - it encodes an entire philosophy, and immediately puts on offer a possible "right way" of viewing FPT. Given that FPT is the most fundamental class of parameterized problems, the fact that we finally have a few lower bounds for placing problems in different places *within* this class is a great source of excitement, and a non-trivial hope for a deeper understanding of problem complexity.

Appendix A

A Problem Compendium

k–path

Input: A graph G and a non-negative integer k.

Question: Does G have a path of length k?

Parameter: k.

Algorithm: Using color coding, an algorithm that takes $2^{O(k)} \cdot O(V^{\omega} \log V)$ worst-case time, where $\omega < 2.376$ is the exponent of matrix multiplication [AYZ95].

Kernel: **No $k^{O(1)}$ kernel** [CFM07].

Pointed k–path

Input: A graph G, a vertex $v \in V(G)$, and a non-negative integer k.

Question: Does G have a path of length k starting at v?

Parameter: k.

Algorithm: Using color coding, an algorithm that takes $2^{O(k)} \cdot O(V^{\omega} \log V)$ worst-case time, where $\omega < 2.376$ is the exponent of matrix multiplication [AYZ95].

Kernel: **No $k^{O(1)}$ kernel** [CFM07].

k k–paths

Input: A graph G, a vertex $v \in V(G)$, and a non-negative integer k.

Question: Does G have k vertex–disjoint paths of length k?

Parameter: k.

Algorithm: Using color coding, an algorithm that takes $2^{O(k^2)} \cdot O(V^{\omega} \log V)$ worst-case time, where $\omega < 2.376$ is the exponent of matrix multiplication [AYZ95].

Kernel: **No $k^{O(1)}$ kernel**.

Disjoint Factors

Input: A word $w \in L_k^*$.

Question: Does w have the Disjoint Factors property?

Parameter: k.

Algorithm: $2^k \cdot p(n)$ Algorithm, using dynamic programming [BDFH08].

Kernel: **No** $k^{O(1)}$ **kernel** [BDFH08].

p**–SAT**

Input: A propositional formula α in conjunctive normal form.

Question: Is α satisfiable?

Parameter: $k :=$ Number of variables of α.

Algorithm: $2^k \cdot p(n)$ algorithm, by enumeration

Kernel: **No** $k^{O(1)}$ **kernel** [CFM07].

pw**–SAT**

Input: A propositional formula α in conjunctive normal form, with clause length bounded by some constant c, and a non-negative integer k.

Question: Is there a satisfying assignment for α with weight at most k?

Parameter: k

Algorithm: $c^k \cdot p(n)$ algorithm, using a search tree.

Kernel: **No** $k^{O(1)}$ **kernel**.

Vertex Disjoint Cycles

Input: A graph G and a non-negative integer k.

Question: Does G have k vertex–disjoint cycles?

Parameter: k

Algorithm: A $2^{O(k \log k)}$ algorithm is known [BDFH08].

Kernel: **No** $k^{O(1)}$ **kernel** [BDFH08].

Bipartite Regular Perfect Code

Input: A bipartite graph $G = (T \cup N, E)$ with every vertex in N having the same degree and a positive integer k.

Question: Is there a vertex subset $N' \subseteq N$ of size at most k such that every vertex in T has exactly one neighbor in N'?

Parameter: $|T|, k$.

Algorithm: $O(2^{(k \cdot |T|)} \cdot (|T| + |N|)^{O(1)}$.

Kernel: $O(2^{|T|})$, **no** $O((|T| + k)^{O(1)})$ **kernel** [DLS09].

Colored Version: N is colored with colors from $\{1, \ldots, k\}$ and N' is required to contain one vertex of each color.

Bounded Rank Disjoint Sets

Input: A set family $\mathcal{F}$ over a universe U with every set $S \in \mathcal{F}$ having size at most d, and a positive integer k.

Question: Is there a subfamily $\mathcal{F}'$ of of $\mathcal{F}$ of size at most k such that every pair of sets S_1, S_2 in $\mathcal{F}$ we have $S_1 \cap S_2 = \emptyset$?

Parameter: k, d.

Algorithm: $2^{O(kd)}(|\mathcal{F}| + |U|)^{O(1)}$ [DF99].

Kernel: $2^{O(kd)}$ [DF99], **no** $O((k + d)^{O(1)})$ **kernel** [DLS09].

Capacitated Vertex Cover (CapVC)

Input: A graph $G = (V, E)$ and a capacity function $cap : V \to \mathbb{N}^+$ and a positive integer k.

Question: Is there a vertex subset $C \subseteq V$ of size at most k and a function $f : E \to C$ that maps every edge to one of its endpoints and so that for all $v \in C$, $|f^{-1}(v)| \leqslant cap(v)$

Parameter: k.

Algorithm: $O(2^{k \log k} \cdot |V|^{O(1)})$ [DLSV08].

Kernel: $O(4^k \cdot k^2)$ [GNW07], **no** $O(k^{O(1)})$ **kernel** [DLS09].

Connected Vertex Cover (ConVC)

Input: A graph $G = (V, E)$ and a positive integer k.

Question: Is there a vertex subset $C \subseteq V$ of size at most k such that $G[C]$ is connected and every edge of G has at least one endpoint in C?

Parameter: k.

Algorithm: $O(2.7606^k \cdot |V|^{O(1)})$ [MRR06].

Kernel: $O(2^k + k + 2k^2)$, **no** $O(k^{O(1)})$ **kernel** [DLS09].

Dominating Set in Graphs with Small Vertex Cover

Input: A graph $G = (V, E)$ with a vertex cover of size t and an integer k.

Question: Is there a vertex set $V' \subseteq V$ of size at most k such that every vertex in $V \setminus V'$ has a neighbor in V'?

Parameter: k, t.

Algorithm: $2^t \cdot |V|^{O(1)}$.

Kernel: $O(2^t)$, **no** $O((k+t)^{O(1)})$ **kernel**.

Remark: Implies that Dominating Set in H-Minor Free Graphs parameterized by solution size k and $|H|$ does not have a $O((k+|H|)^{O(1)})$ kernel, and that Dominating Set parameterized by k and degeneracy d of the input graph has no $O((k+d)^{O(1)})$ kernel [DLS09].

Red-Blue Dominating Set (RBDS)

Input: A graph $G = (T \cup N, E)$ where both T and N are independent sets in G, and an integer k.

