Developing Fixed-Parameter Algorithms to Solve Combinatorially Explosive Biological Problems

Falk Hüffner Rolf Niedermeier Sebastian Wernicke

Institut für Informatik, Friedrich-Schiller-Universität Jena
Ernst-Abbe-Platz 2, D-07743 Jena, Germany
{hueffner,niedermr,wernicke}@minet.uni-jena.de
http://theinf1.informatik.uni-jena.de

Abstract

Fixed-parameter algorithms can efficiently find optimal solutions to some computationally hard (NP-hard) problems. We survey five main practical techniques to develop such algorithms. Each technique is circumstantiated by case studies of applications to biological problems. We also present other known bioinformatics-related applications and give pointers to experimental results.
Key Words: Computationally hard problems; combinatorial explosions; 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. A typical example is the following: Given a series of n experiments of which some pairs have conflicting results (that is, at least one must have been faulty), identify a minimum-size subset of experiments to eliminate so that no conflict remains. This problem is notoriously difficult to solve. For this and many other problems, the root of these difficulties can be identified as their *NP-hardness*, which implies a combinatorial explosion in the solution space that apparently cannot be easily avoided [27]. Thus, whenever a problem is proven to be NP-hard, it is common to employ heuristic algorithms, approximation algorithms, or attempt to sidestep the problem whenever large instances need to be solved. All the concrete case studies we provide in this survey deal with NP-hard problems.

Often, however, it is not only the size of an instance that makes a problem hard to solve, but rather its structure. The concept of *fixed-parameter tractability* (FPT) reflects this observation and renders it more precise by measuring structural hardness by a so-called *parameter*, which is typically a nonnegative integer variable denoted k. This generalizes the concept of "easy special cases" that are known for virtually all NP-hard problems: Whenever the parameter k

turns out to be small, a fixed-parameter algorithm is going to be provably efficient while guaranteeing the optimality of the solution obtained. For instance, our fault identification problem can be solved quickly whenever the number of faulty experiments is small—an assumption easy to make in practice, since otherwise the results would not be worth much anyway.

A particular appeal of FPT is that the parameter can be chosen from basically boundless possibilities. For instance, we might in our example choose the maximum number of conflicts for a single experiment or the size of the largest group of pairwise conflicting experiments to be the parameter. This makes FPT a many-pronged attack that can be adapted to different practical applications of one problem. Note, however, that not all parameters need to lead to efficient algorithms; in fact, FPT provides tools to classify parameters as "not helpful," meaning that we cannot expect efficient solvability even when the parameter is small.

Fixed-parameter algorithms have by now facilitated many success stories in bioinformatics. Several techniques have emerged as being applicable to large classes of problems. In the main part of this work, we present five of these techniques, namely kernelization (Section 2), depth-bounded search trees (Section 3), tree decompositions of graphs (Section 4), color-coding (Section 5), and iterative compression (Section 6). We start each section by giving an overview of basic concepts and ideas, followed by one to three detailed case studies concerning practically relevant bioinformatics problems. Finally, we survey known applications, implementations, and experimental results, thereby highlighting the strengths and fields of applicability of each technique.

Before discussing the main techniques, we continue with a crash course of computational complexity theory and a formal definition for the concept of fixed-parameter tractability. Furthermore, some terms from graph theory are introduced, and we present our running example problem Vertex Cover.

1.1 Computational Complexity Theory

A core topic of computational complexity theory is the evaluation and comparison of different algorithms for a problem [43]. Since most algorithms are designed to work with inputs of arbitrary length, the efficiency (or *complexity*) of an algorithm is not stated just for a single input (*instance*) or a collection of inputs, but as a function that relates the input length n to the number of steps that are required to execute the algorithm. Since instances of the same size might take different amounts of time, the *worst-case* runtime is considered. This figure is given in an asymptotic sense; the standard way for this being the *big-O notation*: we say that $f(n) = O(g(n))$ when $f(n)/g(n)$ is upper-bounded by a positive constant in the limit for large n [17].

Determining the computational complexity of problems (meaning the best possible asymptotic runtime of an algorithm for them) is a key issue in theoretical computer science. Of central importance herein is to distinguish between problems that can be solved efficiently and those that presumably cannot. To this end, theoretical computer scientists have coined the notions of *polynomial-*

time solvable on the one hand and *NP-hard* on the other [27]. In this sense, polynomial-time solvability has become a synonym for efficient solvability. This means that for a size-n input instance of a problem, an optimal solution can be computed in $O(n^c)$ time, where c is some positive constant. By way of contrast, the (unproven, yet widely believed) working hypothesis of theoretical computer science is that NP-hard problems cannot be solved in $O(n^c)$ time for any constant c. More specifically, typical runtimes for NP-hard problems are of the form $O(c^n)$ for some constant $c > 1$, or even worse; that is, we have an exponential growth of the number of computation steps.

