Parameterized Algorithmics for Finding Exact Solutions of NP-Hard Biological Problems

Falk Hüffner[1] Christian Komusiewicz[1] Rolf Niedermeier[1]
Sebastian Wernicke[2]

[1]Institut für Softwaretechnik und Theoretische Informatik, TU Berlin, Germany
{falk.hueffner,christian.komusiewicz,rolf.niedermeier}@tu-berlin.de
[2]Seven Bridges Genomics, Cambridge, MA, USA
sebastian.wernicke@sbgenomics.com

Abstract

Fixed-parameter algorithms are designed to efficiently find optimal solutions to some computationally hard (NP-hard) problems by identifying and exploiting "small" problem-specific parameters. We survey practical techniques to develop such algorithms. Each technique is introduced and supported by case studies of applications to biological problems, with additional pointers to experimental results.

Key Words: Computational intractability; NP-hard problems; algorithm design; exponential running times; discrete problems; fixed-parameter tractability; optimal solutions.

1 Introduction

Many problems that emerge in bioinformatics require vast amounts of computer time to be solved optimally. An illustrative example, though somewhat oversimplified, would be the following: Given a set of n experiments of which some pairs have conflicting results (that is, at least one result must be wrong), identify a minimum-size subset of experiments to eliminate such that no conflict remains. This problem, while simple to describe, has no known algorithm that solves it efficiently on all inputs. From a theoretical standpoint, such computational hardness can be traced back to the NP-*hardness* of a problem. Assuming a widely believed conjecture in complexity theory, the classification of a computational problem as NP-hard implies that the time needed to solve it grows very quickly (usually exponentially) with the input size [62]. However, the demand to solve NP-hard problems commonly arises in practical settings, including bioinformatics. To obtain solutions to these problems despite their NP-hardness, it is common to sacrifice solution quality for efficiency, for example by employing heuristic algorithms or approximation algorithms. A different approach is to insist on exact solutions and accept that the algorithm will not be efficient on all inputs but hopefully on those that arise in the application at hand.

Most theory on computational hardness is based on the assumption that the difficulty of solving an instance of a computational problem is determined by the size of that instance. The crucial observation this chapter is based on is that

often it is not the *size* of an instance that makes a problem computationally hard to solve, but rather its *structure*. *Parameterized algorithmics* renders this observation precise by quantifying structural hardness with so-called *parameters*, typically a nonnegative integer variable denoted by k or a tuple of such variables. A parameterized problem is then called *fixed-parameter tractable* (FPT) if it can be solved efficiently when the parameter is small; the corresponding algorithm is called *fixed-parameter algorithm*. The concept of fixed-parameter tractability thus formalizes and generalizes the concept of "tractable special cases" that are known for virtually all NP-hard problems. For example, as we will discuss in more detail below, our introductory problem can be solved quickly whenever the number of conflicting experiments is small (a reasonable assumption in practical settings, since the results would otherwise not be worth much anyway).

Often, there are many possible parameters to choose from. For example, for solving our introductory problem we could choose the maximum number of conflicts for a single experiment to be the parameter or, alternatively, the size of the largest group of pairwise conflicting experiments. This makes parameterized algorithmics a multipronged attack that can be adapted to different practical applications. Of course, not all parameters lead to efficient algorithms; in fact, parameterized algorithmics also provides tools to classify parameters as "not helpful" in the sense that we cannot expect provably efficient algorithms even when these parameters are small.

Fixed-parameter algorithms have by now facilitated many success stories and several techniques have emerged as being applicable to large classes of problems [84]. This chapter presents several of these techniques, namely kernelization (Section 2), depth-bounded search trees (Section 3), dynamic programming (Section 4), tree decompositions of graphs (Section 5), color-coding (Section 6), and iterative compression (Section 7). We start each section by introducing the basic concepts and ideas, followed by some case studies concerning practically relevant bioinformatics problems. Concluding each section, we survey known applications, implementations, and experimental results, thereby highlighting the strengths and fields of applicability for each technique.

Another commonly used strategy for exactly solving NP-hard problems is to reduce the problem at hand to "general-purpose problems" such as integer linear programming [7, 8] and satisfiability solving [15, 106, 127]. For these, there exist highly optimized tools with years of algorithm engineering effort that went into their development. Therefore, if an NP-hard problem can be efficiently expressed as one of these general-purpose problems, these tools might be able to find an optimum solution without the need for any further algorithm design. In many application scenarios, it will actually make sense to try and combine these general-purpose approaches and the more problem-specific approach of parameterized algorithmics since the specific advantage of fixed-parameter algorithms is that they are usually crafted directly for the problem at hand and thus may allow a better exploitation of problem-specific features to substantially gain efficiency. In particular, the polynomial-time data reduction techniques that are introduced in Section 2 usually combine nicely and productively with the more general solver tools.

Before discussing the main techniques of fixed-parameter algorithms in the following sections, the remainder of this section provides a crash course in computational complexity theory and a few formal definitions related to parameterized complexity analysis. Furthermore, some terms from graph theory are introduced,

and we present our running example problem Vertex Cover.

1.1 Computational Complexity Theory

In this survey, we are concerned with efficiently solving computational problems. A standard format for specifying these problems is to phrase them in an "Input / Task" way that formally specifies the input and desired output. A core topic of computational complexity theory is the evaluation and comparison of different algorithms for a given problem [107, 114]. Since most algorithms are designed to work with variable inputs, the efficiency (or *complexity*) of an algorithm is not just stated for some concrete inputs (*instances*), but rather as a function that relates the input length n to the number of steps that are required to execute the algorithm. Generally, this function is given in an asymptotic sense, the standard way being the *big-O notation* where we write $f(n) = O(g(n))$ to express that $f(n)/g(n)$ is upper-bounded by a positive constant in the limit for large n [46, 87, 123]. Since instances of the same size might take different amounts of time, it is implicitly assumed in this chapter that we are considering the *worst-case* running time among all instances of the same size; that is, we deliberately exclude from our analysis the potentially efficient solvability of some specific input instances of a computational problem.

Determining the computational complexity of problems (meaning the best possible worst-case running time of an algorithm for them) is a key issue in theoretical computer science. Herein, it is of central importance to distinguish between problems that can be solved efficiently and those that presumably cannot. To this end, theoretical computer science has coined the notions of *polynomial-time solvability* on the one hand and NP-*hardness* on the other [62]. Here, polynomial-time solvability means that for every size-n input instance of a problem, an optimal solution can be computed in $n^{O(1)}$ time. In contrast, the (unproven, yet widely believed) working hypothesis of theoretical computer science is that NP-hard problems cannot be solved in $n^{O(1)}$ time. More specifically, typical running times for NP-hard problems are of the form $O(c^n)$ for some constant $c > 1$; that is, we have an exponential growth in the number of computation steps as instances grow larger. In this sense, polynomial-time solvability has become a synonym for efficient solvability.

As there are thousands of known NP-hard optimization problems and their number is continuously growing [107, 115], several approaches have been developed that try to circumvent the assumed computational intractability of NP-hard problems. One such approach is based on polynomial-time approximation algorithms, where one gives up seeking optimal solutions in order to have efficient algorithms [9, 128, 131]. Another common strategy is to use heuristics, where one gives up provable performance guarantees (concerning running time or solution quality) by developing algorithms that behave well in "most" practical applications [105, 107].

1.2 Parameterized Complexity

For many applications, the compromises inherent to approximation algorithms and heuristics are not satisfactory. Fixed-parameter algorithms can provide an alternative by providing exact solutions with useful running time guarantees [54, 60, 110]. The core concept is formalized as follows.

Definition 1. *A parameterized problem instance consists of a problem instance I and a parameter k. A parameterized problem is* fixed-parameter tractable *if it can be solved in $f(k) \cdot |I|^{O(1)}$ time, where f is a (computable) function solely depending on the parameter k.*

For NP-hard problems, $f(k)$ will typically be an exponential function like 2^k rather than a polynomial function.

Note that the concept of fixed-parameter tractability is different from the notion of "polynomial-time solvable for fixed k"; an algorithm running in $|I|^{f(k)}$ time demonstrates that a problem is polynomial-time solvable for any fixed k, but does not show fixed-parameter tractability, since the exponent needs to be a constant; ideally, a fixed-parameter algorithm provides a linear-time algorithm for each fixed k [14].

As an example for this "parameterized perspective", consider again the identification of k faulty experiments among n experiments. We could naively solve this problem in $O(2^n)$ time by trying all possible subsets of the n experiments. However, this would not be practically feasible for $n > 40$. In contrast, a simple fixed-parameter algorithm with running time $O(2^k \cdot n)$ exists for this problem, which allows it to be solved even for $n > 1000$, as long as $k < 20$ (as we will discuss in Section 2.4, real-world instances can often be solved for much larger values of k by an extension of this approach).

Unfortunately, there are parameterized problems for which there is good evidence that they are not fixed-parameter tractable (see Note 1).

A few words on the art of problem parameterization. Typically, a problem allows for more than one parameterization [91, 111]. From a theoretical point of view, parameterization is a key to better understand the nature of computational intractability. The ultimate goal here is to learn how parameters influence the computational complexity of problems. The more we know about these interactions, the more likely it becomes to cope with computational intractability. In a sense, it may be considered as an art to find the most useful parameterizations of a computational problem.

From an applied point of view, the identification of parameters for a concrete problem should go hand-in-hand with an extensive data analysis. One natural way for spotting relevant parameterizations of a problem in real-world applications is to analyze the given input data and check which quantifiable aspects of it appear to be small and might thus be suitable as parameters. For example, if the input is a network, one such observable parameter could be the maximum vertex degree. Often, real-world input instances also carry some hidden structure that might be exploited. Again turning to graphs, well-known parameters such as such as "feedback vertex set number" or "treewidth" measure how tree-like a graph is. These parameters are motivated by the observation that many intractable graph problems become tractable when restricted to trees. For NP-hard string problems, which also occur frequently in bioinformatics, natural parameters are for example the size of the alphabet or the number of occurrences of a letter [36].

1.3 Graph Theory

Many of the problems we deal with in this work can be formulated in graph-theoretic terms [49, 129]. An undirected graph $G = (V, E)$ is given by a set

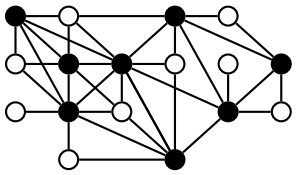

Figure 1: A graph with a size-8 vertex cover (cover vertices are marked black).

of *vertices* V and a set of *edges* E, where each edge $\{v, w\}$ is an undirected connection of two vertices v and w. Throughout this work, we use $n := |V|$ to denote the number of vertices and $m := |E|$ to denote the number of edges. For a set of vertices $V' \subseteq V$, the *induced subgraph* $G[V']$ is the graph $(V', \{\{v, w\} \in E \mid v, w \in V'\})$, that is, the graph G restricted to the vertices in V'. We denote the *open neighborhood* of a vertex v by $N(v) := \{u \mid \{u, v\} \in E\}$ and its *closed neighborhood* by $N[v] := N(v) \cup \{v\}$.

It is not hard to see that we can formalize our introductory problem of recognizing faulty experiments as a graph problem where vertices correspond to experiments and edges correspond to pairs of conflicting experiments. Thus, we need to choose a small set of vertices (the experiments to eliminate) so that each edge is incident with at least one chosen vertex. This is known as the NP-hard Vertex Cover problem, which serves as a running example for several techniques in this work.

> Vertex Cover
> **Input:** An undirected graph $G = (V, E)$ and a nonnegative integer k.
> **Task:** Find a set $C \subseteq V$ of at most k vertices such that each edge in E has at least one of its endpoints in C.

The problem is illustrated in Figure 1. Vertex Cover can well be considered a poster child of fixed-parameter research, as many discoveries that influenced the whole field originated from the study of this single problem.

2 Kernelization: Data Reduction With Guaranteed Effectiveness

The idea of data reduction is to quickly presolve those parts of a given problem instance that are easy to cope with, shrinking it to those parts that form its hard core [76, 95]. Computationally expensive algorithms need then only be applied to this core. In some practical scenarios, data reduction may even reduce instances of a seemingly hard problem to triviality. Once an effective (and efficient) reduction rule has been found, it is typically not only useful in the context of parameterized algorithmics, but also in other problem solving contexts, whether they be heuristic, approximative, or exact.

This section introduces the concept of *kernelization*, that is, polynomial-time data reduction with guaranteed effectiveness. Kernelization is closely connected to fixed-parameter tractability and emerges within its framework.