Question: Is there a vertex subset $N' \subseteq N$ of size at most k such that each vertex in T has at least one neighbor in N'?

Parameter: $|T|, k$.

Algorithm: $O(2^{|T|} \cdot |T \cup N|^{O(1)})$ [FKW04, Lemma 2].

Kernel: $O(2^{|T|} + |T|)$, **no** $O(k^{O(1)})$ **kernel** [DLS09].

Colored Version: N is colored with colors from $\{1, \ldots, k\}$ and N' is required to contain one vertex of each color.

Remark: Equivalent to Small Universe Set Cover.

Small Subset Sum

Input: An integer k, a set S of integers of size at most 2^k and an integer t.
Question: Is there a subset $S' \subseteq S$ with $|S'| \leqslant k$ such that $\sum_{y \in S'} y = t$?
Parameter: k.
Algorithm: $O(2^k \cdot |S|^{O(1)})$.
Kernel: $O(2^k)$, **no** $O(k^{O(1)})$ **kernel** [DLS09].

Small Universe Hitting Set

Input: A set family $\mathcal{F}$ over a universe U with $|U| = d$, and a positive integer k.
Question: Is there a subset $H \subseteq U$ of size at most k such that for every set $S \in \mathcal{F}$, $H \cap S \neq \emptyset$?
Parameter: k, d.
Algorithm: $O(2^d \cdot (|\mathcal{F}| + |U|)^{O(1)})$.
Kernel: $O(2^d)$, **no** $O((k+d)^{O(1)})$ **kernel** [DLS09].
Colored Version: U is colored with colors from $\{1, \ldots, k\}$ and H is required to contain one vertex of each color.
Remark: Implies that Hitting Set parameterized by solution size k and maximum set size d does not have a $O((k+d)^{O(1)})$ kernel.

Small Universe Set Cover

Input: A set family $\mathcal{F}$ over a universe U with $|U| = d$ and a positive integer k.
Question: Is there a subfamily $\mathcal{F}'$ of $\mathcal{F}$ of size at most k such that $\cup_{S \in \mathcal{F}'} S = U$?
Parameter: k, d.
Algorithm: $O(2^{|T|} \cdot (|(T \cup N|)^{O(1)})$ [FKW04, Lemma 2].
Kernel: $O(2^{|T|} + |T|)$, **no** $O(k^{O(1)})$ **kernel** [DLS09].
Remark: Equivalent to Red-Blue Dominating Set. Implies that Set Cover parameterized by solution size k and maximum set size d does not have a $O((k+d)^{O(1)})$ kernel.

Steiner Tree

Input: A graph $G = (T \cup N, E)$ and an integer k.

Question: Is there a vertex subset $N' \subseteq N$ of size at most k such that $G[T \cup N']$ is connected?

Parameter: $|T|, k$.

Algorithm: $O(2^{|T|} \cdot |T \cup N|^{O(1)})$ [BHKK07].

Kernel: $O(2^{|T|} + |T|)$, **no** $O(k^{O(1)})$ **kernel** [DLS09].

Unique Coverage

Input: A set family $\mathcal{F}$ over a universe U and a positive integer k.

Question: Is there a subfamily $\mathcal{F}'$ of $\mathcal{F}$ such that at least k elements of U are contained in exactly one set in $\mathcal{F}'$?

Parameter: k.

Algorithm: $O(4^{k^2}(|\mathcal{F}| + |U|)^{O(1)})$ [MRS07].

Kernel: 4^k [MRS07], **no** $O(k^{O(1)})$ **kernel** [DLS09].

Colored Version: Each set in $\mathcal{F}$ is colored with a colors from $\{1, \ldots, k\}$ and $\mathcal{F}'$ is required to contain exactly one set of each color.

Appendix B

A Kernel for Cycle Packing

Recall that the edge–disjoint cycles problem was the following:

Cycle Packing	
Instance:	Undirected graph $G = (V, E)$ and a non-negative integer k.
Parameter:	k.
Question:	Does G contain at least k edge-disjoint cycles?

We describe here a procedure for obtaining a kernel of $O(k^2 \log^2 k)$ vertices. We follow the presentation in [BDFH08].

We start by removing vertices of degree one and zero, and contracting all vertices of degree two to a neighbor. This gives an equivalent instance with all vertices of degree at least three. Erdös and Posa have shown in 1965 that there is a constant C, such that each graph has:

either a feedback vertex set of size at most $Ck\log k$,
or
at least k vertex disjoint cycles.

Now, we apply a 2-approximation algorithm for Feedback Vertex Set (e.g., from [BBF99, BG96]). If we obtain a feedback vertex set with at least $2Ck\log k$ vertices, then we can resolve the problem and answer 'yes'.

Otherwise, let S be a feedback vertex set of size less than $2Ck\log k$, and let F be the forest $G - S$.

We build a collection $\mathcal{C}$ of disjoint paths between vertices in S. If there are l disjoint paths between s, $s \in S$, then these paths give us $\lfloor l/2 \rfloor$ disjoint cycles. If we have $(2Ck\log k)^2 + 2k$ disjoint paths in $\mathcal{C}$, we have k disjoint cycles. This follows by applying the pigeon–hole principle on a collection of $(2Ck\log k)^2 + 1$ disjoint paths

to get one cycle, and iterating the argument k times on the remaining collection of paths. Thus, in this case, we can decide 'yes', else what we have is a kernel.

Each leaf x in F must be incident to two vertices in S (as its degree in G is three), and we take a path between these two neighbors in S with x as only other vertex in $\mathcal{C}$ (see Figure 1). Thus, we can assume there are $O(k^2 \log^2 k)$ leaves in F , and hence also $O(k^2 \log^2 k)$ vertices in F with at least three neighbors in F.