As there are thousands of practically important NP-hard optimization problems, and the number is continuously growing [44], 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 [5, 51]. Another commonly employed strategy is that of heuristics, where one gives up any provable performance guarantees concerning runtime and/or solution quality by developing algorithms that "usually" behave well in "most" practical applications [40].

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 optimal solutions with useful runtime guarantees [22]. The core concept is formalized as follows.

Definition 1. *An instance of a parameterized problem 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, not on the input size $|I|$.*

For NP-hard problems, $f(k)$ will of course not be polynomial—since otherwise we would have an overall polynomial-time algorithm—but typically be exponential like 2^k. Clearly, fixed-parameter tractability captures the notion of "efficient for small parameter values": for any constant k, we obtain a polynomial-time algorithm. Moreover, the exponent of the polynomial must be independent of k, which means that the combinatorial explosion is *completely confined to the parameter.*

As an example, consider again the identification of k faulty experiments among n experiments. We can solve this problem in $O(2^n)$ time by trying all possible subsets. However, this is not feasible for $n > 40$. In contrast, a fixed-parameter algorithm with runtime $O(2^k \cdot n)$ exists, which allows to solve the problem even for $n = 1000$, if $k < 20$ (as will be discussed in Section 2.3, instances can even be solved for much larger values of k in practice by an extension of this approach).

Note that—as parameterized complexity theory points out—there are problems that are probably not fixed-parameter tractable.

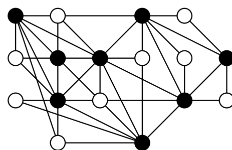

Figure 1: A graph with a size-8 vertex cover (cover vertices are marked black).

Two recent monographs are available on parameterized complexity, one focusing on theoretical foundations [26], and one focusing on techniques and algorithms [41], the latter also being the focus of our considerations here.

1.3 Graph Theory

Many of the problems we deal with in this work are from graph theory [21]. A graph $G = (V, E)$ is given by a set of *vertices* V and a set of *edges* E, where every edge $\{v, w\}$ is an undirected connection of two vertices v and w. Throughout this work, we use n to denote the number of vertices and m 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'.

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

Vertex Cover
Input: A graph $G = (V, E)$ and a nonnegative integer k.
Task: Find a subset of vertices $C \subseteq V$ with k or fewer 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 be considered the *Drosophila* of fixed-parameter research in that many initial discoveries that influenced the whole field originated from studies 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 (relatively) easy to cope with, shrinking it to those parts that form the “really hard” core of the problem. 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 useful in virtually any problem solving context, whether it be heuristic, approximative, or exact.

This section introduces the concept of *kernelizations*, that is, polynomial-time data reduction with guaranteed effectiveness. These are closely connected to and emerge within the FPT framework.

2.1 Basic Concepts

Today, 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 [10]. Obviously, many practitioners are aware of the concept of data reduction in general. The reason why they should also consider FPT in this context is that fixed-parameter theory provides a way to use data reduction rules not only in a heuristic way, but with guaranteed performance using so-called *kernelizations*. For a reduced instance, these guarantee an upper bound on its size that solely depends on the parameter value. To render a precise definition of this:

Definition 2 ([22, 41]). *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 $k' \leq k$ such that—independently of the size of I—the size of I' is guaranteed to only depend on some function in k. Furthermore, the new instance I' must have 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 problem kernel—they are exactly the same. Unfortunately, the runtime 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. Hence, the main use of this equivalence is to establish the fixed-parameter tractability or amenability to kernelization of a problem; it is also useful for showing that we need not search any further (e.g., if a problem is known to be fixed-parameter intractable, we do not need to look for a kernelization).

2.2 Case Study

In this section, we first illustrate the concept of kernelization by a simple example concerning the Vertex Cover problem. We then show how a generalization of this method leads to a very effective kernelization.

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
For degree-1 vertices, put their 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.[1]

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 of degree at least $k+1$, put this vertex 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 to v. This is not possible because the maximum allowed size of the cover is k.

After exhaustively performing Rules VC1 and VC2, no vertex in the remaining graph has a degree higher than k, meaning that 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 is to have 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.

Abstractly speaking, what we have done so far 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 *solely be expressed in terms of the parameter k*. Hence, considering Definition 2, we have found a kernelization for Vertex Cover.

[1] There exist suitable reduction rules when it is of interest to enumerate *all* vertex covers of a given graph, but these are beyond the scope of this work. For instance, Damaschke [18] suggests the notion of a *full kernel* that contains all solutions in a compressed form and thereby allows enumeration of them. This is an emerging field of research, and not many full kernels are known.

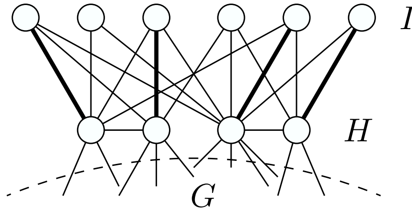

Figure 2: A graph G with a crown $I \cup H$. Note how the thick edges constitute a maximum matching of size $|H|$ in the bipartite graph induced by the edges between I and H.

