Beyond the Worst-Case Analysis of Algorithms

Edited by
Tim Roughgarden

Contents

1
Parameterized Algorithms

Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi

Abstract

Parameterized algorithmics analyzes running time in finer detail than classical complexity theory: instead of expressing the running time of an algorithm as a function of the input size only, dependence on one or more parameters of the input instance is taken into account. In this chapter we sketch some techniques and tools from this rapidly developing area.

1.1 Introduction

Worst-case running time analysis has been at the center of nearly all developments in theoretical computer science since the inception of the field. Nevertheless, this approach to measuring algorithm efficiency has its own drawbacks. It is almost never the case that the input size is the only feature of the input instance that affects the running time of an algorithm. Further, it is rarely the case that the input instances we actually want to solve look like the instances on which the algorithm performs the worst. For this reason, the running time estimates from a worst-case analysis can be overly pessimistic, and algorithms with optimized worst-case behavior often perform poorly on instances arising in applications. Real-world instances are not worst-case instances; they exhibit additional structure that can often be exploited algorithmically. Almost all areas of applications of algorithms are full of parameters. Example parameters include size, topology, shape, depth of the formula, and so on. Parameterized complexity systematically seeks to understand the contribution of such parameters to the overall complexity of the problem. That is, the goal of parameterized complexity is to find ways of solving NP-hard problems more efficiently than brute force: our aim is to restrict the combinatorial explosion to a parameter that is hopefully much smaller than the input size.

1.1.1 Warm-Up: Vertex Cover

Without doubt, Vertex Cover is the most popular problem in parameterized complexity and this is why many people call it the *drosophila melanogaster* of parameterized algorithmics. This is because Vertex Cover is the "simplest" among all parameterized problems. By simplest we mean the following empirical fact: When designing some new algorithmic technique for a parameterized problem it is always useful to check how this technique could be applied on Vertex Cover.

Recall that a *vertex cover* S of a graph G is a set of vertices "covering" every edge of G. In other words, the graph $G - S$ obtained from G by removing the vertices of S has no edges. In the Vertex Cover problem, we are given a graph G and an integer k. The question is to decide whether G contains a vertex cover of size k. Moreover, most of the known algorithms, if the pair (G, k) is a yes-instance, can actually construct the corresponding vertex cover.

Vertex Cover is an NP-complete problem, so it is very unlikely that it will admit a polynomial time algorithm. On the other hand, deciding whether a graph has a vertex cover of size at most 2 can be clearly done in time $\mathcal{O}(n^2 \cdot m)$, which is polynomial in the input size.[1] We just try all pairs of vertices and for every pair we check whether there is an edge not covered by this pair. This running time can be easily improved to $\mathcal{O}(n^2 \cdot n)$ by making use of the following observation: A pair u, v is a vertex cover if and only if for every vertex $w \notin \{u, v\}$ its adjacency list contains no vertices but u and v. Thus if we have a vertex w whose adjacency list is longer than 2, we know that $\{u, v\}$ is not a vertex cover. Otherwise, going through all the lists takes time $\mathcal{O}(n^2 \cdot n)$. This is clearly a polynomial time algorithm.

In general, an algorithm where we enumerate all vertex subsets of size at most k and check whether any of them forms a vertex cover solves the problem in time $O(n^k \cdot k \cdot n)$, which is polynomial for every constant k. We also know that unless $\mathrm{P} \neq \mathrm{NP}$, when k is unbounded, Vertex Cover cannot be solved in polynomial time. This sounds like the end of the story.

A bit surprisingly, this is not the end. We can show that Vertex Cover can be solved in *linear* time for every fixed k. Moreover, Vertex Cover can be solved in polynomial time even when $k = \mathcal{O}(\log n)$. We do it by introducing a bounded depth search tree (recursive) algorithm for the problem.

One of the simplest parameterized algorithms solving Vertex Cover is a recursive algorithm often called the *bounded search tree* or *branching algorithm.* The algorithm is based on the following two observations.

- For a vertex v, any vertex cover must contain either v or *all* of its neighbors $N(v)$.
- Vertex Cover becomes trivial (in particular, can be solved in polynomial time) when the maximum degree of a graph is at most 1.

[1] In what follows, we will always use n and m to denote the number of vertices and the number of edges in the graph, respectively.

The algorithm now proceeds recursively, where G and k will be modified before passing to a recursive call. If in some recursive branch graph G has at least one edge and parameter $k \leq 0$, this instance has no vertex cover of size k and we halt in this branch. Otherwise, we find a vertex $v \in V(G)$ of maximum degree in G. If v is of degree 0, then G has no edges and we found a solution. Otherwise, we recursively branch on two cases by considering either v or $N(v)$ in the vertex cover. In the branch where v is in the vertex cover, we can delete v and decrease the parameter k by 1. In the second branch, we add $N(v)$ to the vertex cover, delete $N(v) \cup \{v\}$ from the graph and decrease k by $|N(v)|$. Since $|N(v)| \geq 1$, in each of the branches we decrease k by at least 1.

To analyze the running time of the algorithm, it is convenient to view this recursive algorithm as a search tree $\mathcal{T}$. The root of this tree corresponds to the initial instance (G, k) and for every node of the tree its children correspond to instances in the recursive calls. Then the running time of the algorithm is bounded by (the number of nodes in the search tree) $\times$ (time taken at each node). It is easy to implement the algorithm so the time taken at each node is bounded by $\mathcal{O}(k \cdot n)$, where n is the number of vertices in G. Thus, if $\tau(k)$ is the number of nodes in the search tree, then the total time used by the algorithm is at most $\mathcal{O}(\tau(k) \cdot k \cdot n)$. Observe that in each recursive call we reduce the parameter k by at least 1. Hence the height of the tree does not exceed k, and hence $\tau(k) \leq 2^{k+1} - 1$.

The above discussions bring us to the following theorem.

Theorem 1.1 Vertex Cover *is solvable in time* $\mathcal{O}(2^k \cdot k \cdot n)$.

Since for every constant k, we have that $\mathcal{O}(2^k \cdot k \cdot n) = \mathcal{O}(n)$, the running time of Theorem 1.1 is linear for every fixed k. Also, for $k = c \log n$, we have $\mathcal{O}(2^k \cdot k \cdot n) = \mathcal{O}(2^{c \log n} \cdot \log n \cdot n) = \mathcal{O}(n^{c+1} \cdot \log n)$, which is polynomial in n.

Let us remark that $\mathcal{O}(2^k \cdot k \cdot n)$ is not the best running time bound for Vertex Cover and the algorithm can be improved easily. For example, the following observation can be helpful. If all the vertices of a graph are of degree at most 2, then the graph is the disjoint union of cycles and paths. In this case, the minimum vertex cover can be easily found in polynomial time (how?). On the other hand, if the graph has a vertex of degree at least 3, then branching on this vertex provides us with a better recurrence and it is possible to show that in this case the branching tree has $\mathcal{O}(1.4656^k)$ vertices. We leave the formal proof of this claim as an exercise (Exercise 1.1). The best known algorithm solves the problem in time $\mathcal{O}(1.2738^k + kn)$; it is based on a combination of kernelization (see Section 1.4) and clever branching techniques [Chen et al. (2010)].

Algorithms with running time $f(k) \cdot n^c$, for a constant c independent of both n and k, are called *fixed-parameter algorithms*, or FPT algorithms. Typically the goal in parameterized algorithmics is to design FPT algorithms, trying to make both the $f(k)$ factor and the constant c in the bound on the running time as small as possible. FPT algorithms can be contrasted with less efficient XP algorithms (for *slice-wise polynomial*), where the running time is of the form $f(k) \cdot n^{g(k)}$, for some functions f, g. There is a tremendous difference in running times of the form $f(k) \cdot n^{g(k)}$ and $f(k) \cdot n^c$.

Inspired by the success we had with Vertex Cover, it is natural to ask whether a similar improvement over brute-force is possible for every NP-hard problem, for every choice of parameter. Of course, this is not true. As an example, consider Vertex Coloring. Here we are given as input a graph G and an integer k, and we need to decide whether G has a proper k-coloring—that is, a coloring where no two adjacent vertices obtain the same color. It is well known that Vertex Coloring is NP-complete already for $k = 3$, so we do not hope for a polynomial-time algorithm for fixed k. Observe that even an XP algorithm with running time $f(k) \cdot n^{g(k)}$ for any functions f and g would imply that P= NP.

The example of Vertex Coloring illustrates that parameterized algorithms may not be all-powerful: there are parameterized problems that do not seem to admit FPT algorithms. However, in this specific example, we could explain very precisely why we are not able to design efficient algorithms, even when the number of colors is small. From the perspective of algorithm designers such an insight is very useful; they can now stop wasting their time trying to design efficient algorithms based only on the fact that the number of colors is small, and start searching for other ways to attack the problem instances. If we are trying to design a polynomial-time algorithm for a problem and failing, it is quite likely that this is because the problem is NP-hard. Is the theory of NP-hardness the right tool also for giving negative evidence for fixed-parameter tractability? In particular, if we are trying to design an $f(k) \cdot n^c$-time algorithm and fail to do so, is it because the problem is NP-hard for some fixed constant value of k, say $k = 100$? Let us look at another example, the Clique problem.

In the Clique problem we are given as input graph G and integer k, and the task is to decide whether G contains a clique on k vertices, that is, a set of k vertices with an edge between every pair of them. Similar to Vertex Cover, there is a simple brute-force $n^{\mathcal{O}(k)}$-time algorithm to check whether there is a clique on at least k vertices. Can we design an FPT algorithm for this problem? After some reflection one can see that the NP-hardness of Clique cannot be used to rule out an FPT algorithm for it.