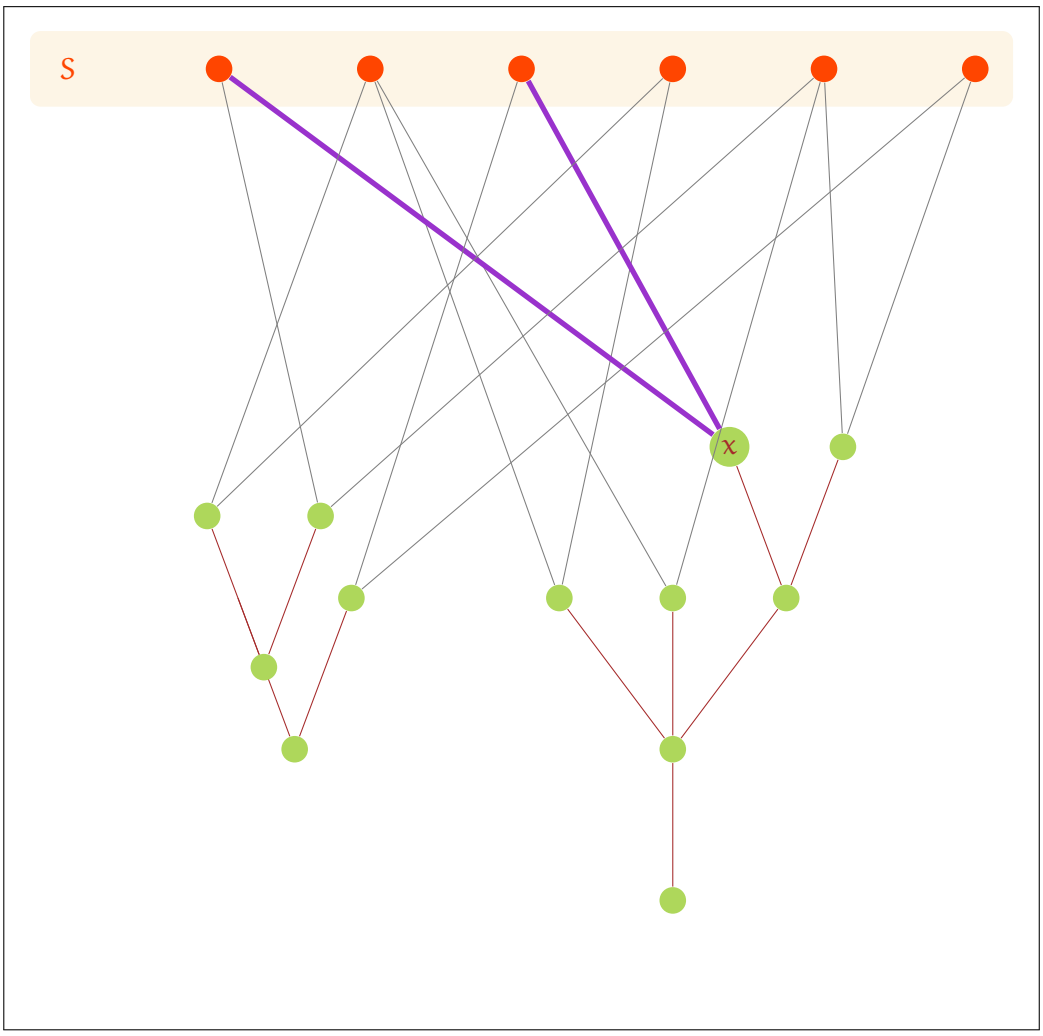


Figure 1: Getting paths from leaves in the forest F.

If F contains a path P with l vertices, each incident to exactly two vertices, then as each vertex in P is incident to a vertex in S, such a path gives us $\lfloor l/2 \rfloor$ paths in $\mathcal{C}$ (see

Figure 2).

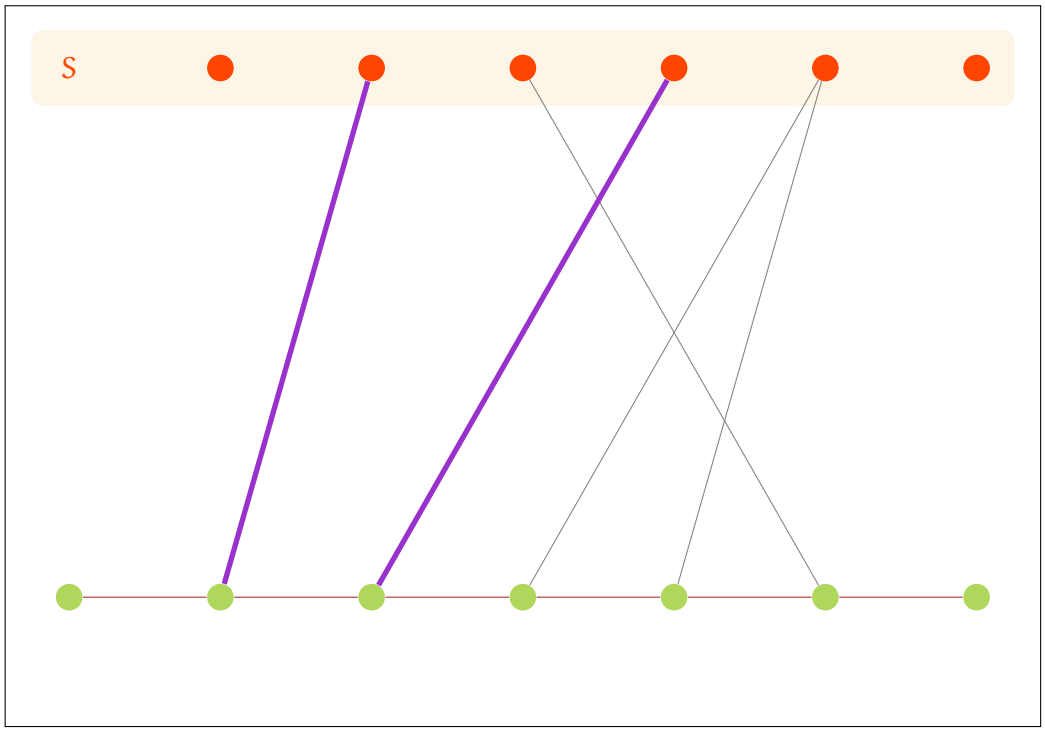


Figure 2: Getting paths from internal nodes in paths of the forest F.

As the number of such paths is bounded by the number of vertices in F with at least three neighbors in F and the leaves of F, we get a bound of $O(k^2 \log^2 k)$ on the number of vertices on these paths.

This construction shows that if F contains at least $5 \cdot (2Ck\log k)^2 + 2k$ vertices, there are k disjoint cycles, and thus we have a kernel of size $O(k^2 \log^2 k)$.

Appendix C

NP–completeness Proofs

In this section, we show the NP–completness of k-Leaf Out-Tree and Disjoint Factors.