2.2.2 Improving Effectiveness by Generalization: Crown Reductions

Besides the simple-to-achieve size-k^2 problem kernel for Vertex Cover we have just discussed, there are several more advanced kernelization techniques for Vertex Cover. The more advanced methods generally feature two important improvements: First, they do not require the parameter k to be stated explicitly beforehand (contrary to Rule VC2), that is, they are *parameter-independent.* Second, they improve the upper bounds on the kernel size to being linear in k. We explore one of these advanced methods in more detail here, namely the so-called *crown reduction* rules [1, 16].

Crown reduction rules are a prominent example for an advanced kernelization for Vertex Cover; they constitute a generalization of the elimination of degree-1 vertices we have seen in Rule VC1. A *crown* in a graph consists of an independent set I (that is, no two vertices in I are connected by an edge) and a set H containing all vertices adjacent to I. In order for $I \cup H$ to be a crown, there has to exist a size-$|H|$ maximum bipartite matching in the bipartite graph induced by the edges between I and H, that is, one in which every element of H is matched. (See, e.g., [17] for an introduction on bipartite matching and the involved algorithmics.) An example for a crown structure is given in Figure 2.

If there is a crown $I \cup H$ in the input graph G, then we need *at least* $|H|$ vertices to cover all edges in the crown. But since all edges in the crown can be covered by taking *at most* $|H|$ vertices into the cover (as I is an independent set), there is a minimum-size vertex cover for G that contains all the vertices in H and none of the vertices in I. We may thus delete any given crown $I \cup H$ from G, reducing k by $|H|$. In a sense, the degree-1 vertices we took care of in Rule VC1 are the most simple crowns.

Two issues remain to be dealt with, namely how to find crowns efficiently and giving an upper bound on the size of the problem kernel that can be obtained via crown reductions. It turns out that finding crowns can be achieved in polynomial time [16]. The size of the thus reduced instance is upper-bounded via the following theorem (a proof of which can be found in [1]):

Theorem 1. *A graph that is crown-free and has a vertex cover of size at most k can contain at most $3k$ vertices.* □

In this way, by generalizing Rule VC1—which by itself does not constitute a kernelization—we have obtained an efficient kernelization that does not require an explicitly stated value for the parameter k and yields a small kernel for Vertex Cover.[2] Even smaller kernels of size upper-bounded by $2k$ are attainable with other methods, for example using the so-called "Nemhauser–Trotter kernelization" (see [41] for details).

2.3 Applications and Implementations

Since many initial and influential discoveries concerning FPT were made from studies of Vertex Cover, it comes as no surprise that the experimental field is more advanced for this problem than for others from the realm of fixed-parameter tractability.

Abu-Khzam et al. [1] studied various kernelization methods for Vertex Cover and their practical performance both with respect to time as well as with respect to the resulting kernel size. For bioinformatics-related networks derived from protein databases and microarray data, they found that crown reductions turn out to be very fast to compute in practice and can be as effective as approaches with a better worst-case bound of $2k$ (such as the Nemhauser–Trotter reduction). Abu-Khzam et al. therefore recommend always using crown reduction as a general preprocessing step when solving Vertex Cover before attempting other, more costly, reduction methods.

Solving Vertex Cover is of relevance to many bioinformatics-related scenarios such as microarray data analysis [15] and the computation of multiple sequence alignments [13]. Besides solving instances of Vertex Cover, another important application of Vertex Cover kernelizations is that of searching maximum cliques (that is, maximum-size fully connected subgraphs) in a graph. Here, use is made of the fact that an n-vertex graph has a maximum clique of size $(n-k)$ if and only if its complement graph[3] has a size-k minimum vertex cover. Details and experimental results for this with applications to computational biology are given, e.g., by Abu-Khzam et al. [2]. State-of-the art implementations of fixed-parameter algorithms and kernelizations for Vertex Cover enable finding cliques and dense subgraphs consisting of 200 or more vertices (e.g., see [15]) in biological networks such as they appear in the analysis of microarray data.

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. [30] showed instances with a solution size of up to $k = 250$ to be solvable in practice. Finally, a further example for a clustering problem where kernelizations have proven to be quite useful in practice is the Cluster Editing problem that we discuss in more detail in Section 3.2.2 [20].

[2] Another example for a data reduction for Vertex Cover that is based on schemes that generalize to arbitrarily large graph substructures is given by Chen et al. [14].

[3] That is, the "edgewise inverse" graph that contains exactly those edges the original graph does not contain.

3 Depth-Bounded Search Trees

Once the data reductions we have 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 related to optimally solving 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 trees 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 [17, 48]). The main contribution of fixed-parameter theory to search tree algorithms is the consideration of search trees whose *depth is bounded from above by 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, there are even better search trees of size $O(1.28^k)$ possible). Since usually $k \ll n$, this can draw the problem into the zone of feasibility even for large graphs (as long as k is small).