Since NP-hardness is insufficient to differentiate between problems with $f(k) \cdot n^{g(k)}$-time algorithms and problems with $f(k) \cdot n^c$-time algorithms, we have to resort to stronger complexity-theoretic assumptions. The theory of *W[1]-hardness* allows us to prove (under certain complexity assumptions) that even though a problem is polynomial-time solvable for every fixed k, the parameter k has to appear in the exponent of n in the running time, that is, the problem is not FPT. This theory has been quite successful for identifying which parameterized problems are FPT and which are unlikely to be. Besides this qualitative classification of FPT versus $W[1]$-hard, more recent developments give us also (an often surprisingly tight) quantitative understanding of the time needed to solve a parameterized problem. Under reasonable assumptions about the hardness of the CNF-SAT problem, it is possible to show that there is no $f(k) \cdot n^c$, or even $f(k) \cdot n^{o(k)}$-time algorithm for finding a clique on k vertices. Thus, up to constant factors in the exponent, the naive $\mathcal{O}(n^k)$-time algorithm is optimal!

Any algorithmic theory is incomplete without an accompanying complexity theory that establishes intractability of certain problems. There is such a complexity theory providing lower bounds on the running time required to solve parameterized problems.

So the common belief is that there is no algorithm for solving CLIQUE with running time $f(k) \cdot n^{o(k)}$. But what if we seek a k-clique in a graph of maximum degree Δ where the parameter is Δ? Notice that the existence of a k-clique implies that $\Delta \geq k - 1$; thus, Δ is a weaker parameter than k, which gives hope for membership in FPT. In fact, it turns out that this can be done quite easily and efficiently when Δ is small: if we guess one vertex v in the clique, then the remaining vertices in the clique must be among the Δ neighbors of v; yhus we can try all of the 2^{Δ} subsets of the neighbors of v, and return the largest clique that we found. The total running time of this algorithm is $\mathcal{O}(2^{\Delta} \cdot \Delta^2 \cdot n)$, which is feasible for $\Delta = 20$ even if n is quite large. Again it is possible to use complexity theoretic assumptions on the hardness of CNF-SAT to show that this algorithm is asymptotically optimal, up to multiplicative constants in the exponent.

What the algorithm above shows is that the CLIQUE problem is FPT when the parameter is the maximum degree Δ of the input graph. At the same time CLIQUE is probably not FPT when the parameter is the solution size k. Thus, the classification of the problem into "tractable" or "intractable" crucially depends on the choice of parameter. This makes a lot of sense; the more we know about our input instances, the more we can exploit algorithmically! For the same problem there can be multiple choices of parameters. *Selecting the right parameter(s) for a particular problem is an art.* We summarize the complexities of the above problems in Fig. 1.1.

1.1.2 Formal Definitions

Definition 1.2 A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ (for alphabet Σ) is called *fixed-parameter tractable* (FPT) if there exists an algorithm $\mathcal{A}$ (called a *fixed-parameter algorithm*), a computable function $f \colon \mathbb{N} \to \mathbb{N}$, and a constant c with the

Problem/Parameter	Good news	Bad news
VERTEX COVER/k	$\mathcal{O}(2^k \cdot k \cdot n)$-time algorithm	NP-hard (probably not in P)
CLIQUE/Δ	$\mathcal{O}(2^\Delta \cdot \Delta^2 \cdot n)$-time algorithm	NP-hard (probably not in P)
CLIQUE /k	$n^{\mathcal{O}(k)}$-time algorithm	$W[1]$-hard (probably not FPT)
VERTEX COLORING/k		NP-hard for $k = 3$ (probably not XP)

Figure 1.1 Overview of the discussed problems.

following property. Given any $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm $\mathcal{A}$ correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |x|^c$ where $|x|$ denotes the length of the input x. The complexity class containing all fixed-parameter tractable problems is called FPT.

1.2 Randomization

Randomness is a powerful resource in designing algorithms and often leads to elegant algorithms, and parameterized algorithms are no exception. Consider for example the classic LONGEST PATH problem. A path is a sequence $v_1, v_2, \ldots, v_\ell$ of *distinct* vertices of a graph, such that for every i there is an edge in the graph from v_i to v_{i+1}. The number ℓ of the vertices in the path is its *length.* In the LONGEST PATH problem the input is a (directed and undirected) graph G together with an integer k, and the goal is to determine whether G contains a path P of length k. For simplicity we shall restrict our attention to undirected graphs, even though most of the discussion applies also to directed graphs.

When $k = n$ the LONGEST PATH problem is precisely the well-known HAMILTONIAN PATH problem, and therefore it is NP-complete. On the other hand, there is a simple $n^{k+\mathcal{O}(1)}$ time algorithm that tries all sequences of k vertices and checks whether any of them forms a k-path. Papadimitriou and Yannakakis [Papadimitriou and Yannakakis (1996)] posed as an open problem whether there exists a polynomial time algorithm that determines whether a graph G on n vertices contains a path of length at least $\log n$. To achieve this it suffices to devise an algorithm with running time $c^k n^{\mathcal{O}(1)}$ for some constant c. In 1995, Alon, Yuster and Zwick [Alon et al. (1995)] invented the *color coding* technique and gave a $(2e)^k n^{\mathcal{O}(1)}$ time randomized algorithm for LONGEST PATH (where $e \approx 2.718$), resolving the question of Papadimitriou and Yannakakis in the affirmative.

The algorithm is based on two key steps, a *random coloring* step, followed by a procedure that finds a *multi-colored* path P, if such a path exists. To describe both steps we need a few definitions: a k-*coloring* of G is a function $c : V(G) \to \{1, \ldots, k\}$. Notice that a coloring simply assigns a number (color) to every vertex, and we do *not* demand that this is a proper coloring in the terminology of graph theory, where edge endpoints need to receive different colors. We will say that a path $P = v_1, v_2, \ldots, v_\ell$ is *multi-colored* by a coloring c if all vertices in P receive distinct colors by c. Formally we require that for every $i \neq j$ we have $c(i) \neq c(j)$.

The first key building block of the algorithm is the insight that, for every path P of length k, a random coloring $c : V(G) \to \{1, \ldots, k\}$ with k colors will multi-color P with "not so small" probability. In particular, there are k^k ways to color $V(P)$ with k colors, and $k!$ of these colorings assign distinct colors to all vertices of P. Hence, the probability that P is multicolored is

$$\frac{k!}{k^k} \geq \frac{\sqrt{2\pi k}}{e^k} \geq e^{-k}. \tag{1.1}$$

Here the first inequality follows from Stirling's approximation for the factorial function.

The second building block is an efficient algorithm that determines whether G contains a path P of length k that is multicolored by a given coloring c. The algorithm uses dynamic programming. We define a function

$$f : 2^{\{1,\ldots,k\}} \times V(G) \to \{\mathsf{true}, \mathsf{false}\}$$

that takes as input a set $S \subseteq \{1, \ldots, k\}$ together with a vertex $v \in V(G)$ and outputs true if there exists a path in $G[S]$ (the subgraph of G induced by S, that is, we retain only the vertices in S and all edges between them) that uses each color of S precisely once and ends at v. Note that such a path necessarily has length $|S|$. It can be easily verified that the function f satisfies the following recurrence relation:

$$f(S, v) = \begin{cases} \mathsf{false} & \text{if } c(v) \notin S \text{ or} \\ \mathsf{true} & \text{if } \{c(v)\} = S \\ \bigvee_{u \in N(v)} f(S \setminus \{c(v)\}, u) & \text{otherwise.} \end{cases} \tag{1.2}$$

The recurrence (1.2) immediately yields an algorithm for determining whether a multi-colored path exists: iterate through every set $S \subseteq \{1, \ldots, k\}$ from small to large and every vertex $v \in V(G)$. In each iteration compute $f(S, v)$ using Equation (1.2) and store the result in a table. Thus, when the algorithm computes $f(S, v)$ it can look up the value of $f(S \setminus \{c(v)\}, u)$ (which was computed in a previous iteration) in the table.

In each of the iterations the value of $f(S, v)$ is computed in at most $d(v)$ table lookups, which we assume take constant time. Thus the total time of the algorithm for determining whether a multi-colored path exists is upper bounded by

$$\mathcal{O}\left(\sum_{S\subseteq\{1,\ldots,k\}}\ \sum_{v\in V(G)} d(v)\right) = \mathcal{O}\left(2^k(n+m)\right).$$

The final algorithm for Longest Path is as follows: iterate through e^k random k-colorings c_i (here i goes from 1 to e^k). For each i check in time $\mathcal{O}\left(2^k(n+m)\right)$ whether there exists a path that is multi-colored by c_i using the dynamic programming algorithm above. If the algorithm finds a multi-colored path, then this path has length k. On the other hand, if a path of length k exists, then the probability that *none* of the c_i's multi-colors it is at most

$$(1-\frac{1}{e^k})^{e^k} \leq \frac{1}{e}.$$

Thus, the algorithm runs in time $\mathcal{O}((2e)^k(n+m))$, always correctly returns "no" on no-instances, and returns "yes" on yes-instances with probability at least $1-\frac{1}{e}$.

Theorem 1.3 (Alon et al. (1995)) *There exists a randomized algorithm with one-sided error for* Longest Path *with running time $\mathcal{O}((2e)^k(n+m))$.*

1.2.1 Random Separation: Set Splitting

Color coding is far from being the only way to use randomness to design parameterized algorithms. A different example is the "random separation" techique. We will see how to apply random separation to design an algorithm for the Set Splitting problem. Here the input is a universe U, a family $\mathcal{F} = \{S_1, S_2, \ldots, S_m\}$ of subsets of U, and an integer $k \leq m$. The goal is to find an assignment $\phi : U \to \{0,1\}$ that *splits* at least k sets in $\mathcal{F}$. Here a set $S_i \in \mathcal{F}$ is *split* by ϕ if it contains at least one element u such that $\phi(u) = 0$ and at least one element v such that $\phi(v) = 1$. The Set Splitting problem is also known as Hypergraph Max Cut, because when all sets S_i have cardinality 2 this is preciesly the classic Max Cut problem.