We say that a subdigraph T of a digraph D is an *out-tree* if T is an oriented tree with only one vertex r of in-degree zero (called the *root*). The vertices of T of out-degree zero are called *leaves*. If T is a spanning out-tree, i.e., $V(T) = V(D)$, then T is called an *out-branching* of D. The Directed Maximum Leaf Out-Tree problem is to find an out-tree in a given digraph with the maximum number of leaves.

Recall that a *willow* graph [DV08] $D = (V, A_1 \cup A_2)$ is a directed graph such that $D' = (V, A_1)$ is a directed path $P = p_1 p_2 \dots p_n$ on all vertices of D and $D'' = (V, A_2)$ is a directed acyclic graph with one vertex r of in-degree 0, such that every arc of A_2 is a backwards arc of P. p_1 is called the *bottom* vertex of the willow, p_n is called the *top* of the willow and P is called the *stem*. A *nice willow* graph $D = (V, A_1 \cup A_2)$ is a willow graph where $p_n p_{n-1}$ and $p_n p_{n-2}$ are arcs of D, neither p_{n-1} nor p_{n-2} are incident to any other arcs of A_2 and $D'' = (V, A_2)$ has a p_n-out-branching.

Lemma 4. *[FFL+09]* k-Leaf Out-Tree *in nice willow graphs is NP-complete under Karp reductions.*

Proof. We reduce from the well known NP-complete Set Cover problem [Kar72]. A *set cover* of a universe U is a family $\mathcal{F}'$ of sets over U such that every element of u appears in some set in $\mathcal{F}'$. In the Set Cover problem one is given a family $\mathcal{F} = \{S_1, S_2, \dots S_m\}$ of sets over a universe U, $|U| = n$, together with a number $b \leqslant m$ and asked whether there is a set cover $\mathcal{F}' \subset \mathcal{F}$ with $|\mathcal{F}'| \leqslant b$ of U. In our reduction we will assume that every element of U is contained in at least one set in $\mathcal{F}$. We will also assume that $b \leqslant m-2$. These assumptions are safe because if either of them does not hold, the Set Cover instance can be resolved in polynomial time. From an instance of Set Cover we build a digraph $D = (V, A_1 \cup A_2)$ as follows. The vertex set V of D is a root r, vertices s_i for each $1 \leqslant i \leqslant m$ representing the sets in $\mathcal{F}$, vertices e_i, $1 \leqslant i \leqslant n$ representing elements in U and finally 2 vertices p and p'.

The arc set A_2 is as follows, there is an arc from r to each vertex s_i, $1 \leqslant i \leqslant m$ and there is an arc from a vertex s_i representing a set to a vertex e_j representing an element if $e_j \in S_i$. Furthermore, rp and rp' are arcs in A_2. Finally, we let $A_1 = \{e_{i+1}e_i : 1 \leqslant i < n\} \cup \{s_{i+1}s_i : 1 \leqslant i < m\} \cup \{e_1 s_m, s_1 p, pp', p'r\}$. This concludes the description of D. We now proceed to prove that there is a set cover $\mathcal{F}' \subset \mathcal{F}$ with $|\mathcal{F}'| \leqslant b$ if and only if there is an out-branching in D with at least $n + m + 2 - b$ leaves.

Suppose that there is a set cover $\mathcal{F}' \subset \mathcal{F}$ with $|\mathcal{F}'| \leqslant b$. We build a directed tree T rooted at r as follows. Every vertex s_i, $1 \leqslant i \leqslant m$, p and p' has r as their parent. For every element e_j, $1 \leqslant i \leqslant n$ we chose the parent of e_j to be s_i such that $e_j \in S_i$ and $S_i \in \mathcal{F}'$ and for every $i' < i$ either $S_{i'} \notin |\mathcal{F}'|$ or $e_j \notin S_{i'}$. Since the only inner nodes of T except for the root r are vertices representing sets in the set cover, T is an out-branching of D with at least $n + m + 2 - b$ leaves.

In the other direction suppose that there is an out-branching T of D with at least $n+m+2-b$ leaves, and suppose that T has the most leaves out of all out-branchings of D. Since D is a nice willow with r as top vertex, Observation 1 implies that T is an r-out-branching of D. Now, if there is an arc $e_{i+1}e_i \in A(T)$ then let s_j be a vertex such that $e_i \in S_j$. Then $T' = (T \setminus e_{i+1}e_i) \cup s_j e_i$ is an r-out-branching of D with as many leaves as T. Hence, without loss of generality, for every i between 1 and n, the parent of e_i in T is some s_j. Let $\mathcal{F}' = \{S_i : s_i \text{ is an inner vertex of } T\}$. $\mathcal{F}'$ is a set cover of U with size at most $n + m + 2 - (n + m + 2 - b) = b$, concluding the proof.

□

Recall that the Disjoint Factors problem is the following:

p-Disjoint Factors	
Instance:	A word $w \in L_k^*$.
Question:	Does w have the Disjoint Factors property?

where a word is said to have the "disjoint factors property" if it has non-overlapping i–factors for all $i \in [k]$ (an i–factor is a substring that begins and ends in the alphabet i).

Lemma 5. *[BDFH08]* Disjoint Factors *is NP-complete under Karp reductions.*

Proof. Clearly, the problem belongs to NP. We show NP–hardness by a transformation from 3–CNF–satisfiability. Let F be a 3–SAT formula with $c+1$ clauses $C_0, \ldots, C_c$. We start our construction of our word W with the prefix

$$123123123456456456 \ldots (3c{+}1)(3c{+}2)(3c{+}3)(3c{+}1)(3c{+}2)(3c{+}3)(3c{+}1)(3c{+}2)(3c{+}3).$$

Here the factor

$$(3i+1)(3i+2)(3i+3)(3i+1)(3i+2)(3i+3)(3i+1)(3i+2)(3i+3)$$

corresponds to the clause C_i, for all $i = 0, \ldots, c$.

Note that the factor 123123123 does not have the disjoint factor property, but fails it only by one. Indeed, one can find two disjoint factors $\{F_1, F_2\}$, or $\{F_1, F_3\}$, etc, but not three disjoint factors $\{F_1, F_2, F_3\}$. Hence, in this prefix of W, one can find two disjoint factors for each clause, but not three.