2.1 Basic Concepts

There are many examples of combinatorial problems that would not be solvable without employing heuristic data reduction and preprocessing algorithms. For example, commercial solvers for hard combinatorial problems such as the integer linear program solver CPLEX heavily rely on data-reducing preprocessors for their efficiency [16]. Obviously, many practitioners are aware of the general concept of data reduction. Parameterized algorithmics adds to this by providing a way to use data reduction rules not only heuristically, but with guaranteed performance quality. These so-called *kernelizations* guarantee an upper bound on the size of the reduced instance, which solely depends on the parameter value. More precisely, the concept is defined as follows:

Definition 2 ([54, 110])**.** *Let I be an instance of a parameterized problem with given parameter k. A* reduction to a problem kernel *(or* kernelization*) is a polynomial-time algorithm that replaces I by a new instance I' and k by a new parameter k' such that*

- *the size of I' and the value of k' are guaranteed to only depend on some function of k, and*
- *the new instance I' has a solution with respect to the new parameter k' if and only if I has a solution with respect to the original parameter k.*

Kernelizations can help to understand the practical effectiveness of some data reduction rules and, conversely, the quest for kernelizations can lead to new and powerful data reduction rules based on deep structural insights.

Intriguingly, there is a close connection between fixed-parameter tractable problems and those problems for which there exists a kernelization—in fact, they are exactly the same [38]. Unfortunately, the running time of a fixed-parameter algorithm directly obtained from a kernelization is usually not practical and, in the other direction, there exists no constructive scheme for developing data reduction rules for a fixed-parameter tractable problem. Nevertheless, this equivalence can establish the fixed-parameter tractability and amenability to kernelization of a problem by knowing just one of these two properties.

2.2 Case Studies

In this section, we first illustrate the concept of kernelization by a simple example concerning the Vertex Cover problem. We then show a more involved kernelization algorithm for the graph clustering problem Cluster Editing. Finally, we discuss the limits of the kernelization approach for fixed-parameter tractable problems and present an extension of the kernelization concept that can be used to cope with the nonexistence of problem kernels.

2.2.1 A Simple Kernelization for Vertex Cover

Consider our running example Vertex Cover. In order to cover an edge in the graph, one of its two endpoints must be in the vertex cover. If one of these is a degree-1 vertex (that is, it has exactly one neighbor), then the other endpoint has the potential to cover more edges than this degree-1 vertex, leading to a first data reduction rule.

Reduction Rule VC1
If there is a degree-1 vertex, then put its neighboring vertex into the cover.

Here, "put into the cover" means adding the vertex to the solution set and removing it and its incident edges from the instance. Note that this reduction rule assumes that we are only looking for *one* optimal solution to the Vertex Cover instance we are trying to solve; there may exist other minimum vertex covers that do include the reduced degree-1 vertex (see Note 2).

After having applied Rule VC1, we can further do the following in the fixed-parameter setting where we ask for a vertex cover of size at most k.

Reduction Rule VC2
If there is a vertex v of degree at least $k+1$, then put v into the cover.

The reason this rule is correct is that if we did not take v into the cover, then we would have to take every single one of its $k+1$ neighbors into the cover in order to cover all edges incident with v. This is not possible because the maximum allowed size of the cover is k.

After exhaustively performing Rules VC1 and VC2, all vertices in the remaining graph have degree at most k. Thus, at most k edges can be covered by choosing an additional vertex into the cover. Since the solution set may be no larger than k, the remaining graph can have at most k^2 edges if it has a solution. Clearly, we can assume without loss of generality that there are no isolated vertices (that is, vertices with no incident edges) in a given instance. In conjunction with Rule VC1, this means that every vertex has degree at least two. Hence, the remaining graph can contain at most k^2 vertices.

Stepping back, what we have just done is the following: After applying two *polynomial-time* data reduction rules to an instance of Vertex Cover, we arrived at a reduced instance whose size can *be expressed solely in terms of the parameter* k. Hence, considering Definition 2, we have found a kernelization for Vertex Cover.

2.2.2 A Kernelization for Cluster Editing

In the above example kernelization for Vertex Cover, there is a notable difference between Rules VC1 and VC2: Rule VC1 is based on a local optimality argument whereas Rule VC2 makes explicit use of the parameter k. In applications, the first type of data reduction rules is usually preferable, as they can be applied independently of the value of k and this value is only used in the analysis of the power of the data reduction rules. For the NP-hard graph clustering problem Cluster Editing, we now present an efficient kernelization algorithm that is based solely on a data reduction rule of the first type.

Cluster Editing
Input: An undirected graph $G = (V, E)$, an edge-weight function $\omega : V^2 \to \mathbb{N}^+$, and a nonnegative integer k.
Task: Find whether we can modify G to consist of vertex-disjoint cliques (that is, fully connected components) by adding or deleting a set of edges whose weights sum up to at most k.

Cluster Editing can be used, for example, to cluster proteins with high sequence similarity [22] and to identify cancer subtypes [133]; a comprehensive

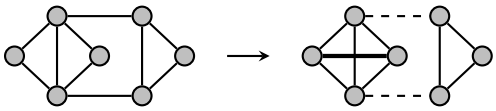

Figure 2: Illustration for Cluster Editing with unit weights: By removing two edges from and adding one edge to the graph on the left (that is, $k = 3$), we can obtain a graph that consists of two vertex-disjoint cliques.

overview of its applications is given by Böcker & Baumbach [21]. Many theoretical studies consider the case in which all edges have weight one, but the weighted version of Cluster Editing is more relevant in biological applications. The positive edge weights describe the cost to delete an existing edge or to insert a missing edge, respectively. Figure 2 shows an instance of the unweighted problem variant together with a solution. A simple kernelization for Cluster Editing uses similar high-degree reduction rules as the Vertex Cover kernelization described above. These rules yield a kernel with $O(k^2)$ vertices [66]. This bound can be improved to $O(k)$ vertices using reduction rules whose correctness is based on local optimality arguments. We now describe such a kernelization algorithm for Cluster Editing that was developed by Cao & Chen [40].

The idea of this kernelization is to examine for each vertex v of the graph whether its neighborhood is already very dense and only loosely connected to the rest of the graph. If this is the case, then it is optimal to put all neighbors of v in the same cluster as v. This knowledge can be used to identify edges that have to be deleted or edges that have to be added. Formally, the algorithm computes the sum of the weights of the missing edges in the neighborhood $N[v]$; this number is denoted by $\delta(v)$. Then it computes the sum of the edge weights between $N[v]$ and $V \setminus N[v]$; this number is denoted $\gamma(v)$. These two measures are combined to form what is called the stable cost of a vertex v defined as $c(v) = 2\delta(v) + \gamma(v)$. Now a vertex v is called *reducible* if $c(v) < |N[v]|$. The main consequence of being reducible is that if $N[v]$ is reducible, then there is an optimal solution such that $N[v]$ is contained in a single cluster. This implies the correctness of the following data reduction rules; an example application of the first two reduction rules is given in Figure 3.

The first rule adds missing edges in neighborhoods of reducible vertices.

> Reduction Rule CE1
> If there is a reducible vertex v and a pair of vertices u, x in $N[v]$ that are not neighbors, then add $\{u, x\}$ to G and decrease k by $\omega(\{u, x\})$.

The next rule finds vertices that have some but only few neighbors in $N[v]$. In an optimal solution, these vertices are never in the same cluster as $N[v]$. Thus, the edges between these vertices and $N[v]$ may be deleted.

> Reduction Rule CE2
> If there is a reducible vertex v and a vertex $u \notin N[v]$ such that it is more costly to add all missing edges between u and $N[v]$ than to remove all edges between u and $N[v]$, then remove all edges between u and $N[v]$ and decrease k accordingly.

The final rule merges $N[v]$ into one vertex and adjusts the edge weights accordingly.

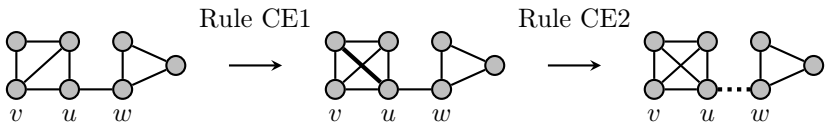

Figure 3: The application of Reduction Rules CE1 and CE2 to an instance of Cluster Editing. In the example, the weight of all existing and missing edges is 1. Initially, the vertex v is reducible. Then, Rule CE1 inserts the missing edge in $N[v]$. Subsequently, Rule CE2 deletes the edge between u and w, since it is more costly to make w adjacent to all vertices of $N[v]$ than to delete this edge.

Reduction Rule CE3
If there is a reducible vertex v to which Rules CE1 and CE2 do not apply, then merge $N[v]$ into a single vertex v'. For each vertex $u \in V \setminus N[v]$ set $\omega(\{u, v'\}) := \sum_{x \in N[v]} \omega(\{u, x\})$.

As long as the instance contains a reducible vertex, the reduction rules will either modify an edge or merge a vertex. If there are no more reducible vertices, then the last trivial step of the kernelization algorithm is to remove all isolated vertices from the instance. Afterwards, the instance has at most $2k$ vertices. The intuition behind this size bound is the following. Every edge has weight at least one, so a solution contains at most k edges. Since each edge has two endpoints, the modifications can affect at most $2k$ vertices. Now if in a cluster every vertex is affected, then the size of the cluster is at least two times the number of edge modifications within the cluster plus the number of edge modifications between this cluster and other clusters. If there is a vertex v in the cluster that is not affected by the solution, then the same bound on the cluster size holds, in this case because v is not reducible. Summing these size bounds over all clusters, we obtain a sum of edges in which each solution edge appears at most twice. This gives the size bound of $2k$ vertices.

2.3 Limits and Extensions of Kernelization

The two example problems Vertex Cover and Cluster Editing are especially amenable to kernelization since they admit *polynomial-size problem kernels*. That is, the size bound for the kernel is a polynomial function in the parameter k. While all fixed-parameter tractable problems admit a problem kernelization, it is not the case that all fixed-parameter tractable problems admit *polynomial* kernels [54, 95]. It is beyond the scope of this paper to introduce the proof techniques for showing nonexistence of polynomial problem kernels. We will give, however, an example of a biologically motivated graph problem that does not admit a polynomial problem kernel and describe one way of circumventing this hardness result.

2-Club
Input: An undirected graph $G = (V, E)$ and a nonnegative integer k.
Task: Find a set $S \subseteq V$ of at least k vertices such that the subgraph induced by S has diameter at most two.

The NP-hard 2-Club problem attempts to identify large cohesive subgroups of an input graph. The idea behind the formulation is to relax the overly restrictive

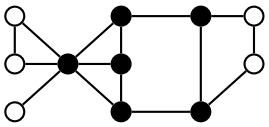

Figure 4: A graph with a 2-club of size six (marked black).

definition of the Clique problem which only accepts solutions that are complete graphs or, equivalently, that have diameter one. The 2-Club problem finds applications in the detection of protein interaction complexes [116]; an instance of 2-Club with a maximum-cardinality solution is shown in Figure 4. As we will see, 2-Club is fixed-parameter tractable with respect to the parameter solution size k. It does not, however, admit a polynomial problem kernel for this parameter [120] (see Note 3 for a brief discussion).

In spite of this hardness result, one can still perform a useful parameterized data reduction for 2-Club. The idea is to reduce the problem to many problem kernels instead of just one. This approach is called *Turing kernelization*. In the case of 2-Club, the Turing kernelization consists of two simple parts. First, one looks for a trivial solution using the following observation: for every vertex v in a graph, its closed neighborhood $N[v]$ has diameter two.

Reduction Rule 2-C
If there is a vertex v with at least $k-1$ neighbors, then return $N[v]$.

After this rule has been applied, we have either obtained a solution or the maximum degree of the graph is at most $k-2$. Now, the Turing kernelization uses only one further observation: To find a largest 2-club it is sufficient to examine for each vertex v of the input graph G the subgraph of G that contains only the vertices which have distance at most two to v. We can now use the fact that the maximum degree is bounded: every vertex v has at most $k-2$ neighbors and each of these has at most $k-3$ further neighbors. Thus, 2-Club can be solved by independently solving n small instances with $O(k^2)$ vertices each. Formally, this means that 2-Club admits a Turing kernelization with $O(k^2)$ vertices.

2.4 Applications and Implementations

Solving Vertex Cover is relevant in many bioinformatics-related scenarios such as analysis of gene expression data [45] and the computation of multiple sequence alignments [41]. Besides solving instances of Vertex Cover, another application of Vertex Cover kernelizations is to search maximum-cardinality cliques (that is, maximum-size complete subgraphs) in a graph. Here, use is made of the fact that an n-vertex graph G has a clique of size $(n-k)$ if and only if its complement graph, that is, the graph that contains exactly the edges not contained in G, has a size-k vertex cover. The best known kernel for Vertex Cover (up to minor improvements) has $2k$ vertices [109]. Abu-Khzam et al. [1] studied various kernelization methods for Vertex Cover and their practical performance on biological networks with respect to running time and resulting kernel size. Experimental results for the computation of large cliques via Vertex Cover are given, for example, by Abu-Khzam et al. [3].