Besides depth-bounding, fixed-parameter theory provides additional means to provably improve the speed of search tree exploration, particularly by interleaving this exploration with kernelizations, that is, data reduction is applied 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 Cover, the most simple such subset is any two vertices that are connected by an edge. By definition of the problem, one of these two vertices *must* 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 proceeds by picking an

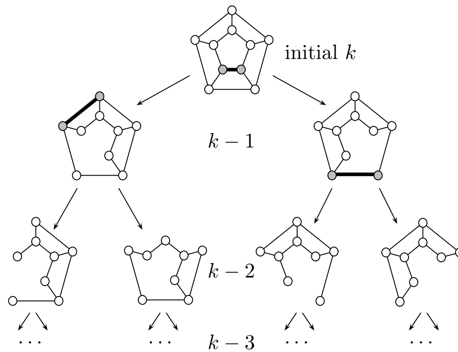

Figure 3: Simple search tree for finding a vertex cover of size at most k in a given graph. The size of the tree is upper-bounded by $O(2^k)$.

arbitrary edge $e = \{v, w\}$ and recursively searching for a vertex cover of size $k-1$ both in $G - v$ and $G - w$.[4] That is, 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 3, 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 upper-bounded by $O(2^k)$. Note how this size is independent of the size of the input instance and only depends on the value of the parameter k.

The currently "best" search trees for Vertex Cover are of worst-case size $O(1.28^k)$ [14] and mainly achieved by elaborate case distinctions. 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 The Cluster Editing Problem

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 the following clustering problem:

Cluster Editing
Input: A graph $G = (V, E)$ and a nonnegative integer k.
Task: Find whether we can modify G to consist of disjoint cliques (that is, fully connected components) by adding or deleting at most k edges.

[4] For a vertex $v \in V$, we define $G - v$ to be the graph G with both the vertex v and the edges incident to v removed.

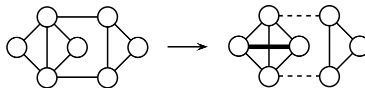

Figure 4: Illustration for the Cluster Editing problem: By removing two edges from and adding one edge to the graph on the left (that is, $k = 3$), we can obtain a graph that consists of two disjoint cliques.

An illustration for this problem is given in Figure 4. Cluster Editing has important applications in graph-modeled data clustering, e.g., to cluster microarray data: Here, the individual datapoints are represented as vertices. Edges connect two vertices if these have similar expression profiles. The underlying assumption is that the datapoints will form dense clusters in the constructed graph that are only sparsely connected to each other. Thus, by adding and removing only a few edges, we can reveal the underlying correlation structure in the form of disjoint cliques.

Similar to Vertex Cover, a search tree for Cluster Editing can be obtained by noting that the desired graph of disjoint cliques forbids a certain structure: If two vertices are connected by an edge, then their neighborhoods must be the same. Hence, whenever we encounter two connected vertices u and v in the input graph G that are connected by an edge and where one vertex, say u, has a neighbor w that is not connected to v, we are compelled to do one of three things: Either remove the edge $\{u, v\}$, or connect v with w, or remove the edge $\{u, w\}$. Note that each such modification counts with respect to the parameter k. Therefore, exhaustively branching into three cases for at most k forbidden substructures, we obtain a search tree of size $O(3^k)$ to solve Cluster Editing. The currently best branching scheme has a search-tree size of $O(1.92^k)$ [29]; interestingly, it was derived by using computer-aided algorithm design. Experimental results for Cluster Editing are reported in [20].

3.2.3 The Center String Problem

The Center String problem is also known as Consensus String or Closest String.

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

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 k of input strings. Both parameters are reasonably small in various

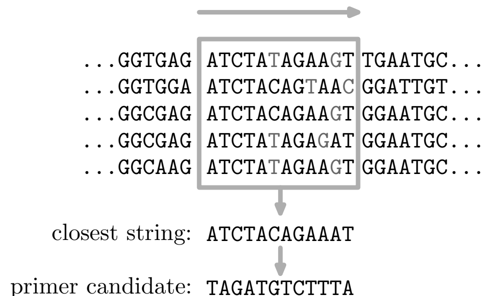

Figure 5: Illustration to show how DNA primer design can be achieved by solving Center String instances on length-ℓ windows of aligned DNA sequences. (Note that the primer candidate is not the center string sought after but its nucleotide-wise complement.)

applications; we refer to Gramm et al. [33] for more details. Here, we focus on the parameter d.

One application scenario where this problem appears is 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. [50] 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 Center String problem for each window position. Figure 5 illustrates this (see [28] for details).

In the remainder of this case study, we sketch a fixed-parameter search tree algorithm for Center String due to Gramm et al. [33], 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 all but 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, and we are not aware of a better one.

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 Center 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, Center String can be solved by exploring a search tree of size $O((d+1)^d)$ [33]. Combining data reduction with this search tree, we arrive at the following.