Each variable appearing in F is a letter of our alphabet. In addition to W, we concatenate for each variable x a factor to W of the form $xp_1p_1p_2p_2 \ldots p_lp_lxq_1q_1 \ldots q_mq_mx$ where the p_i's are the position in which x appears as a positive literal, and the q_i's are the position in which x appears as a negative literal.

The following is an example of the reduction above: let F be the formula

$$F = (x \vee y \vee z) \wedge (y \vee x \vee z) \wedge (x \vee y \vee z),$$

Then the corresponding word is

$$123123123456456456789789789x77x1155xy4488y22yz3399z66z$$

It is easily verified that, given this construction, F is satisfiable if and only if W_F, the word obtained as described above, has the disjoint factor property. This proves the claim of NP–completeness.

□

Appendix D

Dynamic Programming For Disjoint Factors

During the proof of compositionality of Disjoint Factors, we claimed that there is a FPT algorithm that can solve the problem in $O(nk \cdot 2^k)$ time. Inspired by the outline in [BDFH08], we provide such an algorithm in this section.

Recall that the parameterized version of the Disjoint Factors was the following:

p-Disjoint Factors	
Instance:	A word $w \in L_k^*$.
Parameter:	$k \geqslant 1$.
Question:	Does w have the Disjoint Factors property?

We perform dynamic programming over the subsets of L. In particular, we define a table of dimension $2^k \cdot n$, with columns indexed by the subsets of L, arranged in increasing order of size (whenever incomparable, the ordering is immaterial), and rows indexed by elements from $[n]$ in increasing order. An example is demonstrated in Figure 6.2. Let W denote the given word, and let W_i denote the substring of W upto length i (from the beginning of W). Also, let w_i denote the i^{th} letter in the string W.

We populate the entries of the table with quantities $A(S, i)$ and $B(S, i, x)$, defined as below:

$A(S, i) = 1$ if the word W_i has disjoint factors F_j for all $j \in S$, 0 otherwise.

$B(S, i, x) = 1$ if the word W_i has disjoint factors F_j for all $j \in S$ and the word W_i has a x–factor, disjoint from all the j–factors, $j \in S$, and ending at w_i; 0 otherwise.

$$k = 3, L = 1, 2, 3$$

$$S_1 = \{1\}, S_2 = \{2\}, S_3 = \{3\}$$
$$S_4 = \{1, 2\}, S_5 = \{1, 3\}, S_6 = \{2, 3\}$$
$$S_7 = \{1, 2, 3\}$$
$$w = 121333213, |w| = 9.$$

	S_1	S_2	S_3	S_4	S_5	S_6	S_7
1							
12							
121							
1213							
12133							
121333							
1213332							
12133321							
121333213							

Figure 3: Dynamic Programming for Disjoint Factors

Clearly, an examination of the value of $A([k], n)$, the A–entry in the bottom-right entry of the table, tells us if the word is a YES–instance.

We fill the table from left to right, top to bottom, using the reccurrences:

$$A(S, i) = max\{A(S, i-1), B(S \setminus w_i, i-1, w_i)\}$$

and

$$B(S, i, x) = A(S, i-j), \text{ if } w_i = x,$$
$$0 \text{ otherwise,}$$

where j is the largest index for which $w_j = w_i$, in other words, the shortest x–factor "starting" from the right.

Note that $A(S, i) = A(S, i-1)$ whenever $w_i \notin S$. However, when $w_i \in S$, then the witness w_i–factor, if it exists, may or may not be one that ends at i, hence we check both possibilities.

It is clear that it takes time $2^k \cdot nk$ to fill the entries in table (as we may be storing as many as $(k-1)$ B–values and 1 A–value in some of the cells). Thus an algorithm that runs in time $O(n^2 k \cdot 2^k)$ is immediate (as identifying j for filling in B can potentially involve a scan of the entire string, which is $O(n)$ in the worst case). However, this factor of n may be avoided by using a collection of k pointers to "remember" the last i–factor seen in the word, for all $i \in [k]$. This modification can be easily implemented.

Appendix E

Distillations and NP–Complete Problems

We give an outline of the proof of Theorem 4. Recall that we would like to prove:

If any NP-complete problem has a distillation algorithm then
$\mathsf{PH} = \Sigma_3^p$.[[BDFH08, FS08, CFM07]]

We follow the presentation in [BDFH08].

Proof. Let L be an NP–complete problem with a distillation algorithm $\mathcal{A}$, and let $\overline{L}$ denote the complement of L. We show that $\mathsf{PH} = \Sigma_3^p$ by showing that $\mathsf{coNP} \subseteq \mathsf{NP/poly}$, this is sufficient because of Yap's theorem [Yap83].

Recall that a distillation algorithm for L is a polynomial time algorithm $\mathcal{A}$ that receives as inputs finite sequences $x = (x_1, \ldots, x_t)$ with $x_i \in \{0,1\}^*$ for $i \in [t]$ and outputs a string $\mathcal{A}(\overline{x}) \in \{0,1\}^*$ such that

1. $|\mathcal{A}(\overline{x})| = (\max_{i \in [t]} |x_i|)^{O(1)}$
2. $\mathcal{A}(\overline{x}) \in L$ if and only if for some $i \in [t] : x_i \in L$.

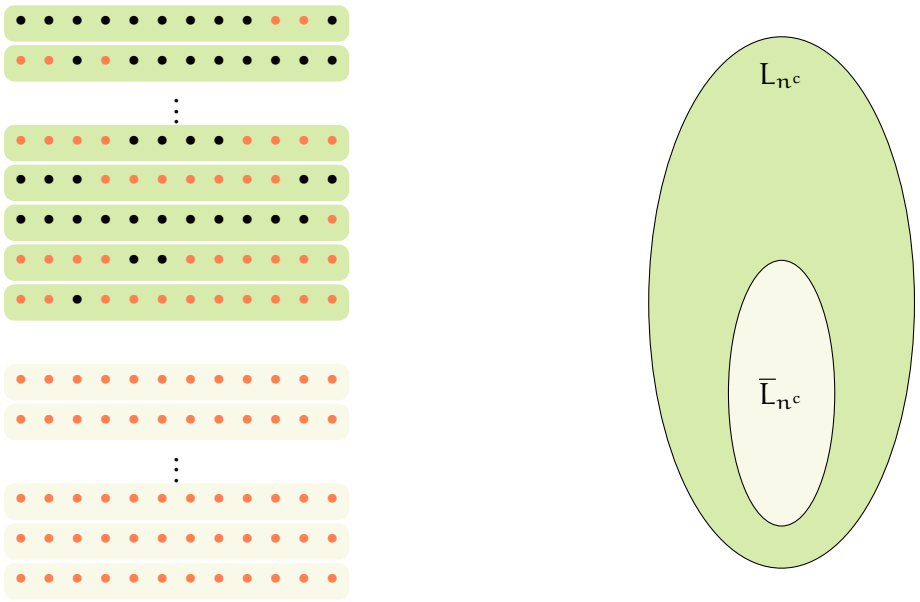
So what we have is a distillation $\mathcal{A}$, which, if thought of as a function from $\{0,1\}^{*^t} \rightarrow \{0,1\}^*$, has the notable property that it will map any subset of t strings in $\overline{L}$ to a string in $\overline{L}$.