Several kernelization approaches including the one presented in Section 2.2.2 have been implemented for Cluster Editing [23, 77]. The Turing kernelization for 2-Club was implemented and experimentally evaluated; it turned out to be a crucial ingredient for obtaining an efficient algorithm for this problem [78]. Another biologically relevant clustering problem where kernelizations have been successfully implemented is the Clique Cover problem. Here, the task is to cover all edges of a graph using at most k cliques (these may overlap). Using data reduction, Gramm et al. [67] solved even large instances with 1 000 vertices and $k \approx 6\,000$ as long as they are sparse ($m \approx 7\,000$).

3 Depth-Bounded Search Trees

Once data reductions as discussed in the previous section have been applied to a problem instance, we are left with the "really hard" problem kernel to be solved. A standard way to explore the huge search space of a computationally hard problem is to perform a systematic exhaustive search. This can be organized in a tree-like fashion, which is the subject of this section.

3.1 Basic Concepts

Search tree algorithms—also known as backtracking algorithms, branching algorithms, or splitting algorithms—certainly are no new idea and have extensively been used in the design of exact algorithms (e.g., see [46, 61, 123]). The main contribution of parameterized algorithmics to search tree algorithms is the consideration of search trees whose *depth is constrained by a function in the parameter*. Combined with insights on how to find useful—and possibly non-obvious—parameters, this can lead to search trees that are much smaller than those of naive brute-force searches. For example, a very naive search tree approach for solving Vertex Cover is to just take one vertex and branch into two cases: either this vertex is in the vertex cover or not. For an n-vertex graph, this leads to a search tree of size $O(2^n)$. As we outline in this section, we can do much better than that and obtain a search tree whose depth is upper-bounded by k, giving a size bound of $O(2^k)$. Extending what we discuss here, even better search trees of size $O(1.28^k)$ are possible [42]. Since usually $k \ll n$, this can draw the problem into the zone of feasibility even for large graphs.

Besides depth-bounding, parameterized algorithmics provides additional means to provably improve the speed of search tree exploration, particularly by interleaving this exploration with kernelizations, that is, applying data reduction to partially solved instances during the exploration.

3.2 Case Studies

Starting with our running example Vertex Cover, this section introduces the concept of depth-bounded search trees by three case studies.

3.2.1 Vertex Cover Revisited

For many search tree algorithms, the basic idea is to find a small subset of the input instance in polynomial time such that at least one element of this subset must be part of an optimal solution to the problem. In the case of Vertex

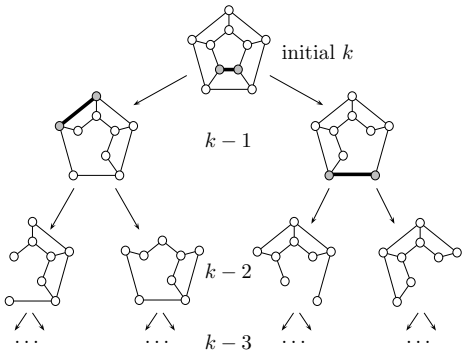

Figure 5: Simple search tree for finding a vertex cover of size at most k in a given graph. The size of the tree is $O(2^k)$.

Cover, the most simple such subset is any set of two adjacent vertices. By definition of the problem, one of these two vertices has to be part of a solution, or the respective edge would not be covered. Thus, a simple search-tree algorithm to solve Vertex Cover on a graph $G = (V, E)$ can proceed by picking an arbitrary edge $e = \{v, w\}$ and recursively searching for a vertex cover of size $k - 1$ both in $G[V \setminus \{v\}]$ and $G[V \setminus \{w\}]$, that is, in the graphs obtained by removing either v and its incident edges or w and its incident edges. In this way, the algorithm branches into two subcases knowing one of them must lead to a solution of size at most k (provided that it exists).

As shown in Figure 5, the recursive calls of the simple Vertex Cover algorithm can be visualized as a tree structure. Because the depth of the recursion is upper-bounded by the parameter value and we always branch into two subcases, the number of cases that are considered by this tree—its size, so to say—is $O(2^k)$. Independent of the size of the input instance, it only depends on the value of the parameter k.

The currently "best" search trees for Vertex Cover have worst-case size $O(1.28^k)$ [42] and are mainly achieved by elaborate case distinctions. These algorithms consist of several branching rules; for example, the degrees of the endpoints of an edge determine which of the branching rules is applied. However, for practical applications it is always concrete implementation and testing that has to decide whether the administrative overhead caused by distinguishing more and more cases pays off. A simpler algorithm with slightly worse search tree size bounds may be preferable.

3.2.2 A Search Tree Algorithm for Cluster Editing

For Vertex Cover, we have found a depth-bounded search tree by observing that at least one endpoint of any given edge must be part of the cover. A somewhat similar approach can be used to derive a depth-bounded search tree for Cluster Editing.

Recall that the aim for Cluster Editing is to modify a graph into a cluster graph, that is, a vertex-disjoint union of cliques, by modifying edges whose weight sums up to at most k. Similar to Vertex Cover, a search tree for Cluster

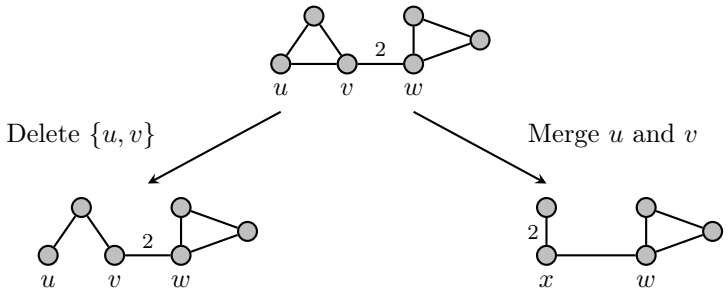

Figure 6: The merge branching for Cluster Editing. In the example instance, all edges and missing edges have unit weight, except $\{v, w\}$ which has weight 2. In one branch, the edge $\{u, v\}$ is deleted and k is reduced by 1. In the other branch, u and v are merged and the edge weights are adjusted. For example, edge $\{x, w\}$ obtains weight 1 since the missing edge $\{u, w\}$ had weight 1. Accordingly, k is decreased by 1. All missing edges between x and other vertices have weight 2.

Editing can be obtained by noting that the desired graph of vertex-disjoint cliques forbids a certain structure: If two vertices in a cluster graph are adjacent, then their neighborhoods must be the same. Hence, whenever we encounter two vertices u and v in the input graph G that are adjacent and where one vertex, say v, has a neighbor w that is not adjacent to u, we are compelled to do one of three things: Either remove the edge $\{u, v\}$, or add the edge $\{u, w\}$, or remove the edge $\{v, w\}$. Note that each such modification incurs a cost of at least one. Therefore, exhaustively branching into three cases, each time decreasing k by one, we obtain a search tree of size $O(3^k)$ to solve Cluster Editing. Using computer-aided algorithm design, this idea can be improved to obtain, for the unit-weight case, a search tree of size $O(1.92^k)$ [65]. The current-best theoretical running time is, however, achieved by exploiting the fact that edge weights make it possible to consider the merging operation in a search tree algorithm. The observation is that in the presence of a conflict as described above, one may either delete the edge $\{u, v\}$ or, otherwise, u and v are in the same cluster of the final cluster graph. Thus, one may merge u and v and adjust the edge weights accordingly. The main trick is that when performing the merging, this still causes some cost: The edge $\{v, w\}$ must be deleted or the edge $\{u, w\}$ must be added. After merging u and v into a new vertex x one may thus "remember" that the new edge $\{x, w\}$ will incur a cost irrespective of whether this edge is deleted or kept by a solution. With a more refined branching strategy, this idea leads to a search tree of size $O(1.82^k)$ for the general case [22] and of size $O(1.62^k)$ for the unit-weight case [20].

3.2.3 The Closest String Problem

The Closest String problem is also known as Consensus String.

Closest String
Input: A set of k length-ℓ strings $s_1, \ldots, s_k$ and a nonnegative integer d.
Task: Find a *consensus string* s that satisfies $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$.

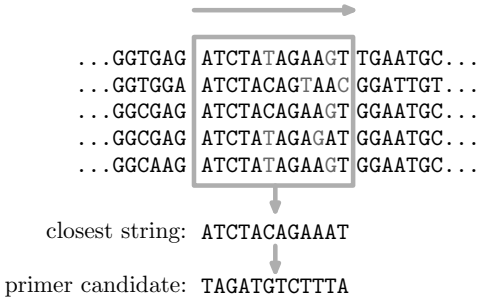

Figure 7: Illustration to show how DNA primer design can be achieved by solving Closest String instances on length-ℓ windows of aligned DNA sequences. The primer candidate is not the computed consensus string but its nucleotide-wise complement.

Here, $d_H(s, s_i)$ denotes the Hamming distance between two strings s and s_i, that is, the number of positions where s and s_i differ. Note that there are at least two immediately obvious parameterizations of this problem. The first is given by choosing the "distance parameter" d and the second is given by the number of input strings k. Both parameters are reasonably small in various applications; we refer to Gramm et al. [70] for more details. Here, we focus on the parameter d.

Closest String appears for example in *primer design*, where we try to find a small DNA sequence called *primer* that binds to a set of (longer) target DNA sequences as a starting point for replication of these sequences. How well the primer binds to a sequence is mostly determined by the number of positions in that sequence that hybridize to it. While often done by hand, Stojanovic et al. [125] proposed a computational approach for finding a well-binding primer of length ℓ. First, the target sequences are aligned, that is, as many matching positions within the sequences as possible are grouped into columns. Then, a "sliding window" of length ℓ is moved over this alignment, giving a Closest String problem for each window position. Figure 7 illustrates this (see [64] for details).

In the remainder of this case study, we sketch a fixed-parameter search tree algorithm for Closest String due to Gramm et al. [70], the parameter being the distance d. Unlike for Vertex Cover and Cluster Editing, the central challenge lies in even *finding* a depth-bounded search tree, which is not obvious at a first glance. Once found, however, the derivation of the upper bound for the search tree size is straightforward. The underlying algorithm is very simple to implement.

The main idea behind the algorithm is to maintain a *candidate string* $\hat{s}$ for the center string and compare it to the strings $s_1, \ldots, s_k$. If $\hat{s}$ differs from some s_i in more than d positions, then we know that $\hat{s}$ needs to be modified in at least one of these positions to match the character that s_i has there. Consider the following observation:

Observation 1. *Let d be a nonnegative integer. If two strings s_i and s_j have a Hamming distance greater than $2d$, then there is no string that has a Hamming*

distance of at most d to both of s_i and s_j.

This means that s_i is allowed to differ from $\hat{s}$ in at most $2d$ positions. Hence, among any $d+1$ of those positions where s_i differs from $\hat{s}$, at least one must be modified to match s_i. This can be used to obtain a search tree that solves Closest String.

We start with a string from $\{s_1, \ldots, s_k\}$ as the candidate string $\hat{s}$, knowing that a center string can differ from it in at most d positions. If $\hat{s}$ already is a valid center string, we are done. Otherwise, there exists a string s_i that differs from $\hat{s}$ in more than d positions, but less than $2d$. Choosing any $d+1$ of these positions, we branch into $(d+1)$ subcases, each subcase modifying a position in $\hat{s}$ to match s_i. This position cannot be changed anymore further down in the search tree (otherwise, it would not have made sense to make it match s_i at that position). Hence, the depth of the search tree is upper-bounded by d, for if we were to go deeper down in the tree, then $\hat{s}$ would differ in more than d positions from the original string we started with. Thus, Closest String can be solved by exploring a search tree of size $O((d+1)^d)$ [70]. Combining data reduction with this search tree, we arrive at the following.

Theorem 1. Center String *can be solved in $O(k \cdot \ell + k \cdot d \cdot (d+1)^d)$ time.*

It might seem as if this result is purely of theoretical interest—after all, the term $(d+1)^d$ becomes prohibitively large already for $d = 15$. Two things, however, should be noted in this respect: First, for one of the main applications of Closest String, primer design, d is very small (often less than 4). Second, empirical analysis reveals that when the algorithm is applied to real-world and random instances, it often beats the proven upper bound by far, solving many real-world instances in less than a second. The algorithm is also faster than a simple integer linear programming formulation of Closest String when the input consists of many strings and ℓ is small [70].

Unfortunately, many variants of Closest String—roughly speaking, these deal with finding a matching *sub*string and distinguish between strings to which the center is supposed to be close and to which it should be distant—are known to be intractable for many standard parameters [57, 68, 104].

3.3 Applications and Implementations

In combination with data reduction, the use of depth-bounded search trees has proven itself quite useful in practice, for example allowing to find vertex covers of more than ten thousand vertices in some dense graphs of biological origin [3]. It should also be noted that search trees trivially allow for a parallel implementation: when branching into subcases, each process in a parallel setting can further explore one of these branches with no additional communication required. Experimental results for Vertex Cover show linear speedups even for thousands of cores [2].