It turns out that a remarkably simple strategy yields an FPT algorithm for Set Splitting. In particular, we shall prove that if there exists an assignment ϕ that splits at least k sets in $\mathcal{F}$, then a *random* assignment ψ splits at least k sets with probability at least $\frac{1}{4^k}$. Indeed, suppose the sets $\{S_1, S_2, \ldots, S_k\}$ are split by ϕ. For every $i \leq k$ let u_i be an element of S_i such that $\phi(u_i) = 0$ and $v_i \in S_i$ be such that $\phi(v_i) = 1$. We remark that u_i and u_j (or v_i and v_j) can be the same element even though $i \neq j$. Let $X = \bigcup_{i\leq k}\{u_i, v_i\}$, and observe that X has the following two properties. First, $|X| \leq 2k$. Second, for every assignment ψ that agrees with ϕ on X (that is, $\psi(x) = \phi(x)$ for every $x \in X$) ψ splits the sets $S_1, \ldots, S_k$ (since $\psi(u_i) = 0$ and $\psi(v_i) = 1$). The probability that ψ agrees with ϕ on X is $2^{-|X|} \geq 2^{-2k} = 4^{-k}$, as claimed.

This gives the following simple algorithm that runs in time $\mathcal{O}(4^k nm)$: try 4^k

random assignments $\psi_1, \ldots, \psi_{4^k}$. If some ψ_i splits at least k sets, return that assignment. If none of the ψ_i's split at least k sets report that no assignment does. Just as for the Longest Path problem, if there is an assignment that splits at least k sets, then the probability that the algorithm fails to find one is at most $(1 - \frac{1}{4^k})^{4^k} \leq 1/e$. This proves the following theorem.

Theorem 1.4 *There exists a randomized algorithm with one-sided error for* Set Splitting *with running time $\mathcal{O}(4^k nm)$.*

We remark that a random assignment ψ will actually split k sets with probability at least $\frac{1}{2^k}$. The proof of this claim is (slightly) more complicated, so we leave it as Exercise 1.4 (see also [Chen and Lu (2009)]).

1.2.2 Derandomization

It would appear that the algorithms of Theorems 1.3 and 1.4 are inherently randomized. It turns out that the randomized step can be replaced by appropriate pseudo-random constructions without compromising (much) on the worst case running time guarantees.

Let us first consider the algorithm for Longest Path. Here we tried e^k random colorings and used the fact that if there exists a path, then with probability at least $1 - 1/e$ at least one of the colorings multi-colors it. Quite remarkably one can deterministically construct a family of colorings such that this property *always* holds, instead of holding with constant probability.

Theorem 1.5 (Naor et al. (1995)) *There exists an algorithm that given a universe U of size n and an integer k runs in time $e^k n^{\mathcal{O}(1)}$ and produces k-colorings $c_1, c_2, \ldots c_\ell$ with $\ell = \mathcal{O}(e^{k+o(k)} \log n)$ such that for every set $S \subseteq U$ of size at most k there exists an i such that c_i multi-colors S.*

Replacing the e^k random k-colorings with the $e^{k+o(k)} n^{\mathcal{O}(1)}$ k-colorings of Theorem 1.5 in the algorithm for Longest Path yields a *deterministic* algorithm with running time $(2e)^{k+o(k)} n^{\mathcal{O}(1)}$.

A similar situation happens for Set Splitting. What we need is a family of assignments $\psi_1, \psi_2, \ldots \psi_\ell$ such that for some unknown set X of size at most $2k$, at least one of the assignments ψ_i agrees with an unknown assignment ϕ on X. Since X and ϕ are un-known, what we really need is that for *every* set X of size at most $2k$ and *every* assignment $\phi_X : X \to \{0, 1\}$ at least one ψ_i agrees with ϕ_X on X. Again, this can be achieved!

Theorem 1.6 (Naor et al. (1995)) *There exists an algorithm that given a universe U of size n and an integer k runs in time $2^{k+o(k)} n^{\mathcal{O}(1)}$ and produces assignments $\psi_1, \psi_2, \ldots \psi_\ell$ with $\ell \leq 2^{k+o(k)} \log n$ such that for every set $X \subseteq U$ of size at most k and every assignment $\phi_X : X \to \{0, 1\}$ at least one ψ_i agrees with ϕ_X.*

Replacing the 4^k random assignments with the $4^{k+o(k)}n^{\mathcal{O}(1)}$ assigments produced by Theorem 1.6 (applied with $|X| \le 2k$) yields a deterministic algorithm for Set Splitting with running time $4^{k+o(k)}n^{\mathcal{O}(1)}m$. We remark that the constructions of Theorems 1.5 and 1.6 are optimal in the sense that ℓ can not be reduced below $\Omega(e^k \log n)$ and $\Omega(2^k \log n)$ respectively.

Theorem 1.7 *There exists a deterministic algorithm for* Longest Path *with running time* $(2e)^{k+o(k)}n^{\mathcal{O}(1)}$. *There exists a deterministic algorithm for* Set Splitting *with running time* $4^{k+o(k)}n^{\mathcal{O}(1)}m$.

1.3 Structural Parameterizations

We have mostly concerned ourselves with the parameter k being the value of the objective function, or just the size of the solution. This does *not* mean that this is the only reasonable choice of parameter! For example, recall that the Graph Coloring problem is NP-complete for $k = 3$ where k is the number of colors, so an FPT algorithm parameterized by k is out of the question. This does not rule out the potential for other interesting parameterized algorithms for the problem. Suppose we want to solve Graph Coloring, but now we know that the input instances have a relatively small vertex cover, say of size at most t. Can we use this to get an efficient algorithm?

Here is an algorithm that is FPT when parameterized by the vertex cover number t. First, compute a vertex cover X of size at most t in time $2^t(n+m)$ using the algorithm of Theorem 1.1. If $k > t$ we can find a coloring with at most $t+1 \le k$ colors in time $\mathcal{O}(m+n)$: just use one color of $\{1,\dots,|X|\}$ per vertex of X and use color $|X|+1$ for all of the remaining vertices. Because X is a vertex cover, this is a proper coloring.

Suppose now that $k \le t$. The algorithm tries all k^t possible colorings of X. For each such choice $c_X : X \to \{1,\dots,k\}$ it checks whether c_X can be extended to a proper coloring of G. A *necessary* condition for this to be possible is that every vertex $y \notin X$ has an available color. Formally, for every $y \notin X$ there should exist an $1 \le i \le k$ so that no neighbor of y has color i. Since no pair of vertices outside of X are adjacent this necessary condition is also *sufficient*—to each $y \notin X$ we can simply assign any of its available colors. This leads to an algorithm for Graph Coloring with running time $\mathcal{O}(k^t(n+m)) \le \mathcal{O}(t^t(n+m))$.

Theorem 1.8 *There exists an algorithm for* Graph Coloring *with running time* $\mathcal{O}(t^t(n+m))$ *where t is the size of the smallest vertex cover of G.*

This algorithm appears quite naive, however, quite surprisingly, one can show that it can not be substantially improved (under appropriate complexity-theoretic assumptions). Indeed, an algorithm with running time $2^{o(t\log t)}n^{\mathcal{O}(1)}$ would contradict the Exponential Time Hypothesis [Lokshtanov et al. (2018)] (see Section 1.5.2).

1.4 Kernelization

Preprocessing is a widely used technique to help cope with computationally hard problems. A natural question in this regard is how to measure the quality of preprocessing rules proposed for a specific problem, yet for a long time the mathematical analysis of polynomial time preprocessing algorithms was neglected. One central reason for this anomaly can be found in the following observation: showing that in polynomial time an instance I of an NP-hard problem can be replaced by an equivalent instance whose size is smaller than the size of I implies that P=NP. The situation has changed drastically with the advent of Parameterized Complexity. Roughly speaking, the objective of preprocessing rules within this framework is to reduce the input size to depend only on the parameter, where the smaller the dependency is, the better the rules are.

1.4.1 Warm-Up: the Buss Rule

Before we delve into the formal definitions, let us see a simple example. For an instance (G, k) of the Vertex Cover problem, consider the following rule.

Rule I. If G contains an isolated vertex v, then remove v from G. The resulting instance is $(G - v, k)$.

This rule takes an instance of the problem, and, if its condition is satisfied, returns an instance of the same problem of smaller size. Such a rule is called a *reduction rule*. Most importantly, this rule is *safe* in the following sense: The instance that it takes as input is a yes-instance if and only if the instance that it outputs is a yes-instance. Indeed, it is immediate to see that the removal of isolated vertices has no effect on the answer of the given instance.

Now, consider yet another reduction rule, known as the *Buss rule*.

Rule II. If G contains a vertex v of degree at least $k+1$, then remove v (along with incident edges) from G, and decrement k by 1. The resulting instance is $(G-v, k-1)$.

The safeness of this rule follows from the observation that any vertex cover of size at most k in G must contain v—indeed, any vertex cover in G that excludes v must contain all of its neighbors, and their number is strictly larger than k.

Lastly, suppose that Rules I and II have been applied exhaustively, thus neither of their conditions is satisfied, and consider the following rule.

Rule III. If G contains more than k^2 edges, then return no.

Here, safeness also follows from a simple observation: As the maximum degree of a vertex in G is k (due to the exhaustive application of Rule II), any set of at most k vertices can cover at most k^2 edges. Thus, if G contains more than k^2 edges, it does not admit a vertex cover of size at most k. To strictly comply with the definition of a reduction rule, the output should be an instance of Vertex Cover rather than yes or no. However, it is acceptable to use yes or no as abbreviations for trivial yes- or no-instances, respectively. For Vertex Cover, concrete trivial yes- and no-instances can be, for example, the graph on an empty vertex set with $k = 0$, and the graph on two vertices connected by an edge with $k = 0$, respectively.

After the consideration of the last rule, we know that the number of edges in G is at most k^2. Furthermore, the number of vertices in G is at most $2k^2$ because G does not contain any isolated vertex (due to the exhaustive application of Rule I). We also observe that the entire process is implementable in polynomial time—each of our three rules can be applied only polynomially many times and each application runs in polynomial time. Thus, in polynomial time, we managed to reduce the size of the input instance (without even trying to solve it!) to be quadratic in k.

1.4.2 Formal Definition and Relation to Membership in FPT

The main definition in Kernelization is that of a kernel, which is derived from a more general notion called *compression.*

Definition 1.9 A *compression* of a parameterized language $Q \subseteq \Sigma^* \times \mathbb{N}$ into a language $R \subseteq \Sigma^*$ is an algorithm that takes as input an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, runs in time polynomial in $|x| + k$, and returns a string y such that:

1 $|y| \leq f(k)$ for some function $f(\cdot)$, and
2 $y \in R$ if and only if $(x, k) \in Q$.