To show that $\mathsf{coNP} \subseteq \mathsf{NP/poly}$. we start with a language in coNP, and design a non-deterministic Turing Machine (NDTM) that (with the help of polynomial advice) can decide it in polynomial time.

A natural candiate coNP language is $\overline{L}$. A natural candidate algorithm to decide $\overline{L}$ must presumably exploit the distillation $\mathcal{A}$ and some polynomial advice. We first discuss what kind of advice sets we may be working with.

Let $n \in \mathbb{N}$ be a suffciently large integer. Denote by $\overline{L}_n$ the subset of strings of length at most n in the complement of L, i.e. $L_n = \{x \in \overline{L} || x| \leqslant n\}$. By its definition, given any $x_1, \ldots, x_t \in \overline{L}_n$, the distillation algorithm $\mathcal{A}$ maps $(x_1, \ldots, x_t)$ to some $y \in \overline{L}_{n^c}$, where c is some constant independent of t.

On the other hand, any sequence containing a string $x_i \in L_n$ is mapped to a string $y \in L_{n^c}$.



To the left are tuples of strings of length n. The dots labelled orange (•) are strings that do not belong to L_n. Any distillation algorithm will map the tuples marked green to L_{n^c}. Notice that if a tuple has even one string in L_n (•), then it gets mapped outside $\overline{L}_{n^c}$.

Figure 4: Thinking of Distillation as a Map

Suppose we have a set $S_n \subseteq \overline{L}_{n^c}$, with $|S_n|$ polynomially bounded in n, such that for any $x \in \Sigma_{\leqslant n}$, we have the following:

1. If $x \in \overline{L}_n$, then there exist strings $x_1, \ldots, x_t \in \Sigma_{\leqslant n}$ with $x_i = x$ for some $i, 1 \leqslant i \leqslant t$, such that $\mathcal{A}(x_1, \ldots, x_t) \in S_n$.

2. If $x \notin \overline{L}_n$, then for all strings $x_1, \ldots, x_t \in \Sigma_{\leqslant n}$ with $x_i = x$ for some $i, 1 \leqslant i \leqslant t$, such that $\mathcal{A}(x_1, \ldots, x_t) \notin S_n$.

Then it would be quite straightforward to construct a non-deterministic Turing Machine that can, with the help of this set as its polynomial advice, decide $\overline{L}$ polynomial time. First guess t strings $x_1, \ldots, x_t \in \Sigma_{\leqslant n}$, and check whether one of them is x. If not, immediately reject. Otherwise, compute $\mathcal{A}(x_1, \ldots, x_t)$, and accepts if and only if the output is in S_n. It is immediate to verify that this procedure correctly determines (in the non-deterministic sense) whether $x \in \overline{L}_n$.

We now show that there exists such an advice $S \subseteq \overline{L}_{n^c}$ as required above.

We view $\mathcal{A}$ as a function mapping strings from $(\overline{L}_n)^t$ to $\overline{L}_{n^c}$, as suggested before. Let's say that a string $y \in \overline{L}_{n^c}$ covers a string $x \in \overline{L}_n$ if there exist $x_1, \ldots, x_t \in \Sigma_{\leqslant n}$ with $x_i = x$ for some $i, 1 \leqslant i \leqslant t$, and with $\mathcal{A}(x_1, \ldots, x_t) = y$. Clearly, our goal is to find polynomial-size subset of $\overline{L}_{n^c}$ which covers all strings in $\overline{L}_n$.

By the pigeonhole principle, there is a string $y \in Y$ for which $\mathcal{A}$ maps at least $|(\overline{L}_n)^t|/|\overline{L}_{n^c}| = |\overline{L}_n|^t/|\overline{L}_{n^c}|$ tuples in $(\overline{L}_n)^t$ to. Taking the t^{th} square root, this gives us $|\overline{L}_n|/|\overline{L}_{n^c}|^{1/t}$ distinct strings in $\overline{L}_n$ which are covered by y. Hence, by letting $t = \log|\overline{L}_{n^c}| = O(n^c)$, this gives us a constant fraction of the strings in $\overline{L}_n$.

It follows that we can repeat this process recursively in order to cover all strings in $\overline{L}_n$ with a polynomial number of strings in $\overline{L}$.

□

Bibliography

[ACK+00] G. Ausiello, P. Crescenzi, V. Kann, Marchetti-Sp, Giorgio Gambosi, and Alberto M. Spaccamela, *Complexity and approximation: Combinatorial optimization problems and their approximability properties*, Springer, January 2000. 4

[AFG+07a] Noga Alon, Fedor V. Fomin, Gregory Gutin, Michael Krivelevich, and Saket Saurabh, *Better algorithms and bounds for directed maximum leaf problems*, FSTTCS (Vikraman Arvind and Sanjiva Prasad, eds.), Lecture Notes in Computer Science, vol. 4855, Springer, 2007, pp. 316–327. 77

[AFG+07b] _____, *Parameterized algorithms for directed maximum leaf problems*, ICALP (Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, eds.), Lecture Notes in Computer Science, vol. 4596, Springer, 2007, pp. 352–362. 77, 78

[AFN02] Jochen Alber, Michael R. Fellows, and Rolf Niedermeier, *Efficient data reduction for dominating set: a linear problem kernel for the planar case*, Proceedings of the 8th Scandinavian Workshop on Algorithm Theory (SWAT'2002), Lecture Notes in Computer Science, no. 2368, Springer, July 2002, pp. 150–159. 15