The merge-based search tree algorithm for Cluster Editing can solve many instances arising in the analysis of protein similarity data [22]; it is part of a software package [132]. A fixed-parameter search tree algorithm was also used to solve instances of the Minimum Common String Partition problem [35]. This NP-hard problem is motivated by applications in comparative genomics; the fixed-parameter algorithm was able to solve the problem on some bacterial genomes.

The parameters exploited by the algorithm are the number of breakpoints and the maximum gene copy number in the genomes. Fixed-parameter search tree algorithms have also been applied for solving the Maximum Agreement Forest problem which arises in the comparison of phylogenetic trees [130]; the fixed-parameter algorithm outperformed two previous approaches for Maximum Agreement Forest, one using a formulation as integer linear program and another one using a formulation as satisfiability problem. Another example is the search for *k-plexes* in graphs, which can be used for example to model functional modules in protein interaction networks. By combining search trees with data reduction, it is often possible to outperform previously-used methods [108].

Besides in parameterized algorithmics, search tree algorithms are studied extensively in the area of artificial intelligence and heuristic state space search. There, the key to speedups are *admissible heuristic evaluation functions* which quickly give a lower bound on the distance to the goal. The reason that admissible heuristics are rarely considered by the parameterized algorithmics community in their works (see [69] for a counterexample) is that they typically cannot improve the asymptotic running time. Still, the speedups obtained in practice can be quite pronounced, as demonstrated for Vertex Cover [58].

As with kernelizations, algorithmic developments outside the fixed-parameter setting can make use of the insights that have been gained in the development of depth-bounded search trees in a fixed-parameter setting. One example for this is the Minimum Quartet Inconsistency problem arising in the construction of evolutionary trees. Here, an algorithm that uses depth-bounded search trees was developed by Gramm & Niedermeier [69]. Their insight was used by Wu et al. [134] to develop a faster (non-parameterized) algorithm for this problem.

In conclusion, depth-bounded search trees with clever branching rules are certainly one of the first approaches to try when solving fixed-parameter tractable problems in practice.

4 Dynamic Programming

Dynamic programming is one of the most useful algorithm design techniques in bioinformatics; it also plays an important role in developing fixed-parameter algorithms. Since dynamic programming is a classic algorithm design technique covered in many standard textbooks [123], we keep the presentation of "fixed-parameter dynamic programming" short.

4.1 Basic Concepts

The general idea is to recursively break down the problem into possibly overlapping subproblems whose optimal solution allows to find an overall optimal solution. The solutions to subproblems are stored in a table, avoiding recalculation. A classic example is sequence alignment of two strings, for instance using the Needleman–Wunsch algorithm [19]. The dynamic programming technique, however, is not restricted to polynomial-time solvable problems.

The running time of dynamic programming depends mainly on the table size, so the main trick in obtaining fixed-parameter dynamic programming algorithms is to bound the size of the table by a function of the parameter times a polynomial in the input size. Two generic methods for this are tree decompositions and

color-coding, described in Sections 5 and 6, respectively. In many cases, however, the table size is obviously bounded in the parameter and thus no additional techniques are necessary to obtain a fixed-parameter algorithm.

4.2 Case Study

One application of dynamic programming is in the interpretation of mass spectrometry data, which contains mass peaks for a sample molecule and for fragments thereof [27, 28]. The method builds a graph where a vertex corresponds to a possible molecular formula of a peak, and an edge corresponds to a hypothetical fragmentation step. Edges are weighted by the likeliness of the corresponding fragmentation step. The goal is then to calculate a maximum scoring subtree of this graph. In this tree, we must use only one of the molecular formulas of a peak. This is achieved by giving each vertex a corresponding color and asking for a *colorful* subtree.

> Maximum Colorful Subtree
> **Input:** A directed graph $D = (V, A)$ with a vertex coloring $c : V \to C$ and arc weights $w : A \to \mathbb{Q}_+$.
> **Task:** Find a subtree of G that uses each color at most once and has maximum total arc weight.

This NP-hard problem can be solved by dynamic programming [27, 28, 121] by building a table $W(v, S)$ for $v \in V$ and $S \subseteq C$. An entry $W(v, S)$ holds the maximum score of a subtree with root v whose vertex set has exactly the colors of S. The table is filled out with the following recurrence:

$$W(v,S) = \max \begin{cases} \max_{u \in V : c(u) \in S \setminus \{c(v)\}} W(u, S \setminus \{c(v)\}) + w(v,u) \\ \max_{\substack{(S_1,S_2) : S_1 \cap S_2 = \{c(v)\} \\ S_1 \cup S_2 = S}} W(v,S_1) + W(v,S_2) \end{cases} \tag{1}$$

with initial condition $W(v, \{c(v)\}) = 0$ and the weight of nonexistent arcs set to $-\infty$. The first line extends a tree by introducing v as new root and adding the arc (v, u), and the second line merges two trees that have the same root but are otherwise disjoint.

The table W has $n \cdot 2^k$ entries where k is the number of different vertex colors, and filling it out can be done in $O(3^k km)$ time. Thus, Maximum Colorful Subtree is fixed-parameter tractable with respect to the parameter k. In the application, the parameter is the number of peaks in the spectrum which is usually small.

4.3 Applications and Implementations

The algorithm described in Section 4.2 was found to be fast and accurate in determining glycan structure [27]. There are several further applications of dynamic programming over exponentially-sized tables. In phylogenetics, for example, the task of reconciling a binary gene tree with a nonbinary species trees can be solved via a dynamic programming algorithm whose table size is exponential only in the maximum outdegree of the species tree [126]. The implementation solves instances based on cyanobacterial gene trees on average in less than one second. In these instances, the parameter value ranges from 2 to 6.

Another application of dynamic programming is in a variant of haplotyping (see also Section 7.2) which deals with the analysis of genomic fragments. Using dynamic programming, solutions to the *weighted minimum error correction* formulation can be found in a running time of $O(2^k \cdot m)$. Here, k is the maximum coverage of any genome position by the input fragments and m is the number of SNPs per sequencing read [117]. The algorithm scales up to $k \approx 20$.

5 Tree Decompositions of Graphs

Many NP-hard graph problems become computationally feasible when they are restricted to cycle-free graphs, that is, *trees* or collections of trees (*forests*). Trees, while potentially simplifying computation, form a very limited class of graphs that seldom suffices as a model for real-life applications. Hence, as a compromise between general graphs and trees, one might want to look at "tree-like" graphs. This tree-likeness can be formalized by the concept of *tree decompositions*. In this section, we survey some important aspects of tree decompositions and their algorithmic use with respect to computational biology and FPT. Surveys on this topic are given by Berger et al. [11] and Bodlaender & Koster [31].

5.1 Basic Concepts

There is a very helpful and intuitive characterization of tree decompositions in terms of a robber–cop game in a graph [29]: A robber stands on a graph vertex and, at any time, he can run at arbitrary speed to any other vertex of the graph as long as there is a path connecting both. The only restriction is that he is not permitted to run through a cop. There can be several cops and, at any time, each of them may either stand on a graph vertex or be in a helicopter (that is, she is above the game board and can move anywhere without being restricted by graph edges). The cops want to land a helicopter on the vertex occupied by the robber. The robber can see a helicopter approaching its landing vertex and he may run to a new vertex before the helicopter actually lands. Thus, the cops want to occupy all vertices adjacent to the robber's vertex, making him unable to move, and to then land one more remaining helicopter on the robber's vertex itself to catch him. The *treewidth* of the graph is the minimum number of cops needed to catch a robber minus one (observe that if the graph is a tree, two cops suffice and trees hence have a treewidth of one) and a corresponding *tree decomposition* is a tree structure that provides the cops with a scheme to catch the robber. Intuitively, the tree decomposition indicates "bottlenecks" (separators) in the graph and thus reveals an underlying scaffold that can be exploited algorithmically.

Formally, tree decompositions and treewidth center around the following somewhat technical definition; Figure 8 shows a graph together with an optimal tree decomposition of width two.

Definition 3. *Let $G = (V, E)$ be an undirected graph. A* tree decomposition *of G is a pair $\langle \{X_i \mid i \in I\}, T \rangle$ where each X_i is a subset of V, called a* bag*, and T is a tree with the elements of I as nodes. The following three properties must hold:*

1. $\bigcup_{i \in I} X_i = V$;

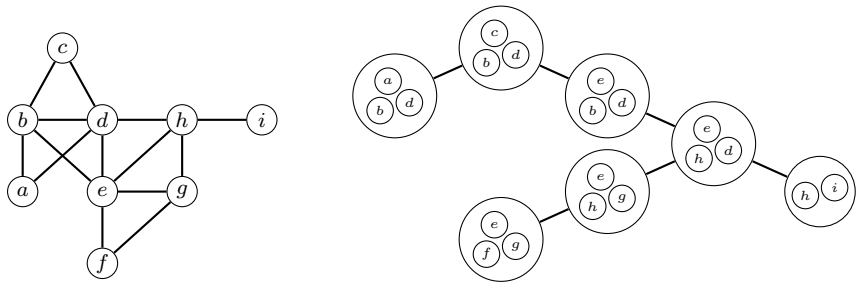

Figure 8: A graph together with a tree decomposition of width 2. Observe that—as demanded by the consistency property—each graph vertex induces a subtree in the decomposition tree.

2. *for every edge* $\{u, v\} \in E$*, there is an* $i \in I$ *such that* $\{u, v\} \subseteq X_i$*; and*
3. *for all* $i, j, k \in I$*, if* j *lies on the path between* i *and* k *in* T *then* $X_i \cap X_k \subseteq X_j$*.*

The width *of* $\langle \{X_i \mid i \in I\}, T \rangle$ *equals* $\max\{|X_i| \mid i \in I\} - 1$*. The* treewidth *of* G *is the minimum* k *such that* G *has a tree decomposition of width* k*.*

The third condition of the definition is often called *consistency property*. It is important in dynamic programming, the main algorithmic tool when solving problems on graphs of bounded treewidth. An equivalent formulation of this property is to demand that for any graph vertex v, all bags containing v form a connected subtree.

For trees, the bags of a corresponding tree decomposition are simply the two-element vertex sets formed by the edges of the tree. In the definition, the subtraction of 1 thus ensures that trees have a treewidth of 1. In contrast, a clique of n vertices has treewidth $n - 1$. The corresponding tree decomposition trivially consists of one bag containing all graph vertices; in fact, no tree decomposition with smaller width is attainable since it is known that every complete subgraph of a graph G is completely "contained" in a bag of G's tree decomposition.

Tree decompositions of graphs are connected to another central concept in algorithmic graph theory: *graph separators* are vertex sets whose removal from the graph separates the graph into two or more connected components. Each bag of a tree decomposition forms a separator of the corresponding graph.

Given a graph, determining its treewidth is an NP-hard problem itself. However, several tools and heuristics exist that construct tree decompositions [30–32], and for some graphs that appear in practice, computing a tree decomposition is easy. Here, we concentrate on the algorithmic use of tree decompositions, assuming that they are provided to us.

5.2 Case Study

Typically, tree decomposition-based algorithms have two stages:

1. Find a tree decomposition of bounded width for the input graph.
2. Solve the problem by dynamic programming on the tree decomposition, starting from the leaves.

Intuitively speaking, a decomposition tree provides us with a scaffold-structure that allows for efficient and consistent processing through the graph. By design, this scaffold leads to optimal solutions even when the utilized tree decompositions are not optimal; however, the algorithm will run slower and consume more memory in that case.

To exemplify dynamic programming on tree decompositions, we make use of our running example Vertex Cover and sketch a fixed-parameter dynamic programming algorithm for Vertex Cover with respect to the parameter treewidth.

Theorem 2. *For a graph G with a given width-ω tree decomposition $\langle \{X_i \mid i \in I\}, T\rangle$, an optimal vertex cover can be computed in $O(2^\omega \cdot \omega \cdot |I|)$ time.*

The basic idea of the algorithm is to examine for each bag X_i all of the at most $2^{|X_i|}$ possibilities to obtain a vertex cover for the subgraph $G[X_i]$. This information is stored in tables A_i, $i \in I$. Adjacent tables are updated in a bottom-up process starting at the leaves of the decomposition tree. Each bag of the tree decomposition thus has a table associated with it. During this updating process it is guaranteed that the "local" solutions for each subgraph associated with a bag of the tree decomposition are combined into a "globally optimal" solution for the overall graph G. (We omit several technical details here; these can be found in [110, Chapter 10].) The following points of Definition 3 guarantee the validity of this approach.

1. The first condition in Definition 3, that is, $V = \bigcup_{i \in I} X_i$, makes sure that every graph vertex is taken into account during the computation.
2. The second condition in Definition 3, that is, $\forall e \in E\ \exists i \in I :\ e \in X_i$, makes sure that all edges can be treated and thus will be covered.
3. The third condition in Definition 3 guarantees the consistency of the dynamic programming, since information concerning a particular vertex v is only propagated between neighboring bags that both contain v.