Theorem 2. 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, say, $d = 15$. However, two things are to be noted in this respect: First, for one of the main applications of Center String, primer design, d is very small (most 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 an Integer Linear Programming formulation of Center String when the input consists of many strings and ℓ is small [33].

Unfortunately, many variants of Center 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 from a fixed-parameter point of view [23, 31, 39].

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 that ten thousand vertices in some dense graphs of biological origin [2]. 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. Cheetham et al. [13] were the first to practically demonstrate this with their parallel Vertex Cover solver; they achieve a near-optimum (i.e., linear with the number of processors employed) speedup on multiprocessor systems, solving instances with $k \geq 400$ in mere hours. Recent research even indicates that in some cases, parallelizing may yield a super-linear speedup because the branches that lead to a solution are explored earlier than in a sequential setting [2].

Besides in fixed-parameter theory, 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 FPT community in their works[5] is that they typically cannot improve the asymptotic runtime. Still, the speedups obtained in practice can be quite pronounced, as demonstrated for VERTEX COVER [24].

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. A recent 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 and Niedermeier [32]. Their insight was recently used by Wu et al. [52] to develop a (non-parameterized) faster 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 Tree Decompositions of Graphs

Many NP-hard graph problems become computationally feasible when they are restricted to graphs without cycles, that is, trees or collections of trees (forests). Trees, however, form a very limited class of graphs that often do not suffice as a model in real-life applications. Hence, as a compromise between general graphs and trees, one might want to look at "tree-like" graphs. This likeness is formalized by the concept of *tree decompositions* of graphs. Introduced by Robertson and Seymour about twenty years ago, tree decompositions nowadays play a central role in algorithmic graph theory [21]. In this section, we survey some important aspects of tree decompositions and their algorithmic use with respect to computational biology and FPT.

4.1 Basic Concepts

There is a very helpful and intuitively appealing characterization of tree decompositions in terms of a *robber–cop game* in a graph [11]: 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. He is not permitted to run through a cop, though. A cop, at any time, either stands at a graph vertex or is in a helicopter (that is, he is above the game board). The goal is to land a helicopter on the vertex occupied by the robber. Note that, due to the use of helicopters, cops are not constrained to move along graph edges. The robber can see a helicopter approaching its landing vertex and he may run to a

[5] See, e.g., [32] for a counterexample

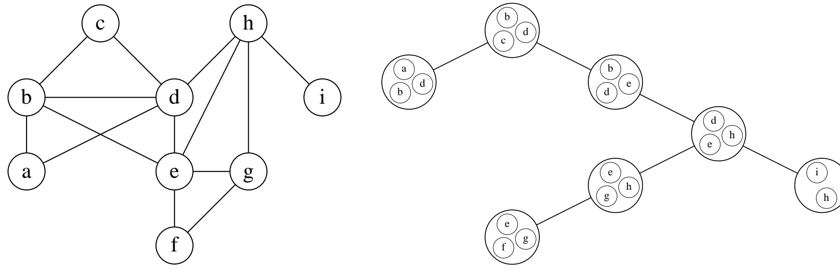

Figure 6: A graph together with a tree decomposition of width 2. Observe that—as demanded by the consistency property—each graph vertex induces a subtree in the decomposition tree.

new vertex before the helicopter actually lands. Thus, for a set of cops the goal is to occupy all vertices adjacent to the robber's vertex and to land one more remaining helicopter on the robber's vertex itself. 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. The tree decomposition indicates bottlenecks in the graph and thus reveals an underlying scaffold-like structure that can be exploited algorithmically.

Formally, tree decompositions and treewidth center around the following somewhat technical definition; Figure 6 shows a graph together with an optimal tree decomposition of width two.

Definition 3. *Let $G = (V, E)$ be a 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. $\bigcup_{i \in I} X_i = V$;
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 every 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. More generally, 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 [12], 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.

4.2 Case Study

Typically, tree decomposition based algorithms proceed according to the following two-stage scheme:

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, the decomposition tree provides us with sort of a scaffold that allows an efficient and consistent processing through the graph in order to solve the given problem. Note that 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 3. *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 [41].)

Observe that the following points of Definition 3 guarantee the validity of the above 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 neighbored bags that both contain v.

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.

4.3 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. Successful implementations for solving Vertex Cover with tree decomposition approaches have been reported by Alber et al. [3] and by Betzler et al. [9].

Another recent practical application of tree decompositions was given by Xu et al. [53] who proposed a tree decomposition based algorithm to solve two problems encountered in protein structure prediction, namely the prediction of backbone structure and side-chain prediction. To this end, they modeled these two problems as a graph labeling problem and showed that the resulting graphs have a very small treewidth in practice, allowing the problems to be solved efficiently.

Besides taking an input graph, computing a tree decomposition for it, and hoping that the resulting tree decomposition has a 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. [49] used a so-called conformational graph to specify the consensus sequence-structure of an RNA family. They were able to prove that the treewidth of this graph is basically determined by the structural elements that appear in the RNA. More precisely, they show 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.