[AKF06] Faisal N. Abu-Khzam and Henning Fernau, *Kernels: Annotated, proper and induced*, IWPEC, 2006, pp. 264–275. 19

[AYZ95] Noga Alon, Raphael Yuster, and Uri Zwick, *Color-coding*, J. ACM **42** (1995), no. 4, 844–856. 8, 41, 91

[BBF99] Bafna, Berman, and Fujito, *A 2-approximation algorithm for the undirected feedback vertex set problem*, SIJDM: SIAM Journal on Discrete Mathematics **12** (1999). 97

[BD08] Paul S. Bonsma and Frederic Dorn, *Tight bounds and a fast FPT algorithm for directed max-leaf spanning tree*, ESA (Dan Halperin and

Kurt Mehlhorn, eds.), Lecture Notes in Computer Science, vol. 5193, Springer, 2008, pp. 222–233. 77

[BDFH08] Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Danny Hermelin, *On problems without polynomial kernels (extended abstract)*, ICALP (1), 2008, pp. 563–574. 25, 26, 27, 31, 37, 42, 61, 66, 92, 97, 103, 105, 109

[BG96] Ann Becker and Dan Geiger, *Optimization of Pearl's method of conditioning and greedy-like approximation algorithms for the vertex feedback set problem*, Artificial Intelligence **83** (1996), no. 1, 167–188. 97

[BHKK07] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto, *Fourier meets möbius: fast subset convolution*, STOC (David S. Johnson and Uriel Feige, eds.), ACM, 2007, pp. 67–74. 96

[Bod87] H. L. Bodlaender, *Dynamic programming on graphs with bounded treewidth*, Tech. report, Cambridge, MA, USA, 1987. 8

[Bod07] Hans L. Bodlaender, *A cubic kernel for feedback vertex set*, STACS, 2007, pp. 320–331. 64

[BTY08] Hans L. Bodlaender, Stéphan Thomassé, and Anders Yeo, *Analysis of data reduction: Transformations give evidence for non-existence of polynomial kernels*, Tech. Report UU-CS-2008-030, Department of Information and Computing Sciences, Utrecht University, 2008. 64

[CC08] Miroslav Chlebík and Janka Chlebíková, *Crown reductions for the minimum weighted vertex cover problem*, Discrete Appl. Math. **156** (2008), no. 3, 292–312. 18

[CDLS02] Marco Cadoli, Francesco M. Donini, Paolo Liberatore, and Marco Schaerf, *Preprocessing of intractable problems*, Inf. Comput. **176** (2002), no. 2, 89–120. 16

[CFKX05] Jianer Chen, Henning Fernau, Iyad A. Kanj, and Ge Xia, *Parametric duality and kernelization: Lower bounds and upper bounds on kernel size*,

STACS (Volker Diekert and Bruno Durand, eds.), Lecture Notes in Computer Science, vol. 3404, Springer, 2005, pp. 269–280. 24

[CFM07] Yijia Chen, Jörg Flum, and Moritz Müller, *Lower bounds for kernelizations*, Electronic Colloquium on Computational Complexity (ECCC) **14** (2007), no. 137. 19, 21, 25, 28, 37, 46, 91, 92, 109

[CG05] L. Sunil Chandran and Fabrizio Grandoni, *Refined memorization for vertex cover*, Inf. Process. Lett. **93** (2005), no. 3, 125–131. 6

[CK95] Richard Chang and Jim Kadin, *On computing boolean connectives of characteristic functions*, Mathematical Systems Theory **28** (1995), no. 3, 173–198. 27

[Coo71] Stephen A. Cook, *The complexity of theorem-proving procedures*, STOC '71: Proceedings of the third annual ACM symposium on Theory of computing (New York, NY, USA), ACM, 1971, pp. 151–158. 62

[DF95a] R. G. Downey and M. R. Fellows, *Parameterized computational feasibility*, P. Clote, J. Remmel (eds.): Feasible Mathematics II, Boston: Birkhäuser, 1995, pp. 219–244. 61

[DF95b] Rod G. Downey and Michael R. Fellows, *Fixed-parameter tractability and completeness II: On completeness for W*[1], Theor. Comput. Sci. **141** (1995), no. 1-2, 109–131. 61

[DF99] Rod G. Downey and M. R. Fellows, *Parameterized complexity*, Springer, November 1999. 8, 14, 93

[DFst] Rod G. Downey and Michael R. Fellows, *Fixed-parameter tractability and completeness I: Basic results*, SIAM J. Comput. **24** (1995, August), no. 4, 873–921. 61

[DGKY08] Jean Daligault, Gregory Gutin, Eun Jung Kim, and Anders Yeo, *FPT algorithms and kernels for the directed* k*-leaf problem*, CoRR **abs/0810.4946** (2008), informal publication. 77

[DLS09] Michael Dom, Daniel Lokshtanov, and Saket Saurabh., *Incompressibility through colors and ids*, ICALP, Lecture Notes in Computer Science, Springer, 2009. 52, 66, 72, 73, 86, 88, 93, 94, 95, 96

[DLSV08] Michael Dom, Daniel Lokshtanov, Saket Saurabh, and Yngve Villanger, *Capacitated domination and covering: A parameterized perspective*, IWPEC (Martin Grohe and Rolf Niedermeier, eds.), Lecture Notes in Computer Science, vol. 5018, Springer, 2008, pp. 78–90. 93

[DV08] Matthew Drescher and Adrian Vetta, *An approximation algorithm for the maximum leaf spanning arborescence problem*, ACM Transactions on Algorithms (2008), In Press. 77, 81, 101

[Fel06] Michael R. Fellows, *The lost continent of polynomial time: Preprocessing and kernelization*, IWPEC, 2006, pp. 276–277. 13

[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*, STACS, 2009, pp. 421–432. 77, 78, 101

[FG06] J. Flum and M. Grohe, *Parameterized complexity theory*, Springer-Verlag New York Inc, 2006. 8, 14, 16

[FGK+08] Fedor V. Fomin, Serge Gaspers, Dieter Kratsch, Mathieu Liedloff, and Saket Saurabh, *Iterative compression and exact algorithms*, MFCS '08: Proceedings of the 33rd international symposium on Mathematical Foundations of Computer Science (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 335–346. 7