While the running time of the dynamic programming part can often be improved over a naive approach, there is evidence that known algorithms for some basic combinatorial problems are essentially optimal [102].

One thing to keep in mind for a practical application is that storing dynamic programming tables requires memory space that grows exponentially in the treewidth. Hence, even for "small" treewidths, say, between 10 and 20, the computer program may run out of memory and break down. Some techniques for limiting memory use have been proposed [12, 56, 71].

5.3 Applications and Implementations

Tree decomposition based algorithms are a valuable alternative whenever the underlying graphs have small treewidth. As a rule of thumb, the typical border of practical feasibility lies somewhere below a treewidth of 20 for the underlying graph, although with advantageous data and careful implementation higher values are possible (e. g. [71]). Successful implementations for solving Vertex Cover with tree decomposition approaches have been reported [4, 12].

A practical application of tree decompositions is found in protein structure prediction, namely the prediction of backbone structures and side-chain prediction. These two problems can be modeled as a graph labeling problem, where the resulting graphs have a very small treewidth in practice, allowing the problems to be solved efficiently [11].

Besides taking an input graph, computing a tree decomposition for it, and hoping that the resulting tree decomposition has small treewidth, there have also been cases where a problem is modeled as a graph problem such that it can be *proven* that the resulting graphs have a tree decomposition with small treewidth that can efficiently be found. As an example, Song et al. [124] used a so-called conformational graph to specify the consensus sequence-structure of an RNA family. They proved that the treewidth of this graph is basically determined by the structural elements that appear in the RNA. More precisely, they showed that if there is a bounded number of crossing stems, say k, in a pseudoknot structure, then the resulting graph has treewidth $(2 + k)$. Since the number of crossing stems is usually small, this yields a fast algorithm for searching RNA secondary structures (see also [135]).

Other biological applications include peptide sequencing and spectral alignment [101], molecule bond multiplicity inference [24], charge group partitioning for biomolecular simulations [39], and NMR interpretation [99]. The idea of exploiting the treewidth of an auxiliary structure describing interdependencies of the input also has attracted much attention in Artificial Intelligence (AI) applications [63, 86].

Besides dynamic programming, a very powerful method to obtain fixed-parameter results for the parameter treewidth is to cast the problem as an expression in *monadic second-order logic* (MSO) [98]. For example for Vertex Cover, the expression is

$$\mathrm{vc}(U) := \forall x, y \in V : \neg(\{x, y\} \in E) \vee x \in U \vee y \in U.$$

Since the worst-case running time obtained from this formulation is extremely bad, this approach was thought to be impractical [110, Chapter 10]. However, recently a solver was presented that indeed just requires the user to provide the MSO expression [88, 97, 98]. If the problem at hand admits a formulation in MSO (as most problems that are fixed-parameter tractable for treewidth do), this provides a quick way to evaluate the feasibility of the treewidth approach for the data at hand, with the option to get a quicker algorithm by designing a customized dynamic programming.

Besides treewidth, a number of alternative concepts have been developed to compare the structure of a graph to a tree, including branch-width, rank-width, and hypertree-width [79, 98].

6 Color-Coding

The color-coding technique due to Alon et al. [6] is a general method for finding small patterns in graphs. In its simplest form, color-coding can solve the Minimum Weight Path problem, which asks for the cheapest path of length k in a graph. This has been successfully employed with protein–protein interaction networks to find signaling pathways [85, 121] and to evaluate pathway similarity queries [122].

6.1 Basic Concepts

A naive approach to discover a small structure of k vertices within a graph of n vertices would be to combinatorially try all of the roughly n^k possibilities of selecting k out of n vertices and then testing the selection for the desired structural property. This approach quickly leads to a combinatorial explosion, making it infeasible even for rather small input graphs of a few hundred vertices. The central idea of color-coding is to randomly color each vertex of a graph with one of k colors and to hope that all vertices in the subgraph searched for obtain different colors (that is, the vertex set becomes *colorful*).

When the structure that is searched for becomes colorful, the task of finding it can be solved by dynamic programming in a running time where the exponential part solely depends on k, the size of the substructure searched for. Of course, given the randomness of the initial coloring, most of the time the target structure will actually not be colorful. Therefore, we have to repeat the process of random coloring and searching (called a *trial*) many times until the target structure is colorful at least once with sufficiently high probability. As we will show, the number of trials also depends only on k (albeit exponentially). Consequently this algorithm has a fixed-parameter running time. Thus it is much faster than the naive approach which needs $O(n^k)$ time.

6.2 Case Study

Formally stated, the problem we consider is the following:

> Minimum Weight Path
> **Input:** An undirected graph G with edge weights $w : E \to \mathbb{Q}^+$ and a nonnegative integer k.
> **Task:** Find a simple length-k path in G that minimizes the sum over its edge weights.

This problem is well-known to be NP-hard [62, ND29]. What makes the problem hard is the requirement of *simple* paths, that is, paths where no vertex may occur more than once (otherwise, it is easily solved by traversing a minimum-weight edge $k-1$ times).

Given a fixed coloring of vertices, finding a minimum-weight path that is colorful can be accomplished by dynamic programming: Assume that for some $i < k$ we have computed a value $W(v, S)$ for every vertex $v \in V$ and every cardinality-i subset S of vertex colors such that $W(v, S)$ denotes the minimum weight of a path that uses each color in S exactly once and ends in v. Clearly, the resulting path is simple because no color is used more than once. We can now use this to compute the values $W(v, S)$ for all cardinality-$(i+1)$ subsets S and vertices $v \in V$, because any colorful length-$(i+1)$ path that ends in a vertex $v \in V$ must be composed of a colorful length-i path that does not use the color of v and ends in a neighbor of v. More precisely, we let

$$W(v, S) = \min_{e=\{u,v\} \in E} \Big(W(u, S \setminus \{\text{color}(v)\}) + w(e) \Big). \tag{2}$$

See Figure 9 for an example.

It is straightforward to verify that on an m-edge graph the dynamic programming takes $O(2^k m)$ time. Whenever the minimum-weight length-k path P in

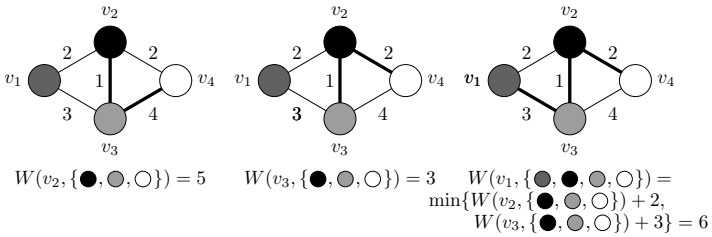

Figure 9: Example for solving Minimum Weight Path using the color-coding technique. Here, using (2) a new table entry (right) is calculated using two already known entries (left and middle).

the input graph is colored with k different colors (that is, its vertex is colorful), then the algorithm finds P. The problem, of course, is that the coloring of the input graph is random and hence many coloring trials have to be performed to ensure that the minimum-weight path is found with a high probability. More precisely, the probability of any length-k path (including the one with minimum weight) being colorful in a single trial is

$$P_c = \frac{k!}{k^k} > \sqrt{2\pi k} e^{-k} \tag{3}$$

because there are k^k ways to arbitrarily color k vertices with k colors and $k!$ ways to color them such that no color is used more than once. Using t trials, a path of length k is found with probability $1-(1-P_c)^t$. Therefore, to ensure that a colorful path is found with a probability greater than $1-\varepsilon$ (for any $0 < \varepsilon \leq 1$), at least

$$t(\varepsilon) = \left\lceil \frac{\ln \varepsilon}{\ln(1-P_c)} \right\rceil = -\ln \varepsilon \cdot O(e^k) \tag{4}$$

trials are needed. This bounds the overall running time by $2^{O(k)} \cdot n^{O(1)}$. While the result is only correct with a certain probability, we can can specify any desired error probability, say 0.1%, noting that even very low error probabilities do not incur excessive extra running time costs.

Note that the number of colors chosen poses a trade-off: While using more than k colors increases the chance of a target structure becoming colorful—and thus decreases the number of trials needed to achieve a given error probability—it increases the running time and memory requirements of the dynamic programming step. As a theoretical analysis points out, using $1.3k$ colors instead of just k improves the worst-case running time of the color-coding algorithm. Moreover, in practice it is often beneficial to increase the number of colors even further [85].

6.3 Applications and Implementations

Protein interaction networks represent proteins by vertices and mutual protein–protein interaction probabilities by weighted edges. They are a valuable source of information for understanding the functional organization of the proteome. Scott et al. [121] demonstrated that *high-scoring simple paths* in the network constitute plausible candidates for linear signal transduction pathways, *simple* meaning that no vertex occurs more than once and *high-scoring* meaning that the product of

edge weights is maximized. To match the above definition of Minimum Weight Path, one works with the *weight* $w(e) := -\log p(e)$ of an edge e with interaction probability $p(e)$ between e's endpoints. Then minimizing the sum of the weights is equivalent to maximizing the product of the probabilities.

The currently most efficient implementation based on color-coding [85] is capable of finding optimal paths of length up to 20 in seconds within a yeast protein interaction network containing about 4 500 vertices.

A particularly appealing aspect of color-coding is that it can be easily adapted to many practically relevant variations of the problem formulation:

- The set of vertices where a path can start and end can be restricted (such as to force it to start in a membrane protein and end in a transcription factor [121]).
- Not only the minimum-weight path can be computed but rather a collection of low-weight paths (typically, one demands that these paths must differ in a certain amount of vertices to ensure that they are diverse and not small modifications of the global minimum-weight path) [85].
- More generally, pathway queries to a network, that is, the task of finding a pathway in a network that is as similar as possible to a query pathway, can be handled with color-coding [122].

Several other works use color-coding for querying in protein interaction networks. For example, the queries can be trees, allowing for identification of non-exact (homeomorphic) matches [53]. Another application is counting non-induced occurrences of subgraph topologies in the form of trees and bounded treewidth subgraphs [5].

A further use of color-coding is to solve the Graph Motif problem. In a biological application of Graph Motif, the query is a set of proteins, and the task is to find a matching set of proteins that are sequence-similar to the query proteins and span a connected region of the network. Bruckner et al. [34] and Betzler et al. [13] provided implementations based on color-coding; they differ in the way insertions and deletions are handled, and are thus not directly comparable.

Further, color-coding has also found applications in string problems: for example, Bonizzoni et al. [33] used it to solve a variant of Longest Common Subsequence that is motivated by a sequence comparison problem. However, to the best of our knowledge no string algorithm using color-coding has been implemented yet.

Related techniques. We mention some techniques that use ideas similar to color-coding. To the best of our knowledge, with one exception none of them has been implemented so far.

Two variants use only two colors to separate the pattern from surrounding vertices (*random separation*) [37] or to divide the graph into two parts for recursion (*divide-and-color*) [89]. Random separation can be used to find small subgraphs with desired properties in sparse graphs. For these problems enumerating connected subgraphs and using color-coding [92] sometimes gives faster algorithms. A further extension known as *chromatic coding* was used to

obtain (theoretically) fast algorithms for the Dense Triplet Inconsistency problem motivated from phylogenetics [73].

Algebraic techniques [93, 94] can improve on the worst-case running time of many color-coding approaches; for example, the currently strongest worst-case bound for Graph Motif is obtained this way [17]. This approach, however, is not as flexible as color-coding, for example with respect to the handling of large weights. Experiments for the unweighted version of Minimum-Weight Path on random graphs have shown that the approach is feasible for a path length of 16 and 8000 vertices [18].

7 Iterative Compression

The main idea of iterative compression is induction: we construct a slightly smaller instance, solve it recursively, and then make use of the solution to solve the actual instance. While induction is a classic algorithmic approach, iterative compression first appeared in a work by Reed et al. in 2004 (see also a 2009 survey [75]). Although it is perhaps not quite as generally applicable as data reduction or search trees, it appears to be useful for solving a wide range of problems and has led to significant breakthroughs in showing fixed-parameter tractability results. Iterative compression is typically used for "minimum obstruction deletion" problems: given a set of items, omit the minimum number of items such that the remaining items exhibit some "nice" structure. Thus, it can sometimes model parsimonious error correction. Vertex Cover is one example fitting this scheme, and it can be solved with iterative compression [118].

7.1 Basic Concepts

The central concept of iterative compression is to employ a so-called *compression routine*.

Definition 4. *A* compression routine *is an algorithm that, given a problem instance and a solution of size k, either calculates a smaller solution or proves that the given solution is of minimum size.*

With a compression routine, we can find an optimal solution for an instance by recursively solving a smaller instance, using the solution for the smaller instance to find a possibly suboptimal solution for the actual instance, and then using the compression routine to find an optimal solution. For "minimum obstruction deletion" problems, the only nontrivial step is the compression routine.