A further strong example in this direction are probabilistic inference networks, which play a vital role in several decision support systems in medical, agricultural, and other applications [37, 6]. Here, tree decomposition based dynamic programming is a key solving method.

5 Color-Coding

The color-coding method due to Alon et al. [4] is a general method for finding small patterns in graphs. It is clearly not as generally applicable as data reduction or search trees, but can lead to very efficient algorithms for certain problems. In its simplest form, it can solve the Minimum Weight Path problem, which asks for the cheapest path of length k in a graph. This has been employed with protein–protein interaction networks to find signaling pathways [36, 46] and to evaluate pathway similarity queries [47].

5.1 Basic Concepts

Naively trying all roughly n^k possibilities of finding a small structure of k vertices within a graph of n vertices quickly leads to a combinatorial explosion, making this approach 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 the graph with one of k colors and to "hope" that all vertices in the subgraph searched for obtain different colors (it becomes *colorful*). When this happens, the task of finding the subgraph is greatly simplified: it can be solved by dynamic programming in a runtime where the exponential part depends only on the size k of the substructure searched for, as opposed to the $O(n^k)$ runtime of the naive approach.

Of course, most of the time the target structure is not actually colorful. Therefore, we have to repeat the process of randomly coloring and then searching (called *trial*) many times with a fresh coloring until with sufficiently high probability at least once our target structure is colorful. Since the number of trials also depends only on k (albeit exponentially), this algorithm has a fixed-parameter runtime.

5.2 Case Study

Formally stated, the problem we consider is the following:

> Minimum Weight Path
> **Input:** An undirected edge-weighted graph G 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 [27, 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 the minimum-weight colorful path is 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 cardinality-i subset S of vertex colors that denotes the minimum weight of a path that uses each color

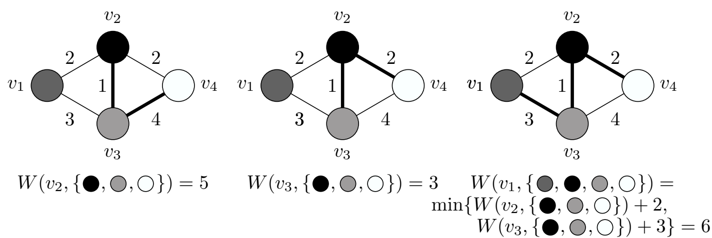

Figure 7: Example for solving Minimum Weight Path using the color-coding technique. Here, using (1) a new table entry (right) is calculated using two already known entries (left and middle).

in S exactly once and ends in v. Clearly, this 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 a colorful length-$(i+1)$ path that ends in a vertex $v \in V$ can 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

$$W(v, S) = \min_{e=\{u,v\}\in E} \Big(W(u, S \setminus \{\text{color}(v)\}) + w(e) \Big). \tag{1}$$

See Fig. 7 for an example.

It is easy to verify that the dynamic programming takes $O(2^k m)$ time. Whenever the minimum-weight length-k path in the input graph is colored with k colors (i.e., every vertex has a different color), then it is found. 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

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

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 $1 - (1 - P_c)^t$. To ensure that a colorful path is found with a probability greater than $1 - \varepsilon$ (for some $0 < \varepsilon \leq 1$), at least

$$t(\varepsilon) = \left\lceil \frac{\ln \varepsilon}{\ln(1 - P_c)} \right\rceil = |\ln \varepsilon| \cdot O(e^k) \tag{3}$$

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

5.3 Applications and Implementations

Protein interaction networks represent proteins by nodes 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. [46] 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.[6]

The currently most efficient implementation based on color-coding [36] is capable of finding optimal paths of length up to 13 in seconds within the yeast protein interaction network, which contains about 4 500 vertices. A variant with better theoretical runtime has recently been suggested [38], but to the best of our knowledge has not yet been implemented.

A particularly appealing aspect of the color-coding method 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 [46]).
- Not only the minimum-weight path can be sought after 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).
- Recently, it has been demonstrated that 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 [47].

Besides path problems, color-coding has also been used to give algorithms that find small trees in graphs [46] and for graph packing problems [38], where the goal is to find many disjoint copies of a pattern in a graph. No application outside the realm of small patterns in graphs is currently known.

6 Iterative Compression

Of the techniques we survey, iterative compression is by far the youngest, appearing first in a work by Reed, Smith, and Vetta in 2004 [45]. 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 already led to significant breakthroughs in showing fixed-parameter tractability results.

[6] 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, such that the goal is to minimize the sum of weights for the edges of a path.