If $|\Sigma| = 2$, the function $f(\cdot)$ is called the *bitsize* of the compression.

Kernelization is the special case of compression where the projection of Q onto Σ^* equals R, that is, the source and target languages are essentially the same. Then, the algorithm is referred to as a *kernelization algorithm* or a *kernel.* Particular attention is given to the case where the function f is polynomial. In this case, we say that the problem admits *polynomial compression or kernelization.* While a polynomial kernel is obviously better than an arbitrary kernel, we have particular interest in polynomial kernels due the following characterization, which uses arbitrary (not necessarily polynomial) kernels.

In one direction, it is easy to see that if a decidable (parameterized) problem admits a kernel for some function f, then it is FPT: for any instance of the problem, we call a (polynomial-time) kernelization algorithm, and then use a decision algorithm to determine the answer to the resulting instance. Since the size of the kernel is bounded by some function f of the parameter, the running time of the decision algorithm depends only on the parameter. More surprising is the converse direction:

Theorem 1.10 *If a parameterized problem L is FPT, then it admits a kernel.*

Proof Suppose that there is an algorithm deciding if $(x, k) \in L$ in time $f(k)|x|^c$ for some computable function f and constant c. We consider two cases. In the first, $|x| \geq f(k)$, and we run the FPT algorithm on the instance in time $f(k)|x|^c \leq |x|^{c+1}$. If the FPT algorithm outputs yes, then the kernelization algorithm outputs a constant size yes-instance, and if the decision algorithm outputs no, then the kernelization algorithm outputs a constant size no-instance. In the second case, $|x| < f(k)$, and the kernelization algorithm outputs x. This yields a kernel of size $f(k)$ for the problem. □

Theorem 1.10 shows that kernelization gives rise to an alternative definition of membership in FPT. So, to decide if a parameterized problem has a kernel, we can employ many known tools already given by Parameterized Complexity. But what if we are interested in kernels that are as small as possible? The size of a kernel obtained using Theorem 1.10 equals the dependence on k in the running time of the best known parameterized algorithm for the problem, which is often quite large (exponential or worse). Can we find better kernels? The answer is yes, we can, but not always. For many problems we can obtain polynomial kernels, but under reasonable complexity-theoretic assumptions, there exist problems in FPT that do not admit kernels of polynomial size (see Section 1.5).

In Section 1.4.1, we have already seen a polynomial (in fact, quadratic) kernel for Vertex Cover, based on three very simple rules, where the central one is known as the Buss rule. Specifically, we proved the following theorem.

Theorem 1.11 Vertex Cover *admits a kernel of size $\mathcal{O}(k^2)$.*

In fact, a general scheme to develop kernelization algorithms is to provide a list of reduction rules, where always the first rule in the list whose condition is satisfied is the one to be executed next; eventually, when no rule is applicable, the size of the instance should be bounded. Next, we adapt the Buss rule to a less expected context. We remark that nowadays, there is a rich tool-kit to design kernels (see Notes). Moreover, extensions of kernelization are explored (see Section 1.6.1).

1.4.3 Generalization of the Buss Rule to Matrix Rank

We will now discuss a more sophisticated example of a polynomial kernel. The *rigidity* of a matrix A for a target rank r over a field $\mathbb{F}$ is the minimum Hamming distance between A and a matrix of rank at most r. Naturally, given a parameter k, the Matrix Rigidity problem asks whether the rigidity of A is at most k.

Theorem 1.12 *There exists a polynomial-time algorithm that, given an instance (A, k, r) of* Matrix Rigidity*, returns an equivalent instance (A', k, r) where A' has $\mathcal{O}((rk)^2)$ entries.*

Proof sketch. Given an instance (A, r, k) of Matrix Rigidity, the algorithm works as follows. For $k + 1$ steps, it repeatedly selects a set of maximum size consisting of rows that are linearly independent, where if the size of this set exceeds $r + 1$, a subset of it of size $r + 1$ is used instead. Each such set of rows is removed from the input matrix, and then it is inserted into the output matrix. At the end of this greedy process, rows that remain in the input matrix are simply discarded. Afterwards, the symmetric process is executed on columns.

Clearly, the output matrix A' has at most $(r + 1)k$ rows and $(r + 1)k$ columns, and therefore $\mathcal{O}((rk)^2)$ entries. Moreover, in the forward direction, because A' is obtained by deleting rows, and then further deleting columns, of A, it is clear that if (A, k, r) is a yes-instance, then so is (A', k, r).

For the reverse direction, we only prove that the operation on rows is safe, as the safeness for columns follows similarly. To this end, let $\widehat{A}$ be the matrix obtained after the operation on rows, and suppose that $(\widehat{A}, k, r)$ is a yes-instance. Let $\widehat{B}$ be a matrix of rank at most r and Hamming distance at most k from $\widehat{A}$. Because the rows of $\widehat{A}$ were taken from A, the matrix B that consists of $\widehat{B}$ and the rows of A outside $\widehat{A}$ is at Hamming distance at most k from A. To prove that (A, k, r) is a yes-instance as well (which will be witnessed by B), we will prove that each row in A that was not inserted into $\widehat{A}$ belongs to the span of the rows of $\widehat{A}$ that also belong to $\widehat{B}$.

Notice that in each set of $r+1$ rows inserted (in the same iteration) into $\widehat{A}$, there must be a difference between $\widehat{A}$ and $\widehat{B}$ as this set in itself is linearly independent. Since $k + 1$ iterations are performed, and the Hamming distance between $\widehat{A}$ and $\widehat{B}$ is k, there must be at least one set that is the same in both, and thus this set must have size at most r. In particular, when this set was inserted into $\widehat{A}$, it was a set of maximum size of linearly independent rows (that was not replaced by a smaller set). Thus, every row in A that was not inserted into $\widehat{A}$ belongs to the span of this set. Because this set is the same in $\widehat{A}$ and $\widehat{B}$, we conclude that B and $\widehat{B}$ have the same rank, and thus (A, k, r) is a yes-instance. □

Let us point out two similarities. While in the Buss rule, we relied on the observation that in a set of $k + 1$ edges incident to the same vertex, that vertex must be picked, here we relied on the observation that in a set of $r + 1$ linearly independent rows, at least one change must be made. Further, both when arguing the correctness of Rule III, and when arguing for the existence of an "untouched" set in the reverse direction, we used an argument based on the pigeonhole principle.

When $\mathbb{F} = \mathbb{R}$, Theorem 1.12 does not yield a kernel for Matrix Rigidity parameterized by $k + r$ because the bitsize to encode each entry may be unbounded in $k + r$. However, for finite fields, we have the following consequence.

Corollary 1.13 Matrix Rigidity *over finite fields admits a kernel of size* $\mathcal{O}((kr)^2 f)$ *where* f *is the field size.*

We remark that parameterization by multiple parameters (among k, r and f) is likely to be essential here, because it can be shown that parameterized by k alone, r alone, or f alone, Matrix Rigidity is W[1]-hard (see Section 1.5.1).

1.5 Hardness and Optimality

1.5.1 W[1]-Hardness

In addition to a rich tool-kit to design parameterized algorithms, research in Parameterized Complexity has also provided complementary methods to show that a problem is unlikely to be FPT. The main technique is the one of parameterized reductions analogous to those employed in the theory of NP-hardness. Here, the concept of *W[1]-hardness* replaces the one of NP-hardness, and for reductions we need not only construct an equivalent instance in FPT time, but also ensure that the size of the parameter in the new instance depends only on the size of the parameter in the original one. If there exists such a reduction transforming a problem known to be W[1]-hard to another problem Π, then the problem Π is W[1]-hard as well. Central W[1]-hard problems include, for example, deciding whether a nondeterministic single-tape Turing machine accepts within k steps, Clique (determine whether a given graph has a clique of size k) parameterized by solution size k, and Independent Set parameterized by solution size k (determine whether a given graph has an independent set of size k). To show that a problem Π is not XP unless P=NP, it is sufficient to show that there exists a fixed k such Π is NP-hard. Then, the problem is said to be *para-NP-hard.*

More formally, a central notion in this context is of a parameterized reduction, defined as follows.

Definition 1.14 Let $A, B \subseteq \Sigma^* \times \mathbb{N}$ be two parameterized problems. A *parameterized reduction* from A to B is an algorithm that, given an instance (x, k) of A, outputs an instance (x', k') of B such that: *(i)* (x, k) is a yes-instance of A if and only if (x', k') is a yes-instance of B; *(ii)* $k' \leq g(k)$ for some computable function g; *(iii)* the running time is upper bounded by $f(k) \cdot |x|^c$ for some computable function f and constant c.

As an example of a very simple parameterized reduction, let A be Clique and B be Independent Set, and consider the following algorithm. Given an instance

(G,k) of CLIQUE, the algorithm outputs the instance $(\overline{G},k)$ of INDEPENDENT SET, where $\overline{G}$ is the complement of G (i.e. $\{u,v\} \in E(G)$ if and only if $\{u,v\} \notin E(\overline{G})$). Then, it is trivial to verify that the three properties in Definition 1.14 are satisfied. Generally, the design of parameterized reductions can often be quite technical as we need to avoid blowing up the parameter. In many cases, it is useful to consider the "colorful version" of a W[1]-hard problem as the source of the reduction. In particular, in COLORFUL CLIQUE we are given a graph G, and a (not necessarily proper) coloring of the vertices of G in k colors, and the goal is to determine whether G has a clique on k vertices where each vertex has a distinct color. Roughly speaking, the main reason why colors help is that they enable the reduction to consist, for each color, of a gadget for the selection of a vertex of that color, rather than k gadgets for the selection of k vertices from the entire graph. In particular, each gadget "works" on a different set of vertices. This generally simplifies the design of the gadgets that need to verify that the selected vertices form a clique.