The main strength of iterative compression is that it allows us to see a problem from a different angle, since the compression routine does not only have the problem instance as input, but also a solution, which carries valuable structural information on the input. Also, the compression routine does not need to find an optimal solution at once, but only any better solution. Therefore, the design of a compression routine can often be simpler than designing a complete algorithm.

Algorithmically, the compression routine is the "complex" step in iterative compression in two regards: First, while the mode of use of the compression routine is usually straightforward, finding the compression routine itself often is not. Second, if the compression routine is a fixed-parameter algorithm with respect to the parameter k, then so is the whole algorithm.

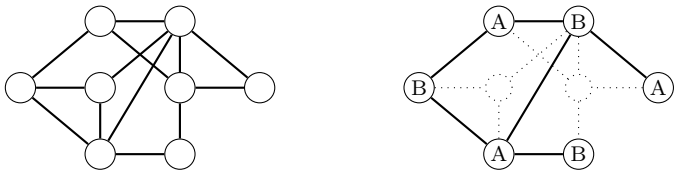

Figure 10: A Vertex Bipartization instance (left), and an optimal solution (right): when deleting two fragments (dashed), the remaining fragments can be allocated to the two chromosome copies (A and B) such that no conflicting fragments get the same assignment.

7.2 Case Studies

The showcase for iterative compression is the Vertex Bipartization problem, also known as Odd Cycle Cover.

> Vertex Bipartization
> **Input:** An undirected graph $G = (V, E)$ and a nonnegative integer k.
> **Task:** Find a set $D \subseteq V$ of at most k vertices such that $G[V \setminus D]$ is bipartite.

This problem appears as Minimum Fragment Removal in the context of SNP haplotyping [113]. When analyzing DNA fragments obtained by shotgun sequencing, it is initially unknown which of the two chromosome copies of a diploid organism a fragment belongs to. We can, however, determine for some pairs of fragments that they cannot belong to the same chromosome copy since they contain conflicting information at some SNP locus. Using this information, it is straightforward to reconstruct the chromosome assignment. We can model this as a graph problem, where the fragments are the vertices and a conflict is represented as an edge. The task is then to color the vertices with two colors such that no vertices with the same color are adjacent. The problem gets difficult in the presence of errors such as parasite DNA fragments which randomly conflict with other fragments. In this scenario, we ask for the least number of fragments to remove such that we can get a consistent fragment assignment (see Figure 10). Using the number of fragments k to be removed as a parameter is a natural approach, since the result is only meaningful for small k anyway.

Iterative compression provided the first fixed-parameter algorithm for Vertex Bipartization with this parameter [119]. We sketch how to apply this to finding an optimal solution (a *removal set*) for a Vertex Bipartization instance $(G = (V, E), k)$. Choose an arbitrary vertex v and let G' be G with v deleted. Recursively find an optimal removal set R' for G' (this recursion terminates after $n = |V|$ steps, where we can yield the empty removal set for the empty graph). Clearly, $R' \cup \{v\}$ is a removal set for G, although it might not be optimal (it can be too large by one). Now using the compression routine for G and $R' \cup \{v\}$, we can find an optimal solution for G'.

The compression routine itself works by examining a number of vertex cuts in an auxiliary graph (that is, a set of vertices whose deletion makes the graph disconnected), a task which can be accomplished in polynomial time by maximum flow techniques. We refer to the literature for details [81, 96, 119]. The running time of the complete algorithm is $O(3^k \cdot mn)$ [81].

7.3 Applications and Implementations

The iterative compression algorithm for Vertex Bipartization has been employed for a number of biological applications. An implementation, improved by heuristics, can solve all Minimum Fragment Removal problems from a testbed based on human genome data within minutes, whereas established methods are only able to solve about half of the instances within reasonable time [81]. The Unordered Maximum Tree Orientation, which models inference of signal transmissions in protein–protein interaction networks based on cause–effect pairs, can be reduced to Graph Bipartization [25]. Also, ordering and orienting contigs produced during genome assembly can be reduced to Graph Bipartization, and this is implemented in the SCARPA scaffolder [52]. Recently, an algorithm with a better worst-case bound of $2.32^k \cdot n^{O(1)}$ based on linear programming was presented [103], which seems like a promising alternative to iterative compression for Vertex Bipartization.

Edge Bipartization, the edge deletion version of Vertex Bipartization, can also be solved by iterative compression [74]. Enhanced with data reduction rules and generalized to the Signed Graph Balancing problem, this algorithm was used to analyze gene regulatory networks [82]. It can solve many networks to optimality, but fails for the largest ones [82]. The Tanglegram Layout problem is about drawing two phylogenetic trees on the same species set in order to facilitate analysis; it can be reduced to Edge Bipartization [26]. The implementation by Hüffner et al. [82] can find exact solutions for all practically relevant Tanglegram Layout instances within seconds [26]. Finally, computing the minimum number of recombination events for general pedigrees with two sites for all members can also be reduced to Edge Bipartization [50].

Another prominent problem amenable to iterative compression is Feedback Vertex Set, which also has applications for genetic linkage analysis [10]. While initial algorithms based on iterative compression [48, 74] had prohibitive worst-case running times, the currently fastest known approach runs in $3.619^k \cdot n^{O(1)}$ time for finding a feedback vertex set of k vertices [90]. However, these algorithms have not been implemented yet.

The Directed Feedback Vertex Set problem was also shown to be fixed-parameter tractable by iterative compression [43], solving a long-standing open question. However, the worst-case running time bound is much worse than for the previously mentioned problems. Still, an experimental evaluation on random graphs [59], employing also data reduction, showed encouraging results for very small parameter values. Directed Feedback Vertex Set has applications in pairwise genome alignment under the duplication-loss model [51] and in the comparison of gene orders [72]. For an application in reconstructing reticulation networks in particular, the authors mention that the parameter could be expected to be very small [100].

Finally, the Cluster Vertex Deletion problem, the "vertex deletion variant" of Cluster Editing, aims to cluster objects by removing objects that do not fit in the cluster structure. It can also be solved by a fixed-parameter algorithm with respect to the number of removed vertices using iterative compression [83].

8 A Roadmap towards Efficient Implementations

Here we try to give some general recommendations on how to go about applying parameterized algorithmics to NP-hard computational problems in practice.

Identification of parameters. The first task is to identify fruitful parameters. As detailed in Section 1.2, it is useful to consider several "structural" parameters, possibly also deduced from a data-driven analysis of the input instances. The usefulness of the parameter clearly depends on whether it is small in the input instances. For graph instances, a tool such as Graphana (http://fpt.akt.tu-berlin.de/graphana/) that calculates a wide range of graph parameters can be helpful. At this point, it is also useful to determine whether the problem is fixed-parameter tractable or W[1]-hard. While a hardness result encourages to look for another parameter or combined parameters, bear in mind that certain techniques such as data reduction can still be effective in practice even without a performance guarantee.

Implementation of brute-force search. The next thing to do is to implement a brute-force search that is as simple as possible. There are several reasons for this: First, it gives some first impression on what solutions look like (for example, can we use their size as parameter?). Second, a simple starting implementation is invaluable in shaking out bugs from later, more sophisticated implementations, in particular if results for random instances are systematically compared. Possibly the best way to get a simple brute-force result is to use an integer linear program (ILP). These sometimes need only a few lines when using a modeling language, but are often surprisingly effective. The second method of choice is a simple search tree (Section 3).

Implementation of data reduction. Data reduction is valuable in combination with any other algorithmic technique such as approximation, heuristics, or fixed-parameter algorithms. In some cases it can even completely solve instances without further effort; it can be considered as essential for the treatment of NP-hard problems. Thus it should always be the first nontrivial technique to be developed and implemented. When combined with even a naive brute-force approach, it can often already solve instances of notable size. For large instances, an efficient implementation of the data reduction rules is necessary. A rule of thumb is to aim for linear running time for most of the implemented data reduction rules and to apply linear-time data reduction rules first [14].

Tuned search trees. After this, the easiest speedups typically come from a more carefully tuned search tree algorithm. Case distinction can help to improve provable running time bounds, although it has often been reported that a too complicated branching actually leads to a slowdown. Heuristic branching priorities can help, as well as admissible heuristic evaluation functions [58]. Further, interleaving with data reduction can lead to a speedup [112].

Non-traditional techniques. When search trees are not applicable or too slow, less clear instructions can be given. The best thing to do is to look at other fixed-parameter algorithms and techniques for inspiration: are we looking

for a small pattern in the input? Possibly color-coding (Section 6) helps. Are we looking for minimum modifications to obtain a nice combinatorial structure? Possibly iterative compression (Section 7) is applicable. In this way, possibly using some of the less common approaches of fixed-parameter algorithms, one might still come up with a fixed-parameter algorithm.

Here, one should be wary of exponential-space algorithms as these can often fill the memory within seconds and therefore become unusable in practice. In contrast, one should *not* be too afraid of bad upper bounds for fixed-parameter algorithms—the analysis is worst-case and often much too pessimistic.

Heuristic speedups. Some of the largest speedups experienced in experiments come from techniques that can be considered heuristic in the sense that they do not improve worst-case time bounds or the kernel size. The general idea of most heuristics is to recognize early that some branches or subcases cannot lead to an optimal solution and to skip those. Their potential effectiveness, even when no performance guarantees can be given, should always be kept in mind when implementing algorithms.

Furthermore, most algorithms will have numerous degrees of freedom concerning their actual implementation, execution order, and the value of some thresholds for example concerning the fraction of search tree nodes to which data reduction should be applied. There are tools for algorithm configuration that can exploit this freedom and may yield magnitudes of speedup [80].

9 Conclusion

We surveyed several techniques for developing efficient fixed-parameter algorithms for computationally hard (biological) problems. Since many of these problems appear to "carry small parameters," we firmly believe that there will continue to be a strong interaction between parameterized complexity analysis and algorithmic bioinformatics. To make this as fruitful as possible, it is necessary to analyze real-world data in search for "hidden structure" which can be captured by suitable parameterizations. A subsequent parameterized complexity analysis can then determine which of these parameterizations yield field-parameter algorithms. This data-driven line of algorithmic research is still underdeveloped and should receive increased attention in future research. Moreover, in order to obtain the practically most useful algorithms, it may often be good to combine fixed-parameter algorithms (particularly, data reduction and kernelization) with general-purpose tools for solving computationally hard problems, including SAT solving and integer linear programming. This certainly will need a lot of experimentation going far beyond purely theoretical algorithm design.

10 Notes

1. To show that a problem is unlikely to be fixed-parameter tractable, the concept of W[1]-hardness was developed. It is widely assumed that a W[1]-hard problem cannot have a fixed-parameter algorithm (W[t]-hardness, $t \geq 2$ has the same implication). For example, the Clique problem to find

a clique (complete subgraph) in an undirected graph is W[1]-hard with respect to the parameter "number of vertices in the clique". To show that a problem is W[1]-hard, a *parameterized reduction* from a known W[1]-hard problem can be used (see e.,g. [44, 55]).

2. There exist suitable data reduction rules when it is of interest to enumerate *all* minimal vertex covers of a given graph. For example, Damaschke [47] suggests the notion of a *full kernel* that contains all minimal solutions in a compressed form and thereby allows enumeration of them.

3. One technique to show that a polynomial kernel is unlikely is called *composition* [54, 95]. A composition is an algorithm that combines the inputs of many instances of a problem into one "equivalent" instance. For 2-Club, the composition is to take the disjoint union of the input graphs of the instances: Any solution to such a combined instance has to live completely inside one of its connected components, which are completely contained in one of the original input instances. Thus, the combined instance has a solution if and only if at least one of the input instances has one. The existence of a composition and a polynomial kernelization leads to an implausible complexity-theoretic collapse. Thus, it is widely assumed that there is no polynomial problem kernel for problems with a composition [54, 95].

Acknowledgments. This is a completely revised, updated, and significantly expanded version of the previous book chapter "Developing Fixed-Parameter Algorithms to Solve Combinatorially Explosive Biological Problems" authored by Hüffner, Niedermeier, and Wernicke.

Falk Hüffner was supported by the Deutsche Forschungsgemeinschaft (DFG), project ALEPH (HU 2139/1).

References

[1] Abu-Khzam, F. N., Collins, R. L., Fellows, M. R., Langston, M. A., Suters, W. H., & Symons, C. T. (2004). Kernelization algorithms for the vertex cover problem: Theory and experiments. In *Proc. 6th Workshop on Algorithm Engineering and Experiments (ALENEX '04)*, (pp. 62–69). SIAM.

[2] Abu-Khzam, F. N., Daudjee, K., Mouawad, A. E., & Nishimura, N. (2013). An easy-to-use scalable framework for parallel recursive backtracking. Technical Report arXiv:1312.7626, arXiv.