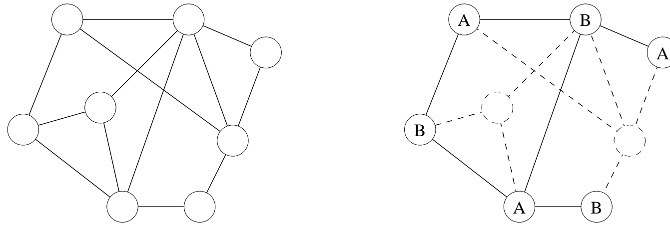

Figure 8: A Minimum Fragment Removal instance (left), and an optimal solution (right): when deleting two fragments (dashed), the remaining fragments can be allocated to the two chromosome copies (A and B) such that no conflicting fragments get the same assignment.

6.1 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.*

Using this routine, one finds an optimal solution to a problem by inductively building up the problem structure and iteratively compressing intermediate solutions. The point is that if the compression routine is a fixed-parameter algorithm with respect to the parameter k, then so is the whole algorithm.

The main strength of iterative compression is that it allows one to see the problem from a different angle. For the compression routine, we do 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 fixed-parameter algorithm.

However, while the mode of use of the compression routine is usually straightforward, finding the compression routine itself is not—even when we already know a problem to be fixed-parameter tractable, it is not clear that a compression routine with interesting runtime exists.

6.2 Case Studies

The showcase for iterative compression is the Minimum Fragment Removal problem, also known as Graph Bipartization. This problem appears in the context of SNP haplotyping [42]. 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 connected by an edge.

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. 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 MINIMUM FRAGMENT REMOVAL with this parameter [45]. We sketch how to apply this to finding an optimal solution (a *removal set*) for MINIMUM FRAGMENT REMOVAL: Starting with an empty graph G' and an empty fragment removal set C', we add one vertex v at a time from the original graph to both G' (including edges to vertices already in G') and to the removal set. Then C' is still a valid removal set. While C' may not be optimal (it can be too large by 1), we can find an optimal removal set for G' by applying the compression routine to G' and C'. Since eventually $G' = G$, we obtain an optimal removal set 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 [35, 45]. The runtime of the complete algorithm is $O(3^k \cdot mn)$.

6.3 Applications and Implementations

Nearly all of the currently known iterative compression algorithms solve *feedback set problems* in graphs, that is, problems where one wishes to destroy certain cycles in a graph by deleting at most k vertices or edges (see [25] for a survey on feedback set problems). Probably the most prominent among them is FEEDBACK VERTEX SET, which also has applications for genetic linkage analysis [8]. However, the currently known iterative compression algorithms [19, 34] exhibit a rather bad combinatorial explosion; a randomized fixed-parameter algorithm from [7] so far appears to be better for practical applications.

While thus no convincing practical results for iterative compression-based algorithms for FEEDBACK VERTEX SET are currently known, first experimental results for iterative compression-based algorithms for MINIMUM FRAGMENT REMOVAL are quite encouraging. An implementation, improved by heuristics, can solve all 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 [35].

7 Conclusion

We surveyed several techniques for developing efficient fixed-parameter algorithms for computationally hard (biological) problems. A broader perspective is given in the recent monograph [41]. Since many biologically relevant computational 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. Since the theory of fixed-parameter algorithmics is still very young (basically starting in the nineties of the last century), the whole field is still vividly developing, offering many tools and challenges with potentially high impact particularly in the field of computational (molecular) biology.

8 Notes

1. Parameter choice: Fixed-parameter algorithms are the better the smaller the parameter value is. Hence, try to find parameterizations with small parameter values.

2. A natural way to find significant problem parameterizations is to identify parameters that measure the distance from tractable (that is, efficiently solvable) problem instances.

3. Avoid exponential-space algorithms when there are alternatives, since in practice this usually turns out to be more harmful than exponential time.

4. Always start by designing data reduction rules. They are helpful in combination with basically any algorithmic technique you might try later.

5. Use data reduction not only as a preprocessing step. In search tree algorithms, branching can produce new opportunities for data reduction. Therefore, apply data reductions repeatedly during the course of the whole algorithm.

6. In search tree algorithms, try to avoid complicated case distinctions when a simpler search strategy yields almost the same running time bounds.

7. Use high-level programming languages that allow for quicker implementation of ideas and are less error prone. For exponential-time algorithms, algorithmic improvements can often lead to a speedup by several orders of magnitude, dwarfing the speedup of 2–10 gained by choice of programming language.

8. Try to augment search tree algorithms with admissible heuristic evaluation functions (see Section 3.3).

9. Many fixed-parameter algorithms can be easily adapted to run on parallel machines.

10. Do not be afraid of bad upper bounds for fixed-parameter algorithms—the analysis is worst-case and often much too pessimistic.

Acknowledgments This work was supported by the Deutsche Forschungsgemeinschaft, Emmy Noether research group PIAF (fixed-parameter algorithms), NI 369/4 (Falk Hüffner), and the Deutsche Telekom Stiftung (Sebastian Wernicke).

References

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

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

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

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

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

[6] E. Bachoore and H. L. Bodlaender. Weighted treewidth: Algorithmic techniques and results. Technical Report UU-CS-2006-013, Department of Information and Computing Sciences, Universiteit Utrecht, 2006.