We remark that for some choices of A and B, even when both problems are NP-hard, we do not expect to have a parameterized reduction. For example, let A be CLIQUE and B be VERTEX COVER. As already noted in Section 1.1.1, VERTEX COVER is FPT while CLIQUE is W[1]-hard. Then, although there exists a polynomial-time reduction from CLIQUE to VERTEX COVER (because both problems are NP-complete), a parameterized one would imply that CLIQUE is FPT, which is considered unlikely. We remark that known reductions from CLIQUE to VERTEX COVER blow-up the parameter in the output instance, so it no longer depends only on the parameter in the input instance but on the entire input instance size. Interestingly, while we know of a parameterized reduction from INDEPENDENT SET to DOMINATING SET (through COLORFUL INDEPENDENT SET, see Notes), we do not know of a parameterized reduction from DOMINATING SET to INDEPENDENT SET. In fact, we do not expect that such a reduction exists, since INDEPENDENT SET and DOMINATING SET lie in different levels of the *W-hierarchy*. Specifically, INDEPENDENT SET is complete for the first level of this squirearchy, while DOMINATING SET is complete for the second one. For more details, we refer to the Notes.

1.5.2 ETH and SETH

To obtain (essentially) tight conditional lower bounds for the running times of algorithms, we can rely (among other hypotheses) on the *Exponential-Time Hypothesis (ETH)* and *Strong Exponential-Time Hypothesis (SETH)*. To formalize the statements of ETH and SETH, first recall that given a formula φ in conjuctive normal form (CNF) with n Boolean variables and m clauses, the task of CNF-SAT is to decide whether there is a truth assignment to the variables that satisfies φ. In the p-CNF-SAT problem, each clause is restricted to have at most p literals. First, ETH asserts that 3-CNF-SAT cannot be solved in time $\mathcal{O}(2^{o(n)})$. Second, SETH asserts that for every fixed $\epsilon < 1$, there exists a (large) integer $p = p(\epsilon)$ such that

p-CNF-SAT cannot be solved in time $\mathcal{O}((2-\epsilon)^n)$. We remark that for every fixed integer p, we know that p-CNF-SAT is solvable in time $\mathcal{O}(c^n)$ for $c < 2$ that depends on p (notice that this does not contradict SETH).

Parameterized reductions (as in Definition 1.14) can be used in conjunction with ETH or SETH to provide more fine-grained lower bounds. For an example, consider the classic reduction from 3-CNF-SAT to VERTEX COVER described by e.g. Sipser (1996). This reduction has the following properties: Given a 3-CNF-SAT instance ϕ with n variables and m clauses, the reduction outputs in polynomial time a graph G on $3m$ vertices, such that G has a vertex cover of size at most m if and only if ϕ is satisfiable.

This reduction, together with the ETH, rules out the possibility of a $2^{o(|V(G)|^{1/3})}$ time algorithm for VERTEX COVER: if such an algorithm were to exist, we could feed the output of the reduction into the algorithm and solve 3-CNF-SAT in time

$$2^{o(|V(G)|^{1/3})} \leq 2^{o((3m)^{1/3})} \leq 2^{o(n)}.$$

This would contradict the ETH. In the last transition we used that the number of clauses m in a 3-SAT instance is at most $\mathcal{O}(n^3)$.

Some key points to take away from this reduction. First, good old NP-hardness reductions are by themselves sufficient to provide running time lower bounds assuming the ETH, we just had to carefully keep track how the parameters of the instance produced by the reduction depend on the parameters of the input instance. Second, the running time lower bound of $2^{o(|V(G)|^{1/3})}$ is very far off from the $2^{\Theta(n)}$ running times of the currently best known algorithms for VERTEX COVER. Luckily, Impagliazzo et al. (2001) provide a handy tool to bridge this gap.

Theorem 1.15 *Assuming the ETH,* 3-CNF-SAT *has no algorithm with running time* $2^{o(n+m)}$

Theorem 1.15, combined with the reduction above shows that VERTEX COVER can not have an algorithm with running time $2^{o(n)}$, since this would yield an algorithm for 3-CNF-SAT with running time $2^{o(m)}$, contradicting ETH. Thus, assuming the ETH, *up to constants in the exponent* the existing algorithms for VERTEX COVER (in fact even the naive 2^n time algorithm) are the best one can do!

Notice also that this gives a lower bound on the running time of FPT algorithms for VERTEX COVER when parameterized by solution size k. Indeed, since $k \leq |V(G)|$, a $2^{o(k)}|V(G)|^{\mathcal{O}(1)}$ time algorithm for VERTEX COVER would contradict the ETH. From this starting point one can get numerous lower bounds that miraculously match the running times of the best known algorithms, just by tracing the existing reductions. Of course in many cases clever new reductions have to be designed to get tight bounds (see the survey Lokshtanov et al. (2011)). For example, assuming the ETH, all of the following algorithms are already best possible, up to constants in the exponent: The naive $n^{k+\mathcal{O}(1)}$ time algorithms for CLIQUE and DOMINATING SET, the $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$ time algorithms for LONGEST PATH and SET

Splitting from Theorems 1.3 and 1.4, and the $\mathcal{O}(t^t(n+m))$ time algorithm for Graph Coloring from Theorem 1.8.

All of the so-called 'tight' lower bounds assuming the ETH come with the fine print "*up to constants in the exponent*". This is of course unsatisfactory - there is a huge difference between a 2^n time algorithm and a 1.00001^n time algorithm and, so far, no one has been able to use the ETH to rule out an algorithm with running time 1.00001^n for a problem that does have an algorithm with running time 100^n. If one is willing to assume the much stronger hypothesis that is SETH, then one can pin down the precise dependence on k for some problems [Lokshtanov et al. (2011)]. However, so far we are lacking a result akin to Theorem 1.15 for SETH, quite severly limiting its applicability.

1.5.3 Hardness and Optimality of Kernelization

We showed that Longest Path can be solved in time $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$, where n is the number of vertices in the input graph G (Theorem 1.7). Thus by Theorem 1.10, we deduce that Longest Path admits a kernel of size $2^{\mathcal{O}(k)}$. But what about a kernel of polynomial size?

We argue that intuitively this should not be possible. Assume that Longest Path admits a polynomial kernel of size k^c, where c is some fixed constant. We take many instances,

$$(G_1, k), (G_2, k), \ldots, (G_t, k),$$

of the Longest Path problem, where in each instance $|V(G_i)| = n$, $1 \le i \le t$, and $k \le n$. If we make a new graph G by just taking the disjoint union of the graphs $G_1, \ldots, G_t$, we see that G contains a path of length k if and only if G_i contains a path of length k for some $i \le t$. Now run the kernelization algorithm on G. Then kernelization algorithm would in polynomial time return a new instance (G', k') such that $|V(G')| \le k^c \le n^c$, a number potentially much smaller than t, for example set $t = n^{1000c}$. This means that in some sense, the kernelization algorithm considers the instances $(G_1, k), (G_2, k), \ldots, (G_t, k)$ and in *polynomial time* figures out which of the instances are the most likely to contain a path of length k. More precisely, if we have to preserve the value of the OR of our instances while being forced to forget at least one of the inputs entirely, then we have to make sure that the input being forgotten was not the only one whose answer is yes (otherwise we turn a yes-instanceinto a no-instance). However, at least intuitively, this seems almost as difficult as solving the instances themselves, and since the Longest Path problem is NP-complete, this seems unlikely. In 2009, a methodology to rule out polynomial kernels has been developed in [Bodlaender et al. (2009); Fortnow and Santhanam (2008)]. The existence of polynomial kernels are ruled out, in this framework, by linking the availability of a polynomial kernel to an unlikely collapse in classical complexity. These developments deepen the connection between classical and parameterized complexity. Using this methodology one can show that LONGEST PATH does not admit a polynomial kernel unless coNP $\subseteq \frac{\text{NP}}{\text{poly}}$. In fact, in Dell and van Melkebeek (2010), the kernel lower bound methodology established in [Bodlaender et al. (2009); Fortnow and Santhanam (2008)] was generalized further to provide lower bounds based on different polynomial functions for the kernel size. For an example we can show that the $\mathcal{O}(k^2)$ kernel for VERTEX COVER given by 1.11 is optimal. That is, VERTEX COVER does not have kernels of size $\mathcal{O}(k^{2-\varepsilon})$ unless coNP $\subseteq \frac{\text{NP}}{\text{poly}}$. We refer to Notes for more details.

1.6 Outlook: New Paradigms and Application Domains

The main idea of parameterized algorithms is very general: to measure the running time in terms of both input size as well as a parameter that captures structural properties of the input instance. This idea of a multivariate algorithm analysis holds the potential to address the need for a framework for refined algorithm analysis for all kinds of problems across all domains and subfields of computer science. Indeed, parameterized complexity has permeated other algorithmic paradigms as well as to other application domains. In this section we look at some of these.

1.6.1 FPT-approximation and lossy kernels

Until now we have only seen decision problems in the realm of fixed parameter tractability. However, to define the notion of FPT-approximation we need to move away from decision problems and define the notion of optimization problems. However, this chapter is not the right place to do so. We will work with some ad-hoc definitions and use them to show glimpses of the budding field of FPT-approximation. For the purpose of this section, we will call an approximaiton algorithm, an FPT-approximation algorithm, if its running time is $f(k) \cdot n^{\mathcal{O}(1)}$, for a parameter k (k need not be the solution size).

We will exemplify this paradigm via the PARTIAL VERTEX COVER problem. In this problem, we are given an undirected graph G and a positive integer k, and the task is to find a subset X of k vertices that covers as many edges as possible. This problem can easily be shown to be W[1]-hard via a reduction from the INDEPENDENT SET problem. In what follows, we give an algorithm that for every $\epsilon > 0$, runs in time $f(\epsilon, k) \cdot n^{\mathcal{O}(1)}$, and produces a $(1+\epsilon)$-approximate solution for the problem.