[3] Abu-Khzam, F. N., Langston, M. A., Shanbhag, P., & Symons, C. T. (2006). Scalable parallel algorithms for FPT problems. *Algorithmica*, *45*(3), 269–284.

[4] Alber, J., Dorn, F., & Niedermeier, R. (2005). Empirical evaluation of a tree decomposition based algorithm for vertex cover on planar graphs. *Discrete Applied Mathematics*, *145*(2), 219–231.

[5] Alon, N., Dao, P., Hajirasouliha, I., Hormozdiari, F., & Sahinalp, S. C. (2008). Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, *24*(13), i241–i249.

[6] Alon, N., Yuster, R., & Zwick, U. (1995). Color-coding. *Journal of the ACM*, *42*(4), 844–856.

[7] Althaus, E., Klau, G. W., Kohlbacher, O., Lenhof, H., & Reinert, K. (2009). Integer linear programming in computational biology. In *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *LNCS*, (pp. 199–218). Springer.

[8] Atias, N. & Sharan, R. (2012). Comparative analysis of protein networks: hard problems, practical solutions. *Communications of the ACM*, *55*(5), 88–97.

[9] Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti-Spaccamela, A., & Protasi, M. (1999). *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*. Springer.

[10] Becker, A., Geiger, D., & Schäffer, A. (1998). Automatic selection of loop breakers for genetic linkage analysis. *Human Genetics*, *48*(1), 49–60.

[11] Berger, B., Singht, R., & Xu, J. (2008). Graph algorithms for biological systems analysis. In *Proc. 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '08)*, (pp. 142–151). SIAM.

[12] Betzler, N., Niedermeier, R., & Uhlmann, J. (2006). Tree decompositions of graphs: Saving memory in dynamic programming. *Discrete Optimization*, *3*(3), 220–229.

[13] Betzler, N., van Bevern, R., Fellows, M. R., Komusiewicz, C., & Niedermeier, R. (2011). Parameterized algorithmics for finding connected motifs in biological networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, *8*(5), 1296–1308.

[14] van Bevern, R. (2014). *Fixed-Parameter Linear-Time Algorithms for NP-hard Graph and Hypergraph Problems Arising in Industrial Applications*. PhD thesis, TU Berlin.

[15] Biere, A., Heule, M., van Maaren, H., & Walsh, T. (Eds.). (2009). *Handbook of Satisfiability*. IOS Press.

[16] Bixby, R. E. (2002). Solving real-world linear programs: A decade and more of progress. *Operations Research*, *50*, 3–15.

[17] Björklund, A., Kaski, P., & Kowalik, Ł. (2013). Probably optimal graph motifs. In *Proc. 30th International Symposium on Theoretical Aspects of Computer Science (STACS '13)*, volume 20 of *LIPIcs*, (pp. 20–31). Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[18] Björklund, A., Kaski, P., & Kowalik, Ł. (2014). Fast witness extraction using a decision oracle. In *Proc. 22th Annual European Symposium on Algorithms (ESA '14)*, volume 8737 of *LNCS*, (pp. 149–160). Springer.

[19] Böckenhauer, H.-J. & Bongartz, D. (2007). *Algorithmic Aspects of Bioinformatics*. Springer.

[20] Böcker, S. (2012). A golden ratio parameterized algorithm for cluster editing. *Journal of Discrete Algorithms*, *16*, 79–89.

[21] Böcker, S. & Baumbach, J. (2013). Cluster editing. In *Proc. 9th Conference on Computability in Europe (CiE '13)*, volume 7921 of *LNCS*, (pp. 33–44). Springer.

[22] Böcker, S., Briesemeister, S., Bui, Q. B. A., & Truß, A. (2009). Going weighted: Parameterized algorithms for cluster editing. *Theoretical Computer Science*, *410*(52), 5467–5480.

[23] Böcker, S., Briesemeister, S., & Klau, G. W. (2011). Exact algorithms for cluster editing: Evaluation and experiments. *Algorithmica*, *60*(2), 316–334.

[24] Böcker, S., Bui, Q. B. A., & Truß, A. (2011). Computing bond orders in molecule graphs. *Theoretical Computer Science*, *412*(12–14), 1184–1195.

[25] Böcker, S. & Damaschke, P. (2012). A note on the parameterized complexity of unordered maximum tree orientation. *Discrete Applied Mathematics*, *160*(10-11), 1634–1638.

[26] Böcker, S., Hüffner, F., Truss, A., & Wahlström, M. (2009). A faster fixed-parameter approach to drawing binary tanglegrams. In *Proc. 4th International Workshop on Parameterized and Exact Computation (IWPEC '09)*, volume 5917 of *LNCS*, (pp. 38–49). Springer.

[27] Böcker, S., Kehr, B., & Rasche, F. (2011). Determination of glycan structure from tandem mass spectra. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, *8*(4), 976–986.

[28] Böcker, S. & Rasche, F. (2008). Towards *de novo* identification of metabolites by analyzing tandem mass spectra. *Bioinformatics*, *24*(16), 49–55.

[29] Bodlaender, H. L. (1998). A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, *209*, 1–45.

[30] Bodlaender, H. L., Fomin, F. V., Koster, A. M. C. A., Kratsch, D., & Thilikos, D. M. (2012). On exact algorithms for treewidth. *ACM Transactions on Algorithms*, *9*(1), 12:1–12:23.

[31] Bodlaender, H. L. & Koster, A. M. C. A. (2008). Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, *51*(3), 255–269.

[32] Bodlaender, H. L. & Koster, A. M. C. A. (2010). Treewidth computations I: Upper bounds. *Information and Computation*, *208*(3), 259–275.

[33] Bonizzoni, P., Vedova, G. D., Dondi, R., & Pirola, Y. (2010). Variants of constrained longest common subsequence. *Information Processing Letters*, *110*(20), 877–881.

[34] Bruckner, S., Hüffner, F., Karp, R. M., Shamir, R., & Sharan, R. (2010). Topology-free querying of protein interaction networks. *Journal of Computational Biology*, *17*(3), 237–252.

[35] Bulteau, L., Fertin, G., Komusiewicz, C., & Rusu, I. (2013). A fixed-parameter algorithm for minimum common string partition with few duplications. In *Proc. 13th International Workshop on Algorithms in Bioinformatics (WABI '13)*, volume 8126 of *LNCS*, (pp. 244–258). Springer.

[36] Bulteau, L., Hüffner, F., Komusiewicz, C., & Niedermeier, R. (2014). Multivariate algorithmics for NP-hard string problems. *Bulletin of the EATCS*, *114*, 31–73.

[37] Cai, L., Chan, S. M., & Chan, S. O. (2006). Random separation: A new method for solving fixed-cardinality optimization problems. In *Proc. 2nd International Workshop on Parameterized and Exact Computation (IWPEC '06)*, volume 4169 of *LNCS*, (pp. 239–250). Springer.

[38] Cai, L., Chen, J., Downey, R. G., & Fellows, M. R. (1997). Advice classes of parameterized tractability. *Annals of Pure and Applied Logic*, *84*(1), 119–138.

[39] Canzar, S., El-Kebir, M., Pool, R., Elbassioni, K. M., Mark, A. E., Geerke, D. P., Stougie, L., & Klau, G. W. (2013). Charge group partitioning in biomolecular simulation. *Journal of Computational Biology*, *20*(3), 188–198.

[40] Cao, Y. & Chen, J. (2012). Cluster editing: Kernelization based on edge cuts. *Algorithmica*, *64*(1), 152–169.

[41] Cheetham, J., Dehne, F. K. H. A., Rau-Chaplin, A., Stege, U., & Taillon, P. J. (2003). Solving large FPT problems on coarse-grained parallel machines. *Journal of Computer and System Sciences*, *67*(4), 691–706.

[42] Chen, J., Kanj, I. A., & Xia, G. (2010). Improved upper bounds for vertex cover. *Theoretical Computer Science*, *411*(40–42), 3736–3756.

[43] Chen, J., Liu, Y., Lu, S., O'Sullivan, B., & Razgon, I. (2008). A fixed-parameter algorithm for the directed feedback vertex set problem. *Journal of the ACM*, *55*(5).

[44] Chen, J. & Meng, J. (2008). On parameterized intractability: Hardness and completeness. *The Computer Journal*, *51*(1), 39–59.

[45] Chesler, E. J., Lu, L., Shou, S., Qu, Y., Gu, J., Wang, J., Hsu, H. C., Mountz, J. D., Baldwin, N. E., Langston, M. A., Threadgill, D. W., Manly, K. F., & Williams, R. W. (2005). Complex trait analysis of gene expression uncovers polygenic and pleiotropic networks that modulate nervous system function. *Nature Genetics*, *37*, 233–242.

[46] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[47] Damaschke, P. (2006). Parameterized enumeration, transversals, and imperfect phylogeny reconstruction. *Theoretical Computer Science*, *351*(3), 337–350.

[48] Dehne, F. K. H. A., Fellows, M. R., Langston, M. A., Rosamond, F. A., & Stevens, K. (2007). An $O(2^{O(k)}n^3)$ FPT algorithm for the undirected feedback vertex set problem. *Theory of Computing Systems*, *41*(3), 479–492.

[49] Diestel, R. (2010). *Graph Theory* (4th ed.)., volume 173 of *Graduate Texts in Mathematics*. Springer.

[50] Doan, D. D. & Evans, P. A. (2011). An FPT haplotyping algorithm on pedigrees with a small number of sites. *Algorithms for Molecular Biology*, *6*, 8.

[51] Dondi, R. & El-Mabrouk, N. (2013). Aligning and labeling genomes under the duplication-loss model. In *Proc. 9th Conference on Computability in Europe (CiE '13)*, volume 7921 of *LNCS*, (pp. 97–107). Springer.

[52] Donmez, N. & Brudno, M. (2013). SCARPA: scaffolding reads with practical algorithms. *Bioinformatics*, *29*(4), 428–434.

[53] Dost, B., Shlomi, T., Gupta, N., Ruppin, E., Bafna, V., & Sharan, R. (2008). QNet: A tool for querying protein interaction networks. *Journal of Computational Biology*, *15*(7), 913–925.

[54] Downey, R. G. & Fellows, M. R. (2013). *Fundamentals of Parameterized Complexity*. Texts in Computer Science. Springer.

[55] Downey, R. G. & Thilikos, D. M. (2011). Confronting intractability via parameters. *Computer Science Review*, *5*(4), 279–317.

[56] Fafianie, S., Bodlaender, H. L., & Nederlof, J. (2015). Speeding up dynamic programming with representative sets: An experimental evaluation of algorithms for steiner tree on tree decompositions. *Algorithmica*, *71*(3), 636–660.

[57] Fellows, M. R., Gramm, J., & Niedermeier, R. (2006). On the parameterized intractability of motif search problems. *Combinatorica*, *26*(2), 141–167.

[58] Felner, A., Korf, R. E., & Hanan, S. (2004). Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, *21*, 1–39.

[59] Fleischer, R., Wu, X., & Yuan, L. (2009). Experimental study of FPT algorithms for the directed feedback vertex set problem. In *Proc. 17th Annual European Symposium on Algorithms (ESA '09)*, volume 5757 of *LNCS*, (pp. 611–622). Springer.

[60] Flum, J. & Grohe, M. (2006). *Parameterized Complexity Theory*. Springer.

[61] Fomin, F. V. & Kratsch, D. (2010). *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. Springer.

[62] Garey, M. R. & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman.

[63] Gottlob, G., Pichler, R., & Wei, F. (2010). Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artificial Intelligence*, *174*(1), 105–132.

[64] Gramm, J. (2003). *Fixed-Parameter Algorithms for the Consensus Analysis of Genomic Sequences*. PhD thesis, WSI für Informatik, Universität Tübingen, Germany.

[65] Gramm, J., Guo, J., Hüffner, F., & Niedermeier, R. (2004). Automated generation of search tree algorithms for hard graph modification problems. *Algorithmica*, *39*, 321–347.

[66] Gramm, J., Guo, J., Hüffner, F., & Niedermeier, R. (2005). Graph-modeled data clustering: Exact algorithms for clique generation. *Theory of Computing Systems*, *38*(4), 373–392.

[67] Gramm, J., Guo, J., Hüffner, F., & Niedermeier, R. (2008). Data reduction and exact algorithms for clique cover. *ACM Journal of Experimental Algorithmics*, *13*, 2.2:1–2.2:15.

[68] Gramm, J., Guo, J., & Niedermeier, R. (2006). Parameterized intractability of distinguishing substring selection. *Theory of Computing Systems*, *39*(4), 545–560.

[69] Gramm, J. & Niedermeier, R. (2003). A fixed-parameter algorithm for minimum quartet inconsistency. *Journal of Computer and System Sciences*, *67*(4), 723–741.

[70] Gramm, J., Niedermeier, R., & Rossmanith, P. (2003). Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, *37*(1), 25–42.

