Rolf Niedermeier

Invitation to Fixed-Parameter Algorithms

October 23, 2002

Preface

This work is based on my occupation with parameterized complexity and fixed-parameter algorithms which began more than four years ago. During that time, my three most important research partners were Jochen Alber, Jens Gramm, and Peter Rossmanith. Nearly all of my work in these years emerged from research with one or another of these three people. I also profited from joint research with Hans L. Bodlaender, Michael R. Fellows, Henning Fernau, Jiong Guo, and Ton Kloks which led to some findings in this project. Clearly, I owe sincere thanks to all of the mentioned people and many more unnamed ones.

My research was generously supported by the Deutsche Forschungsgemeinschaft (DFG) through the projects "Parameterized Complexity and Exact Algorithms (PEAL)", NI 369/1-1,2, and "Optimal Solutions for Hard Problems from Computational Biology (OPAL)", NI 369/2-1, which paid for the work of Jochen Alber, Jens Gramm, and several students.

In the subsequent text, which is mainly based on own research material, I always give the connections to the corresponding publications. In order to keep the presentation within reasonable size, from time to time (awkward) technical details and proofs are omitted—all of them can be found in the cited literature.

Tübingen, September 2002 *Rolf Niedermeier*

Contents

1. Foundations

1.1 Keep the Parameter Fixed

How to cope with computational intractability? Several methods to deal with this problem have been developed: approximation algorithms [23, 152], average-case analysis [153], randomized algorithms [197], and heuristic methods [190]. All of them have their drawbacks, such as the difficulty of approximation, lack of mathematical tools and results, limited power of the method itself, or the lack of provable performance guarantees at all. Clearly, the direct way of attacking *NP*-hard problems is in providing deterministic, exact algorithms. However, in this case, one has to deal with exponential running times. Currently, there is an increasing interest in faster exact solutions for *NP*-hard problems. In particular, performance bounds are to be *proven.* Despite their exponential running times, these algorithms may be interesting from a theoretical as well as a practical point of view. With respect to the latter, note that for some applications, really exact solutions are needed or the input instances are of modest size, so that exponential running times can be tolerated.

Parameterized complexity theory, whose leitmotif can be characterized by the words "not all forms of intractability are created equal"[1] [88], is another proposal on how to cope with computational intractability in some cases. In a sense, so-called "fixed-parameter algorithms" form a variant of exact, exponential time solutions mainly for *NP*-hard problems. This is the basic theme of this work and some closely related survey articles [11, 89, 92, 104, 202, 224].

Many hard computational problems have the following general form: given an object x and a nonnegative integer k, does x have some property that depends on k? For instance, the *NP*-complete Vertex Cover problem is: given an undirected graph $G = (V, E)$ and a nonnegative integer k, does G have a vertex cover of size at most k? Herein, a vertex cover is a subset of vertices $C \subseteq V$ such that each edge in E has at least one of its endpoints in C. In parameterized complexity theory, k is called the *parameter*. In many applica-

[1] Specifically, this means that one can often distinguish between hard problems that nevertheless turn out to be efficiently worst-case solvable in many applications and others that are not.

tions, the parameter k can be considered to be "small" in comparison with the size $|x|$ of the given object x. Hence, it may be of high interest to ask whether these problems have deterministic algorithms that are exponential *only* with respect to k and polynomial with respect to $|x|$. In this sense, parameterized complexity is nothing but a *two-dimensional* complexity theory.[2]

The basic observation of parameterized complexity, as chiefly developed by Downey and Fellows [88], is that for many hard problems, the seemingly inherent "combinatorial explosion" really can be restricted to a (hopefully) "small part" of the input, the parameter. So, for instance, VERTEX COVER allows for an algorithm with running time $O(kn+1.29^k)$ [60, 204, 207], where the parameter k is a bound on the maximum size of the vertex cover set we are looking for and n is the number of vertices of the given graph. The best known "non-parameterized" solution for VERTEX COVER is due to Robson [231, 232]. He showed that INDEPENDENT SET and, thus, VERTEX COVER can be solved in time $O(1.19^n)$. However, for $k \leq 0.79\,n$, the above mentioned fixed-parameter solution turns out to be better. Note that in several applications $k \ll n$ is a natural assumption.