A natural greedy heuristic for the problem is to *output a set X containing k vertices of the highest degree*. The case in which X could cover far fewer edges than an optimal solution is when the number of edges with both end-points in X is proportional to the total number of edges that X covers. We will show that this case can be addressed with an FPT algorithm. Let us fix $C = 2\binom{k}{2}/\epsilon$ and and let

$v_1, \ldots, v_n$ be the vertices of the graph ordered by non-increasing degree. In the first case we assume that $d(v_1) \geq C$. Here, $d(v_i)$ denotes the degree of v_i. In this case the greedy heuristic outputs our desired solution. Indeed, $X = \{v_1, \ldots, v_k\}$ covers at least $\sum_{i=1}^{k} d(v_i) - \binom{k}{2}$ edges. The last inequality follows from the fact that any simple graph on k vertices has at most $\binom{k}{2}$ edges. Observe that optimal the solution is always upper bounded by $\sum_{i=1}^{k} d(v_i)$. Thus, the quality of the desired solution is

$$\frac{\sum_{i=1}^{k} d(v_i) - \binom{k}{2}}{\sum_{i=1}^{k} d(v_i)} \geq 1 - \frac{\binom{k}{2}}{C} \geq 1 - \frac{\epsilon}{2} \geq \frac{1}{1+\epsilon},$$

at most $(1+\epsilon)$ times the optimum.

We can therefore assume that $d(v_1) < C = 2\binom{k}{2}/\epsilon$. Hence, the optimal solution is upper bounded by $\sum_{i=1}^{k} d(v_i) < Ck$. That is, in this case the maximum number of edges, say t, that any set of size k can cover becomes a function of k and ϵ. What if we parameterize Partial Vertex Cover by the maximum number of edges, say t, that are covered by a set of size k? Indeed, one can show that Partial Vertex Cover is solvable in time $2^{\mathcal{O}(t)} n^{\mathcal{O}(1)}$ using the method of color coding explained in Section 1.2. We leave this as an exercise (Exercise 1.5). With this exercise in hand we can prove the following theorem.

Theorem 1.16 *For every $\epsilon > 0$, there exists an algorithm for* Partial Vertex Cover *that runs in time $f(\epsilon, k) \cdot n^{\mathcal{O}(1)}$, and produces a $(1+\epsilon)$-approximate solution for the problem.*

Partial Vertex Cover is not the only problem for which FPT-approximation algorithm exists. There are several other problem but arguably the most notable one is s-Way Cut, parameterized by s (delete minimum number of edges such that the resulting graph has at least s connected components). The problem admits a factor 2-approximation algorithm in polynomial time. On the other hand assuming some well known complexity theory assumption it is not possible to improve this approximation algorithm. Furthermore, the problem is known to be W[1]-hard, parameterized by s [Downey et al. (2003)]. Gupta et al. (2018) obtained $2-\epsilon$-factor approximation algorithm, a fixed constant $\epsilon > 0$, running in time $f(s) \cdot n^{\mathcal{O}(1)}$. After couple of improvements, Kawarabayashi and Lin (2020) have recently obtained a nearly 5/3 factor FPT-approximation algorithm. On the other hand the area of FPT-inapproximability has also flourished. In particular, assuming FPT$\neq W[1]$, Dominating Set does not admit an FPT-approximation algorithm with factor $o(k)$ [Karthik C. S. et al. (2019)]. On the other hand, assuming a gap version of ETH, one can show that Clique does not admit any FPT-approximation algorithm [Chalermsook et al. (2017)].

Can we combine the theory of kernelization (Section 1.4) with FPT approximation? Unfortunately, the answer is no. Despite the success of kernelization, the basic definition has an important drawback: *it does not combine well with approximation* *algorithms.* This is a serious problem since after all the ultimate goal of parameterized algorithms, or for that matter of any algorithmic paradigm, is to eventually solve the given input instance. Thus, the application of a pre-processing algorithm is always followed by an algorithm that finds a solution to the reduced instance. In practice, even after applying a pre-processing procedure, the reduced instance may not be small enough to be solved to optimality within a reasonable time bound. In these cases one gives up on optimality and resorts to approximation algorithms (or heuristics) instead. Thus it is *crucial* that the solution obtained by an approximation algorithm when run on the reduced instance provides a good solution to the original instance, or at least *some* meaningful information about the original instance.

The main reason that the existing notion of kernelization does not combine well with approximation algorithms is that the definition of a kernel is deeply rooted in decision problems, while approximation algorithms are optimization problems. This led to the new definition of *lossy kernels*, coined by Lokshtanov et al. (2017) which extends the notion of kernelization to optimization problems. The main object here is a definition of *α-approximate kernels.* We do not give the formal definition here, but, loosely speaking, an α-approximate kernel of size $g(k)$ is a polynomial time algorithm that given an instance (I, k) outputs an instance (I', k') such that $|I'| + k' \leq g(k)$ and any c-approximate solution s' to the instance (I', k') can be turned in polynomial time into a $(c \cdot \alpha)$-approximate solution s to the original instance (I, k).

We again exemplify the idea of α-approximate kernels by giving a suitable algorithm for Partial Vertex Cover. Here, we will rely on the observation that the first case in the proof of Theorem 1.16) is handled in polynomial time. We formally show this in the next theorem.

Theorem 1.17 Partial Vertex Cover *admits α-approximate kernels for every $\alpha > 1$.*

Proof We give an α-approximate kernelization algorithm for the problem for every $\alpha > 1$. Let $\epsilon = 1 - \frac{1}{\alpha}$ and $\beta = \frac{1}{\epsilon}$. Let (G, k) be the input instance. Let $v_1, v_2, \ldots, v_n$ be the vertices of G in the non-increasing order of degree, i.e $d_G(v_i) \geq d_G(v_j)$ for all $1 \geq i > j \geq n$. The kernelization algorithm has two cases based on the degree of v_1.

Case 1: $d_G(v_1) \geq \beta\binom{k}{2}$. In this case $S = \{v_1, \ldots, v_k\}$ is a α-approximate solution. The number of edges incident to S is at least $(\sum_{i=1}^{k} d_G(v_i)) - \binom{k}{2}$, because at most $\binom{k}{2}$ edges have both end points in S and they are counted twice in the sum $(\sum_{i=1}^{k} d_G(v_i))$. The value of the optimum solution is at most $\sum_{i=1}^{k} d_G(v_i)$. As in Theorem 1.16, we can show that

$$\frac{(\sum_{i=1}^{k} d_G(v_i)) - \binom{k}{2}}{\sum_{i=1}^{k} d_G(v_i)} \geq 1 - \frac{\binom{k}{2}}{d_G(v_1)} \geq 1 - \frac{1}{\beta} = \frac{1}{\alpha}$$

The above inequality implies that S is an α-approximate solution. So the kernelization algorithm outputs a trivial instance $(\emptyset, 0)$ in this case.
Case 2: $d_G(v_1) < \beta\binom{k}{2}$**.** Let $V' = \{v_1, v_2, \ldots, v_{k\lceil\beta\binom{k}{2}\rceil+1}\}$. In this case the algorithm outputs (G', k), where $G' = G[N_G[V']]$ (the subgraph of G induced by the vertices of V' and all of their neighbors). Let $OPT(G,k)$ denote the optimum value of the instance. We first clam that $OPT(G',k) = OPT(G,k)$. Since G' is a subgraph of G, $OPT(G',k) \leq OPT(G,k)$. Now it is enough to show that $OPT(G',k) \geq OPT(G,k)$. Towards that, we prove that there is an optimum solution that contains only vertices from the set V'. Suppose not, then consider the solution S which is lexicographically smallest in the ordered list $v_1, \ldots v_n$. The set S contains at most $k-1$ vertices from V' and at least one from $V \setminus V'$. Since the degree of each vertex in G is at most $\lceil\beta\binom{k}{2}\rceil - 1$ and $|S| \leq k$, we have that $|N_G[S]| \leq k\lceil\beta\binom{k}{2}\rceil$. This implies that there exists a vertex $v \in V'$ such that $v \notin N_G[S]$. Hence by including the vertex v and removing a vertex from $S \setminus V'$, we can cover at least as many edges as S can cover. This contradicts our assumption that S is lexicographically smallest. Since G' is a subgraph of G any solution of G' is also a solution of G. Thus we have shown that $OPT(G',k) = OPT(G,k)$. So the algorithm returns the instance (G',k) as the reduced instance. Since G' is a subgraph of G, in this case, the solution lifting algorithm takes a solution S' of (G',k) as input and outputs S' as a solution of (G,k). Since $OPT(G',k) = OPT(G,k)$, it follows that

The number of vertices in the reduced instance is $O(k \cdot \lceil\frac{1}{\epsilon}\binom{k}{2}\rceil^2) = O(k^5)$. The running time of the algorithm is polynomial in the size of G. $\square$

It is also important to note here that as for classical kernelization (Section 1.5.3), there are tools to rule out lossy kernelization (refer to Lokshtanov et al. (2017) for further details).

1.6.2 FPT in P

While initially the main focus of parameterized complexity was on NP-complete problems, the idea of going beyond the worst case analysis by exploiting the structural properties of the input instance is applicable to problems in P too.

Several such algorithms can be found in the literature. For example, by the result of Fomin et al. (2018a), a maximum matching in a graph of treewidth at most k can be constructed in time $\mathcal{O}(k^4 \cdot n \log^2 n)$, while the best known worst-case running time on general graphs is $\mathcal{O}(n^\omega)$, which is due to Mucha and Sankowski (2004). Abboud et al. (2016) proved that the Diameter problem can be solved in time $2^{\mathcal{O}(k \log k)} \cdot n^{1+o(1)}$ on graphs of treewidth k, but achieving running time of the form $2^{o(k)} \cdot n^{2-\varepsilon}$ for any $\varepsilon > 0$ would already contradict the Strong Exponential Time Hypothesis (SETH).

1.6.3 Streaming Algorithms

The ideas used in parameterized algorithms have reached areas beyond classical algorithmic settings including streaming, distributed and dynamic algorithms. In this section we restrict our discussions to graph streaming algorithms. In the streaming model, a graph is presented as a sequence of edges. In the simplest version of this model, we have a stream of edge arrivals (where each edge is added to the graph seen so far), or a mixture of arrivals and departures of edges. In either case, the basic objective is to quickly answer some basic questions over the current state of the graph, such as finding a (maximal) matching over the current graph edges, or finding a (minimum) vertex cover, while storing only a small amount of information. There is a vast literature on streaming algorithms and this survey is not the right place to cover it. Thus, we directly jump to a scenario that is handled using ideas from parameterized algorithms.