[71] Groër, C., Sullivan, B. D., & Weerapurage, D. (2012). INDDGO: Integrated network decomposition & dynamic programming for graph optimization. Technical Report ORNL/TM-2012/176, Oak Ridge National Laboratory.

[72] Guillemot, S. (2011). Parameterized complexity and approximability of the longest compatible sequence problem. *Discrete Optimization*, *8*(1), 50–60.

[73] Guillemot, S. & Mnich, M. (2013). Kernel and fast algorithm for dense triplet inconsistency. *Theoretical Computer Science*, *494*, 134–143.

[74] Guo, J., Gramm, J., Hüffner, F., Niedermeier, R., & Wernicke, S. (2006). Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *Journal of Computer and System Sciences*, *72*(8), 1386–1396.

[75] Guo, J., Moser, H., & Niedermeier, R. (2009). Iterative compression for exactly solving NP-hard minimization problems. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, (pp. 65–80). Springer.

[76] Guo, J. & Niedermeier, R. (2007). Invitation to data reduction and problem kernelization. *ACM SIGACT News*, *38*(1), 31–45.

[77] Hartung, S. & Hoos, H. H. (2015). Programming by optimisation meets parameterised algorithmics: A case study for cluster editing. In *Proc. 9th Learning and Intelligent OptimizatioN Conference (LION'15)*, volume 8994 of *LNCS*, (pp. 43–58). Springer.

[78] Hartung, S., Komusiewicz, C., & Nichterlein, A. (2015). Parameterized algorithmics and computational experiments for finding 2-clubs. *Journal of Graph Algorithms and Applications*, *19*(1), 155–190.

[79] Hlinený, P., Oum, S., Seese, D., & Gottlob, G. (2008). Width parameters beyond tree-width and their applications. *The Computer Journal*, *51*(3), 326–362.

[80] Hoos, H. H. (2012). Programming by optimization. *Communications of the ACM*, *55*(2), 70–80.

[81] Hüffner, F. (2009). Algorithm engineering for optimal graph bipartization. *Journal of Graph Algorithms and Applications*, *13*(2), 77–98.

[82] Hüffner, F., Betzler, N., & Niedermeier, R. (2010). Separator-based data reduction for signed graph balancing. *Journal of Combinatorial Optimization*, *20*(4), 335–360.

[83] Hüffner, F., Komusiewicz, C., Moser, H., & Niedermeier, R. (2010). Fixed-parameter algorithms for cluster vertex deletion. *Theory of Computing Systems*, *47*(1), 196–217.

[84] Hüffner, F., Niedermeier, R., & Wernicke, S. (2008). Techniques for practical fixed-parameter algorithms. *The Computer Journal*, *51*(1), 7–25.

[85] Hüffner, F., Wernicke, S., & Zichner, T. (2008). Algorithm engineering for color-coding with applications to signaling pathway detection. *Algorithmica*, *52*(2), 114–132.

[86] Kask, K., Dechter, R., Larrosa, J., & Dechter, A. (2005). Unifying tree decompositions for reasoning in graphical models. *Artificial Intelligence*, *166*(1-2), 165–193.

[87] Kleinberg, J. M. & Tardos, É. (2006). *Algorithm Design*. Addison-Wesley.

[88] Kneis, J., Langer, A., & Rossmanith, P. (2011). Courcelle's theorem—a game-theoretic approach. *Discrete Optimization*, *8*(4), 568–594.

[89] Kneis, J., Mölle, D., Richter, S., & Rossmanith, P. (2006). Divide-and-color. In *Proc. 32nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG '06)*, volume 4271 of *LNCS*, (pp. 58–67). Springer.

[90] Kociumaka, T. & Pilipczuk, M. (2014). Faster deterministic feedback vertex set. *Information Processing Letters*, *114*(10), 556–560.

[91] Komusiewicz, C. & Niedermeier, R. (2012). New races in parameterized algorithmics. In *Proc. 37th International Symposium on Mathematical Foundations of Computer Science (MFCS '12)*, volume 7464 of *LNCS*, (pp. 19–30). Springer.

[92] Komusiewicz, C. & Sorge, M. (2015). An algorithmic framework for fixed-cardinality optimization in sparse graphs applied to dense subgraph problems. *Discrete Applied Mathematics*. To appear.

[93] Koutis, I. (2008). Faster algebraic algorithms for path and packing problems. In *Proc. 35th International Colloquium on Automata, Languages and Programming (ICALP '08)*, volume 5125 of *LNCS*, (pp. 575–586). Springer.

[94] Koutis, I. & Williams, R. (2009). Limits and applications of group algebras for parameterized problems. In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP '09)*, volume 5555 of *LNCS*, (pp. 653–664). Springer.

[95] Kratsch, S. (2014). Recent developments in kernelization: A survey. *Bulletin of the EATCS*, *113*, 58–97.

[96] Krithika, R. & Narayanaswamy, N. S. (2013). Another disjoint compression algorithm for odd cycle transversal. *Information Processing Letters*, *113*(22-24), 849–851.

[97] Langer, A., Reidl, F., Rossmanith, P., & Sikdar, S. (2012). Evaluation of an MSO-solver. In *Proc. 14th Workshop on Algorithm Engineering and Experiments (ALENEX '12)*, (pp. 55–63). SIAM.

[98] Langer, A., Reidl, F., Rossmanith, P., & Sikdar, S. (2014). Practical algorithms for MSO model-checking on tree-decomposable graphs. *Computer Science Review*, *13-14*, 39–74.

[99] Liberti, L., Lavor, C., & Mucherino, A. (2013). The discretizable molecular distance geometry problem seems easier on proteins. In *Distance Geometry: Theory, Methods, and Applications* (pp. 47–60). Springer.

[100] Linz, S., Semple, C., & Stadler, T. (2010). Analyzing and reconstructing reticulation networks under timing constraints. *Journal of Mathematical Biology*, *61*(5), 715–737.

[101] Liu, C., Song, Y., Yan, B., Xu, Y., & Cai, L. (2006). Fast de novo peptide sequencing and spectral alignment via tree decomposition. In *Proc. 11th Pacific Symposium on Biocomputing (PSB '06)*, (pp. 255–266).

[102] Lokshtanov, D., Marx, D., & Saurabh, S. (2011). Known algorithms on graphs on bounded treewidth are probably optimal. In *Proc. 22nd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '11)*, (pp. 777–789). SIAM.

[103] Lokshtanov, D., Narayanaswamy, N. S., Raman, V., Ramanujan, M. S., & Saurabh, S. (2014). Faster parameterized algorithms using linear programming. *ACM Transactions on Algorithms*, *11*(2), 15:1–15:31.

[104] Marx, D. (2008). Closest substring problems with small distances. *SIAM Journal on Computing*, *38*(4), 1382–1410.

[105] Michalewicz, Z. & Fogel, D. B. (2004). *How to Solve It: Modern Heuristics* (2nd ed.). Springer.

[106] Miranda, M., Lynce, I., & Manquinho, V. M. (2014). Inferring phylogenetic trees using pseudo-boolean optimization. *AI Communications*, *27*(3), 229–243.

[107] Moore, C. & Mertens, S. (2011). *The Nature of Computation*. Oxford University Press.

[108] Moser, H., Niedermeier, R., & Sorge, M. (2012). Exact combinatorial algorithms and experiments for finding maximum k-plexes. *Journal of Combinatorial Optimization*, *24*(3), 347–373.

[109] Nemhauser, G. L. & Trotter, L. E. (1975). Vertex packings: Structural properties and algorithms. *Mathematical Programming*, *8*(1), 232–248.

[110] Niedermeier, R. (2006). *Invitation to Fixed-Parameter Algorithms*. Oxford University Press.

[111] Niedermeier, R. (2010). Reflections on multivariate algorithmics and problem parameterization. In *Proc. 27th International Symposium on Theoretical Aspects of Computer Science (STACS '10)*, volume 5 of *LIPIcs*, (pp. 17–32). Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[112] Niedermeier, R. & Rossmanith, P. (2000). A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, *73*, 125–129.

[113] Panconesi, A. & Sozio, M. (2004). Fast hare: A fast heuristic for single individual SNP haplotype reconstruction. In *Proc. 4th Workshop on Algorithms in Bioinformatics (WABI '04)*, volume 3240 of *LNCS*, (pp. 266–277). Springer.

[114] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[115] Papadimitriou, C. H. (1997). NP-completeness: A retrospective. In *Proc. 24th International Colloquium on Automata, Languages and Programming (ICALP '97)*, volume 1256 of *LNCS*, (pp. 2–6). Springer.

[116] Pasupuleti, S. (2008). Detection of protein complexes in protein interaction networks using n-Clubs. In *Proc. 6th European Conference on Evolutionary Computation, Machine Learning and Data Mining in Bioinformatics (EvoBIO '06)*, volume 4973 of *LNCS*, (pp. 153–164). Springer.

[117] Patterson, M., Marschall, T., Pisanti, N., van Iersel, L., Stougie, L., Klau, G. W., & Schönhuth, A. (2015). WhatsHap: Weighted haplotype assembly for future-generation sequencing reads. *Journal of Computational Biology*, *22*(6), 498–509.

[118] Peiselt, T. (2007). An iterative compression algorithm for vertex cover. Studienarbeit, Institut für Informatik, Friedrich-Schiller-Universität Jena.

[119] Reed, B., Smith, K., & Vetta, A. (2004). Finding odd cycle transversals. *Operations Research Letters*, *32*(4), 299–301.

[120] Schäfer, A., Komusiewicz, C., Moser, H., & Niedermeier, R. (2012). Parameterized computational complexity of finding small-diameter subgraphs. *Optimization Letters*, *6*(5), 883–891.

[121] Scott, J., Ideker, T., Karp, R. M., & Sharan, R. (2006). Efficient algorithms for detecting signaling pathways in protein interaction networks. *Journal of Computational Biology*, *13*(2), 133–144.

[122] Shlomi, T., Segal, D., Ruppin, E., & Sharan, R. (2006). QPath: a method for querying pathways in a protein–protein interaction network. *BMC Bioinformatics*, *7*, 199.

[123] Skiena, S. S. (2008). *The Algorithm Design Manual* (2nd ed.). Springer.

[124] Song, Y., Liu, C., Malmberg, R. L., Pan, F., & Cai, L. (2005). Tree decomposition based fast search of RNA structures including pseudoknots in genomes. In *Proc. 4th International IEEE Computer Society Computational Systems Bioinformatics Conference (CSB 2005)*, (pp. 223–234). IEEE Computer Society.

[125] Stojanovic, N., Florea, L., Riemer, C., Gumucio, D., Slightom, J., Goodman, M., Miller, W., & Hardison, R. (1999). Comparison of five methods for finding conserved sequences in multiple alignments of gene regulatory regions. *Nucleic Acids Research*, *27*(19), 3899–3910.

[126] Stolzer, M., Lai, H., Xu, M., Sathaye, D., Vernot, B., & Durand, D. (2012). Inferring duplications, losses, transfers and incomplete lineage sorting with nonbinary species trees. *Bioinformatics*, *28*(18), 409–415.

[127] Vardi, M. Y. (2014). Boolean satisfiability: theory and engineering. *Communications of the ACM*, *57*(3), 5.

[128] Vazirani, V. V. (2001). *Approximation Algorithms*. Springer.

[129] West, D. B. (2000). *Introduction to Graph Theory* (2 ed.). Prentice Hall.

[130] Whidden, C., Beiko, R. G., & Zeh, N. (2010). Fast FPT algorithms for computing rooted agreement forests: Theory and experiments. In *Proc. 9th International Symposium on Experimental Algorithms (SEA '10)*, volume 6049 of *LNCS*, (pp. 141–153). Springer.

[131] Williamson, D. P. & Shmoys, D. B. (2011). *The Design of Approximation Algorithms*. Cambridge University Press.

[132] Wittkop, T., Emig, D., Lange, S., Rahmann, S., Albrecht, M., Morris, J. H., Böcker, S., Stoye, J., & Baumbach, J. (2010). Partitioning biological data with transitivity clustering. *Nature Methods*, *7*(6), 419–420.

[133] Wittkop, T., Emig, D., Truss, A., Albrecht, M., Böcker, S., & Baumbach, J. (2011). Comprehensive cluster analysis with transitivity clustering. *Nature Protocols*, *6*(3), 285–295.

[134] Wu, G., You, J.-H., & Lin, G. (2005). A lookahead branch-and-bound algorithm for the maximum quartet consistency problem. In *Proc. 5th Workshop on Algorithms in Bioinformatics (WABI '05)*, volume 3692 of *LNCS*, (pp. 65–76). Springer.

[135] Zhao, J., Malmberg, R. L., & Cai, L. (2008). Rapid ab initio prediction of RNA pseudoknots via graph tree decomposition. *Journal of Mathematical Biology*, *56*(1–2), 145–159.