[7] A. Becker, R. Bar-Yehuda, and D. Geiger. Randomized algorithms for the loop cutset problem. *Journal of Artificial Intelligence Research*, 12:219–234, 2000.

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

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

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

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

[12] H. L. Bodlaender. Discovering treewidth. In *Proc. 31st Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '05)*, volume 3381 of *LNCS*, pages 1–16. Springer, 2005.

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

[14] J. Chen, I. A. Kanj, and G. Xia. Improved parameterized upper bounds for vertex cover. In *Proc. 31st International Symposium on Mathematical Foundations of Computer Science (MFCS '06)*, volume 4162 of *LNCS*, pages 238–249. Springer, 2006.

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

[16] B. Chor, M. R. Fellows, and D. W. Juedes. Linear kernels in linear time, or how to save k colors in $O(n^2)$ steps. In *Proc. 30th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '04)*, volume 3353 of *LNCS*, pages 257–269. Springer, 2004.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

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

[19] F. K. H. A. Dehne, M. R. Fellows, M. A. Langston, F. A. Rosamond, and K. Stevens. An $O(2^{O(k)}n^3)$ FPT algorithm for the undirected feedback vertex set problem. In *Proc. 11th International Computing and Combinatorics Conference (COCOON '05)*, volume 3595 of *LNCS*, pages 859–869. Springer, 2005. Long version to appear in *Theory of Computing Systems*.

[20] F. K. H. A. Dehne, M. A. Langston, X. Luo, S. Pitre, P. Shaw, and Y. Zhang. The cluster editing problem: Implementations and experiments. In *Proc. 2nd International Workshop on Parameterized and Exact Computation (IWPEC '06)*, volume 4169 of *LNCS*, pages 13–24. Springer, 2006.

[21] R. Diestel. *Graph Theory*. Springer, 3rd edition, 2005.

[22] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

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

[24] A. Felner, R. E. Korf, and S. Hanan. Additive pattern database heuristics. *Journal of Artificial Intelligence Research*, 21:1–39, 2004.

[25] P. Festa, P. M. Pardalos, and M. G. C. Resende. Feedback set problems. In D. Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization, Vol. A*, pages 209–258. Kluwer, 1999.

[26] J. Flum and M. Grohe. *Parameterized Complexity Theory*. Springer, 2006.

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

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

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

[30] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Data reduction, exact, and heuristic algorithms for clique cover. In *Proc. 8th Workshop on Algorithm Engineering and Experiments (ALENEX '06)*, pages 86–94. SIAM, 2006. Long version to appear under the title "Data reduction and exact algorithms for clique cover" in *ACM Journal of Experimental Algorithmics*.

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

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

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

[34] J. Guo, J. Gramm, F. Hüffner, R. Niedermeier, and S. Wernicke. Improved fixed-parameter algorithms for two feedback set problems. In *Proc. 9th Workshop on Algorithms and Data Structures (WADS '05)*, volume 3608 of *LNCS*, pages 158–168. Springer, 2005. Long version to appear under the title "Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization" in *Journal of Computer and System Sciences*.

[35] F. Hüffner. Algorithm engineering for optimal graph bipartization. In *Proc. 4th International Workshop on Efficient and Experimental Algorithms (WEA '05)*, volume 3503 of *LNCS*, pages 240–252. Springer, 2005.

[36] F. Hüffner, S. Wernicke, and T. Zichner. Algorithm engineering for color-coding to facilitate signaling pathway detection. In *Proc. 5th Asia-Pacific Bioinformatics Conference (APBC '07)*, Advances in Bioinformatics and Computational Biology. World Scientific, 2007. To appear.

[37] F. V. Jensen. *Bayesian Networks and Decision Graphs*. Springer, 2001.

[38] J. Kneis, D. Mölle, S. Richter, and P. Rossmanith. Divide-and-color. In *Proc. 32nd International Workshop on Graph-Theoretic Concepts in Computer Science (WG '06)*, LNCS. Springer, 2006. To appear.

[39] D. Marx. The closest substring problem with small distances. In *Proc. of the 46th Annual IEEE Symposium on Foundations of Computer Science (FOCS '05)*, pages 63–72, 2005.

[40] Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer, 2nd edition, 2004.

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

[42] A. Panconesi and M. Sozio. 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*, pages 266–277. Springer, 2004.

[43] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

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

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

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

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

[48] S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1998.

[49] Y. Song, C. Liu, R. L. Malmberg, F. Pan, and L. Cai. 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)*, pages 223–234. IEEE Computer Society, 2005.

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

[51] V. V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[52] G. Wu, J.-H. You, and G. Lin. 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*, pages 65–76. Springer, 2005.

[53] J. Xu, F. Jiao, and B. Berger. A tree-decomposition approach to protein structure prediction. In *Proc. 4th International IEEE Computer Society Computational Systems Bioinformatics Conference (CSB 2005)*, pages 247–256. IEEE Computer Society, 2005.