In *parameterized streaming algorithms* we seek algorithms whose space and time complexities are bounded with respect to k. Notice that this implies that for certain ranges of k, the space and time usage of the algorithm will be sublinear in the size of the input. Two basic problems that have been studied with respect to parameterized streaming algorithms are Vertex Cover and Maximum Matching.

There are several ways to formalize our objective with respect to Vertex Cover and Maximum Matching when placed in the streaming model, and in the literature several natural models have been considered. The basic case is when the input consists of a sequence of edge arrivals only, for which one seeks a parameterized streaming algorithm (PSA). More challenging problems arise when the input stream is dynamic, and contains both deletions and insertions of edges. In this case one seeks a dynamic parameterized streaming algorithm (DPSA). Notice that when an edge in the current matching is deleted, we sometimes need substantial work to repair the solution, and have to ensure that the algorithm has enough information to do so, while keeping only a bounded amount of working space. If we are promised that at every timestamp there is a solution of cost k, then we seek a promise dynamic parameterized streaming algorithm (PDPSA). These notions were formalized in Chitnis et al. (2015, 2016), and several results for Vertex Cover and Maximum Matching were presented there.

It is an interesting exercise to adapt the $\mathcal{O}(k^2)$ kernel for Vertex Cover into a PSA with space complexity $\mathcal{O}(k^2)$ and time complexity $2^k k^{\mathcal{O}(1)}$ (at any point in the stream, check if the vertex cover is at most k or not). However, other results require a different set of tools and use randomization crucially. There is also a way to show lower bounds on the space used by these algorithms via reduction from some hard problems in communication complexity. For all these we refer to Chitnis et al. (2015, 2016) and Fafianie and Kratsch (2014).

1.6.4 Application Domains

Parameterized complexity focused largely on problems on graphs in the first two decades of its existence. However, in the last decade, problems arising in different domains such as computational geometry, bioinformatics, machine learning and computational social choice theory have been considered through the lens of parameterized complexity. A plethora of parameters such as the dimension of the input, solution size, the number of input strings, patterns in people's preferences, number of candidates, and number of voters have been used as parameters in these domains. These led to numerous results in these areas and several parametrized algorithms or non-existence of such algorithms have been proved. We refer to the following surveys for these [Faliszewski and Niedermeier (2016); Bredereck et al. (2014); Panolan et al. (2019); Giannopoulos et al. (2008)].

1.7 The Big Picture

Parameterized algorithms and complexity has been a spectacular success from a theoretical perspective. From the more applied perspective, only very recently, the PACE challenge kickstarted research into practical implementations for basic FPT problems like Vertex Cover, Treewidth, or Steiner Tree. (See Dell et al. (2017, 2018).) These experimental contests have shown the following:

- Parameterized Complexity seems particularly useful/efficient for problems arising in graph theory and networks.
- PACE implementation challenge has shown that to improve the practical performance of parameterized algorithm one needs to fine-tune parameters and apply "time saving optimization" tricks on top of it. Also, the algorithm that has the "best running time" (this is not unique as the running time is governed by two parameters and thus we can only talk about Pareto-optimality) may not be the best performing algorithm in practice. Thus, in practice we should use parameterized algorithms as a skeleton and optimize other parameters for performance.
- There is huge discrepancy between the running time predictions derived from theoretical analysis of parameterized algorithms (which is worst case and hence overly pessimistic) and the performance of algorithms in practice.

Every success is accompanied with its own list of failures. Unfortunately, there is lack of study of weighted discrete optimization problems, or problems over continuous domains. For example problems arising in the field of computational geometry or mathematical programming. One reason for this is that with respect to solution size, which has become go to parametrization in the field, the problem generally turns out to be intractable. Similarly, for problems where input consists of vectors in $\mathbb{R}^d$, most problems turn out to be W-hard with respect to dimension d.

Thus, for such problems a user should either not use parameterized complexity as a tool to design an algorithm or should avoid using so called classical parameters. Finally, choice of a parameter is an art. One should not restrict oneself with just one parameter for the same problem. It is possible that different parameters are "small" for different families of instances of the same problem and hence together they represent wider tractability of the problem. One thing that the field must do is to interact with other fields and develop tools based on this synergy. For example, combining approximation algorithms and parameterized complexity has yielded several positive results (of course, as well as some exciting negative results).

1.8 Notes

The history of parameterized algorithms can be traced back to 1970s and 80s, including the algorithm of Dreyfus and Wagner (1971) for Steiner Tree, the algorithm of Farber et al. (1986) for Achromatic number and the celebrated result of Robertson and Seymour (1995) on Disjoint Paths. The foundations of parameterized complexity, the toolkit for analyzing parameterized algorithms, were laid in 1990s in the series of papers Abrahamson et al. (1995); Downey and Fellows (1992, 1995a,b). The classic reference on parameterized complexity is the book of Downey and Fellows (1999). The new edition of this book Downey and Fellows (2013) is a comprehensive overview of the state of the art in many areas of parameterized complexity. The book of Flum and Grohe (2006) is an extensive introduction to the area with a strong emphasis on the complexity viewpoint. An introduction to basic algorithmic techniques in parameterized complexity up to 2006 is given in the book of Niedermeier (2006). The recent textbook of Cygan et al. (2015) provides a coherent account of the most recent tools and techniques in the area. The book of Fomin et al. (2019) gives a detailed overview of algorithmic and complexity techniques used in kernelization.

The reference point for branching algorithms is the work of Davis and Putnam (1960) (see also Davis et al. (1962)) on the design and analysis of algorithms to solve some satisfiability problems. Branching algorithms and techniques for their analysis are discussed in detail in the book of Fomin and Kratsch (2010). A branching algorithm with running time $\mathcal{O}(2^k(n+m))$ for Vertex Cover appears in the book of Mehlhorn (1984). After a long sequence of improvements, the current champion algorithm runs in time $\mathcal{O}(1.2738^k + kn)$ and is due to Chen et al. (2010). The method of color coding was introduced in the seminal paper of Alon et al. (1995).

ETH and SETH were first introduced in the work of Impagliazzo and Paturi (2001), which built upon earlier work of Impagliazzo et al. (2001).

Theorem 1.10 on equivalence of kernelization and fixed-parameter tractability is due to Cai et al. (1997). The reduction rules for Vertex Cover discussed in this chapter are attributed to Buss in Buss and Goldsmith (1993) and are often referred to as Buss kernelization in the literature. A more refined set of reduction rules for Vertex Cover was introduced in Balasubramanian et al. (1998). The kernelization algorithm for matrix rigidity is from Fomin et al. (2018b). We refer to the following books an [Downey and Fellows (1999, 2013); Flum and Grohe (2006); Niedermeier (2006); Cygan et al. (2015); Fomin et al. (2019); Fomin and Kratsch (2010)] for detailed overview of all the techniques presented in this survey and more. We also refer the readers to the book of van Rooij et al. (2019) for an application to questions of intractability, with respect to both classical and parameterized complexity analysis, in cognitive science.

References

Abboud, Amir, Williams, Virginia Vassilevska, and Wang, Joshua R. 2016. Approximation and Fixed Parameter Subquadratic Algorithms for Radius and Diameter in Sparse Graphs. Pages 377–391 of: *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM.

Abrahamson, Karl R., Downey, Rodney G., and Fellows, Michael R. 1995. Fixed-Parameter Tractability and Completeness IV: On Completeness for W[P] and PSPACE Analogues. *Ann. Pure Appl. Logic*, **73**(3), 235–276.

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

Balasubramanian, R., Fellows, Michael R., and Raman, Venkatesh. 1998. An Improved Fixed-Parameter Algorithm for Vertex Cover. *Information Processing Letters*, **65**(3), 163–168.

Bodlaender, Hans L., Downey, Rodney G., Fellows, Michael R., and Hermelin, Danny. 2009. On problems without polynomial kernels. *J. Computer and System Sciences*, **75**(8), 423–434.

Bredereck, Robert, Chen, Jiehua, Faliszewski, Piotr, Guo, Jiong, Niedermeier, Rolf, and Woeginger, Gerhard J. 2014. Parameterized Algorithmics for Computational Social Choice: Nine Research Challenges. *CoRR*, **abs/1407.2143**.

Buss, Jonathan F., and Goldsmith, Judy. 1993. Nondeterminism within P. *SIAM J. Computing*, **22**(3), 560–572.

Cai, Liming, Chen, Jianer, Downey, Rodney G., and Fellows, Michael R. 1997. Advice Classes of Parameterized Tractability. *Ann. Pure Appl. Logic*, **84**(1), 119–138.

Chalermsook, Parinya, Cygan, Marek, Kortsarz, Guy, Laekhanukit, Bundit, Manurangsi, Pasin, Nanongkai, Danupon, and Trevisan, Luca. 2017. From Gap-ETH to FPT-Inapproximability: Clique, Dominating Set, and More. Pages 743–754 of: *58th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2017, Berkeley, CA, USA, October 15-17, 2017*. IEEE Computer Society.

Chen, Jianer, and Lu, Songjian. 2009. Improved Parameterized Set Splitting Algorithms: A Probabilistic Approach. *Algorithmica*, **54**(4), 472–489.

Chen, Jianer, Kanj, Iyad A., and Xia, Ge. 2010. Improved upper bounds for vertex cover. *Theoretical Computer Science*, **411**(40-42), 3736–3756.

Chitnis, Rajesh, Cormode, Graham, Esfandiari, Hossein, Hajiaghayi, Mohammad-Taghi, McGregor, Andrew, Monemizadeh, Morteza, and Vorotnikova, Sofya.

2016. Kernelization via sampling with applications to finding Mmatchings and related problems in dynamic graph streams. Pages 1326–1344 of: *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).* SIAM.

Chitnis, Rajesh Hemant, Cormode, Graham, Hajiaghayi, Mohammad Taghi, and Monemizadeh, Morteza. 2015. Parameterized Streaming: Maximal Matching and Vertex Cover. Pages 1234–1251 of: *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015.* SIAM.