[FKW04] Fedor V. Fomin, Dieter Kratsch, and Gerhard J. Woeginger, *Exact (exponential) algorithms for the dominating set problem*, WG (Juraj Hromkovic, Manfred Nagl, and Bernhard Westfechtel, eds.), Lecture Notes in Computer Science, vol. 3353, Springer, 2004, pp. 245–256. 68, 94, 95

[FS08] Lance Fortnow and Rahul Santhanam, *Infeasibility of instance compression and succinct pcps for np*, STOC '08: Proceedings of the 40th an-

nual ACM symposium on Theory of computing (New York, NY, USA), ACM, 2008, pp. 133–142. 24, 27, 109

[GGHN03] Jens Gramm, Jiong Guo, Falk Hüffner, and Rolf Niedermeier, *Graph-modeled data clustering: Fixed-parameter algorithms for clique generation*, In Proc. 5th CIAC, volume 2653 of LNCS, Springer, 2003, pp. 108–119. 18

[GJ79] Michael R. Garey and David S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, 1979. 51, 68

[GN07] Jiong Guo and Rolf Niedermeier, *Invitation to data reduction and problem kernelization*, SIGACT News **38** (2007), no. 1, 31–45. 19

[GNW07] J. Guo, R. Niedermeier, and S. Wernicke, *Parameterized complexity of vertex cover variants*, ACM Transactions on Computer Systems **41** (2007), 501–520. 93

[Gro02] Martin Grohe, *Parameterized complexity for the database theorist*, SIGMOD Rec. **31** (2002), no. 4, 86–96. 8

[HKMN08] Falk Hüffner, Christian Komusiewicz, Hannes Moser, and Rolf Niedermeier, *Fixed-parameter algorithms for cluster vertex deletion*, LATIN (Eduardo Sany Laber, Claudson F. Bornstein, Loana Tito Nogueira, and Luerbio Faria, eds.), Lecture Notes in Computer Science, vol. 4957, Springer, 2008, pp. 711–722. 7

[HN06] Danny Harnik and Moni Naor, *On the compressibility of np instances and cryptographic applications*, FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (Washington, DC, USA), IEEE Computer Society, 2006, pp. 719–728. 24

[HNW08] Falk Hüffner, Rolf Niedermeier, and Sebastian Wernicke, *Techniques for practical fixed-parameter algorithms*, The Computer Journal **51** (2008), no. 1, 7–25. 8

[Hoc97] Dorit S. Hochbaum (ed.), *Approximation algorithms for NP-hard problems*, PWS Publishing Co., Boston, MA, USA, 1997. 4

[Kar72] R. M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations (R. E. Miller and J. W. Thatcher, eds.), Plenum Press, 1972, pp. 85–103. 62, 101

[Kho02] Subhash Khot, *On the power of unique 2-prover 1-round games*, Proceedings of the 34th Annual ACM Symposium on Theory of Computing, STOC'2002 (Montreal, Quebec, Canada, May 19-21, 2002) (New York), ACM Press, 2002, pp. 767–775. 24

[KLR08] Joachim Kneis, Alexander Langer, and Peter Rossmanith, *A new algorithm for finding trees with many leaves*, ISAAC (Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, eds.), Lecture Notes in Computer Science, vol. 5369, Springer, 2008, pp. 270–281. 77

[KR08] Subhash Khot and Oded Regev, *Vertex cover might be hard to approximate to within 2-ϵ*, J. Comput. Syst. Sci. **74** (2008), no. 3, 335–349. 24

[Lib01] Paolo Liberatore, *Monotonic reductions, representative equivalence, and compilation of intractable problems*, J. ACM **48** (2001), no. 6, 1091–1125. 16

[Mar08] Daniel Marx, *Parameterized complexity and approximation algorithms*, The Computer Journal **51** (2008), no. 1, 60–78. 8

[MR96] Rajeev Motwani and Prabhakar Raghavan, *Randomized algorithms*, ACM Comput. Surv. **28** (1996), no. 1, 33–37. 4

[MRR06] Daniel Mölle, Stefan Richter, and Peter Rossmanith, *Enumerate and expand: Improved algorithms for connected vertex cover and tree cover*, CSR (Dima Grigoriev, John Harrison, and Edward A. Hirsch, eds.), Lecture Notes in Computer Science, vol. 3967, Springer, 2006, pp. 270–280. 94

[MRS07] Hannes Moser, Venkatesh Raman, and Somnath Sikdar, *The parameterized complexity of the unique coverage problem*, ISAAC (Takeshi Tokuyama, ed.), Lecture Notes in Computer Science, vol. 4835, Springer, 2007, pp. 621–631. 96

[Nie06] Rolf Niedermeier, *Invitation to fixed parameter algorithms (oxford lecture series in mathematics and its applications)*, Oxford University Press, USA, March 2006. 5, 6, 7, 8, 14, 15, 18, 24

[PRS09] Geevarghese Philip, Venkatesh Raman, and Somnath Sikdar, *Polynomial kernels for dominating set in* $K_{i,j}$*-free and d-degenerate graphs*, CoRR **abs/0903.4521** (2009), informal publication. 86

[Ram97] Venkatesh Raman, *Parameterized complexity*, Proceedings of the 7th National Seminar on Theoretical Computer Science (Chennai, India), 1997. 8

[RSV04] Bruce A. Reed, Kaleigh Smith, and Adrian Vetta, *Finding odd cycle transversals*, Oper. Res. Lett. **32** (2004), no. 4, 299–301. 7

[Tho09] Stéphan Thomassé, *A quadratic kernel for feedback vertex set*, SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 2009, pp. 115–119. 18, 64

[Wei98] Karsten Weihe, *Covering trains by stations or the power of data reduction*, OnLine Proceedings of ALEX'98 - 1st Workshop on Algorithms and Experiments, 1998. 15

[Woe03] Gerhard J. Woeginger, *Exact algorithms for np-hard problems: A survey*, Proc. 5th Int. Worksh. Combinatorial Optimization – Eureka, You Shrink (Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi, eds.), Lecture Notes in Computer Science, no. 2570, Springer-Verlag, 2003, pp. 185–207. 4

[Yap83] Chee-Keng Yap, *Some consequences of non-uniform conditions on uniform classes*, Theor. Comput. Sci. **26** (1983), 287–300. 27, 109