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

Davis, Martin, and Putnam, Hilary. 1960. A computing procedure for quantification theory. *J. ACM*, **7**, 201–215.

Davis, Martin, Logemann, George, and Loveland, Donald. 1962. A machine program for theorem-proving. *Communications of the ACM*, **5**, 394–397.

Dell, Holge, and van Melkebeek, Dieter. 2010. Satisfiability Allows No Nontrivial Sparsification Unless The Polynomial-Time Hierarchy Collapses. Page to appear of: *STOC 2010.*

Dell, Holger, Husfeldt, Thore, Jansen, Bart M. P., Kaski, Petteri, Komusiewicz, Christian, and Rosamond, Frances A. 2017. The First Parameterized Algorithms and Computational Experiments Challenge. Pages 30:1–30:9 of: Guo, Jiong, and Hermelin, Danny (eds), *11th International Symposium on Parameterized and Exact Computation (IPEC 2016).* Leibniz International Proceedings in Informatics (LIPIcs), vol. 63. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Dell, Holger, Komusiewicz, Christian, Talmon, Nimrod, and Weller, Mathias. 2018. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. Pages 30:1–30:12 of: Lokshtanov, Daniel, and Nishimura, Naomi (eds), *12th International Symposium on Parameterized and Exact Computation (IPEC 2017).* Leibniz International Proceedings in Informatics (LIPIcs), vol. 89. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Downey, Rodney G., and Fellows, Michael R. 1992. Fixed-parameter tractability and completeness. *Proceedings of the 21st Manitoba Conference on Numerical Mathematics and Computing. Congr. Numer.*, **87**, 161–178.

Downey, Rodney G., and Fellows, Michael R. 1995a. Fixed-Parameter Tractability and Completeness I: Basic Results. *SIAM J. Computing*, **24**(4), 873–921.

Downey, Rodney G., and Fellows, Michael R. 1995b. Fixed-Parameter Tractability and Completeness II: On Completeness for W[1]. *Theoretical Computer Science*, **141**(1&2), 109–131.

Downey, Rodney G., and Fellows, Michael R. 1999. *Parameterized complexity.* New York: Springer-Verlag.

Downey, Rodney G., and Fellows, Michael R. 2013. *Fundamentals of Parameterized Complexity.* Texts in Computer Science. Springer.

Downey, Rodney G., Estivill-Castro, Vladimir, Fellows, Michael R., Prieto-Rodriguez, Elena, and Rosamond, Frances A. 2003. Cutting Up is Hard to Do: the Parameterized Complexity of k-Cut and Related Problems. *Electr. Notes Theor. Comput. Sci.*, **78**, 209–222.

Dreyfus, Stuart E., and Wagner, Robert A. 1971. The Steiner problem in graphs. *Networks*, **1**(3), 195–207.

Fafianie, Stefan, and Kratsch, Stefan. 2014. Streaming kernelization. Pages 275–286 of: *Proceedings of the 39th International Symposium Mathematical Foundations of Computer Science (MFCS)*. Lecture Notes in Comput. Sci., vol. 8635. Springer.

Faliszewski, Piotr, and Niedermeier, Rolf. 2016. Parameterization in Computational Social Choice. Pages 1516–1520 of: *Encyclopedia of Algorithms*.

Farber, Martin, Hahn, Gena, Hell, Pavol, and Miller, Donald J. 1986. Concerning the achromatic number of graphs. *J. Combinatorial Theory Ser. B*, **40**(1), 21–39.

Flum, Jörg, and Grohe, Martin. 2006. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Berlin: Springer-Verlag.

Fomin, Fedor V., and Kratsch, Dieter. 2010. *Exact Exponential Algorithms*. Springer. An EATCS Series: Texts in Theoretical Computer Science.

Fomin, Fedor V., Lokshtanov, Daniel, Saurabh, Saket, Pilipczuk, Michal, and Wrochna, Marcin. 2018a. Fully Polynomial-Time Parameterized Computations for Graphs and Matrices of Low Treewidth. *ACM Trans. Algorithms*, **14**(3), 34:1–34:45.

Fomin, Fedor V., Lokshtanov, Daniel, Meesum, Syed Mohammad, Saurabh, Saket, and Zehavi, Meirav. 2018b. Matrix Rigidity from the Viewpoint of Parameterized Complexity. *SIAM J. Discrete Math.*, **32**(2), 966–985.

Fomin, Fedor V., Lokshtanov, Daniel, Saurabh, Saket, and Zehavi, Meirav. 2019. *Kernelization. Theory of parameterized preprocessing*. Cambridge University Press, Cambridge.

Fortnow, Lance, and Santhanam, Rahul. 2008. Infeasibility of instance compression and succinct PCPs for NP. Pages 133–142 of: *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC)*. ACM.

Giannopoulos, Panos, Knauer, Christian, and Whitesides, Sue. 2008. Parameterized Complexity of Geometric Problems. *Comput. J.*, **51**(3), 372–384.

Gupta, Anupam, Lee, Euiwoong, and Li, Jason. 2018. An FPT Algorithm Beating 2-Approximation for k-Cut. Pages 2821–2837 of: *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*. SIAM.

Impagliazzo, Russell, and Paturi, Ramamohan. 2001. On the Complexity of k-SAT. *J. Computer and System Sciences*, **62**(2), 367–375.

Impagliazzo, Russell, Paturi, Ramamohan, and Zane, Francis. 2001. Which problems have strongly exponential complexity. *J. Computer and System Sciences*, **63**(4), 512–530.

Karthik C. S., Laekhanukit, Bundit, and Manurangsi, Pasin. 2019. On the Parameterized Complexity of Approximating Dominating Set. *J. ACM*, **66**(5), 33:1–33:38.

Kawarabayashi, Ken-Ichi, and Lin, Bingkai. 2020. A nearly 5/3-approximation FPT Algorithm for Min-k-Cut. Page to appear of: *Proceedings of the Thirty First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, Utah, USA, January 6-8, 2020*. SIAM.

Lokshtanov, Daniel, Marx, Dániel, and Saurabh, Saket. 2011. Lower bounds based on the Exponential Time Hypothesis. *Bulletin of the EATCS*, **105**, 41–72.

Lokshtanov, Daniel, Panolan, Fahad, Ramanujan, M. S., and Saurabh, Saket. 2017. Lossy kernelization. Pages 224–237 of: *Proceedings of the 49th Annual ACM Symposium on Theory of Computing (STOC)*. ACM.

Lokshtanov, Daniel, Marx, Dániel, and Saurabh, Saket. 2018. Slightly Superexponential Parameterized Problems. *SIAM J. Comput.*, **47**(3), 675–702.

Mehlhorn, Kurt. 1984. *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness*. EATCS Monographs on Theoretical Computer Science, vol. 2. Springer.

Mucha, Marcin, and Sankowski, Piotr. 2004. Maximum Matchings via Gaussian Elimination. Pages 248–255 of: *FOCS 2004*. IEEE Computer Society.

Naor, Moni, Schulman, Leonard J., and Srinivasan, Aravind. 1995. Splitters and Near-Optimal Derandomization. Pages 182–191 of: *Proceedings of the 36th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE.

Niedermeier, Rolf. 2006. *Invitation to fixed-parameter algorithms*. Oxford Lecture Series in Mathematics and its Applications, vol. 31. Oxford: Oxford University Press.

Panolan, Fahad, Saurabh, Saket, and Zehavi, Meirav. 2019. Parameterized computational geometry via decomposition theorems. Pages 15–27 of: *Proceedings of the 13th International Conference on Algorithms and Computation (WALCOM)*. Lecture Notes in Comput. Sci., vol. 11355. Springer.

Papadimitriou, Christos H., and Yannakakis, Mihalis. 1996. On Limited Nondeterminism and the Complexity of the V-C Dimension. *J. Computer and System Sciences*, **53**(2), 161–170.

Robertson, Neil, and Seymour, Paul D. 1995. Graph minors. XIII. The disjoint paths problem. *J. Combinatorial Theory Ser. B*, **63**(1), 65–110.

Sipser, Michael. 1996. *Introduction to the Theory of Computation*. 1st edn. International Thomson Publishing.

van Rooij, Iris, Blokpoel, Mark, Kwisthout, Johan, and Wareham, Todd. 2019. *Cognition and Intractability: A Guide to Classical and Parameterized Complexity Analysis*. Cambridge University Press.

Exercises

1.1 Give an algorithm for Vertex Cover that runs in time $1.4656^k \cdot n^{\mathcal{O}(1)}$.

1.2 In the Cluster Editing problem, we are given a graph G and an integer k, and the objective is to check whether we can turn G into a cluster graph (a disjoint union of cliques) by making at most k edge editions, where each edition is adding or deleting one edge. Obtain a $3^k n^{\mathcal{O}(1)}$-time algorithm for Cluster Editing.

1.3 Let $\mathcal{F}$ be a set of graphs. We say that a graph G is *$\mathcal{F}$-free* if G does not contain any induced subgraph isomorphic to a graph in $\mathcal{F}$; in this context the elements of $\mathcal{F}$ are sometimes called *forbidden induced subgraphs*. For a fixed set $\mathcal{F}$, consider a problem where, given a graph G and an integer k, we ask to turn G into a $\mathcal{F}$-free graph by:

(vertex deletion) deleting at most k vertices;
(edge deletion) deleting at most k edges;
(completion) adding at most k edges;
(edition) performing at most k editions, where every edition is adding or deleting one edge.

Prove that, if $\mathcal{F}$ is finite, then for each of the four aforementioned problems there exists a $2^{\mathcal{O}(k)}n^{\mathcal{O}(1)}$-time FPT algorithm. (Note that the constants hidden in the $\mathcal{O}()$-notation may depend on the set $\mathcal{F}$.)

1.4 Prove that a random assignment ψ splits k sets with probability at least $\frac{1}{2^k}$ (see Section 1.2.1).

1.5 Show that Partial Vertex Cover is solvable in time $2^{O(t)}n^{O(1)}$, where t is the number of covered edges.