The aim of this work is not to list as many parameterized problems as possible together with (if existing) their fixed-parameter algorithms, but to give a, in a sense, application-oriented introduction to the prosperous field of developing and analyzing efficient fixed-parameter algorithms.

To achieve this, we start with more methodologically oriented considerations introducing basic techniques and results, followed by a series of case studies where fixed-parameter algorithms have been successfully developed and applied. Finally, we give a brief collection of some of many facets related to fixed-parameter algorithms, ranging from structural complexity theory to implementation and experimentation issues.

At the necessary risk of being incomplete and biased, the whole presentation is kept within clear, relatively small dimensions. Following the given references to the extensive literature it might serve (and it is meant as) a springboard into this young, fast developing, and promising field of research and its applications.

1.2 Preliminaries and Agreements

In this section, we briefly summarize some of the notation used throughout the work. Still, however, we assume some familiarity with the fundamentals of algorithms and complexity, cf., e.g., [69, 148, 189, 212, 241].

[2] Hromkovič [154] also points to close connections between fixed-parameter tractability and pseudo-polynomial-time algorithms. He considers the concept of parameterized complexity as a generalization of the concept of pseudo-polynomial-time algorithms.

Basic Sets. Finite alphabets are usually denoted by Σ. We often deal with the set of nonnegative integer numbers, also denoted by $\mathcal{N}$. By way of contrast, $\mathcal{R}$ refers to the real numbers, where $\mathcal{R}^+$ is the subset of all positive reals.

Problems. This work is about a particular kind of exact algorithms that solve computationally hard problems. We present problems in an "input-question-style." This first of all refers to *decision problems* where it is asked whether for a given input instance of a problem the answer is "yes" or "no." As a rule, however, there exists a naturally corresponding *optimization problem* which asks to minimize or maximize a certain cost value. All algorithms in this work can be used not only to output "yes" or "no" in order to answer the decision question but also can be easily adapted to constructively output a desired solution object—most of the time they already do. Moreover, in most cases the algorithms can also be modified to deliver *optimal* solutions (i.e., not only fulfilling the given constraints, but being optimal among all these solutions) to the corresponding optimization problem. We do not make a sharp distinction between decision or optimization and the subsequently derived parameterized problems (Subsection 1.3.1) because it will always be made clear from the context what is meant. As a matter of fact, in a certain sense one may identify decision and parameterized problems in many cases; note, however, that in the latter case solution objects will be constructed.

Model of Computation. The *Random Access Machine (RAM)* model of computation will be used to give a machine-independent description of algorithms, sometimes using pseudo-code in the same style as it was usual in the standard textbooks on algorithms. Particular features of RAM computation are that each "simple" operation (basic arithmetic, assignments, if-statements, etc.) takes one time unit, as well as every access to one memory cell (with some reasonable word size) does. In particular, the word size is big enough to hold all numbers occurring in the presented algorithms. None of the described RAM features will be misused in the sense that the corresponding algorithms could not be implemented on existing computers in an efficient way. This is true although the RAM model, in order to simplify the analysis of algorithms, for instance, does not distinguish between different levels of the memory hierarchy (cache versus disk etc.).

Running times. We use the "big Oh notation" to analyze the running times of our algorithms. Hence, we ignore constant factors but, if appropriate, we point to cases where the involved constant factors of the algorithms are large and, thus, might threaten or destroy the practical usefulness of the algorithms. All running time analysis in this work is *worst-case* analysis, that is, the presented bounds hold over *all* input instances of a given problem. At few points (in particular, concerning applications in computational biology) we will indicate that sometimes the given bounds or the worst-case analysis

as such are too pessimistic when the algorithm is applied in practice—to mathematically analyze any kind of average-case complexity, however, is out of the scope of this work.[3]

Strings. A *word* or, equivalently, a *string* over a finite alphabet Σ is a sequence of elements from Σ. Strings will play a particularly important role in those fixed-parameter algorithms related to computational biology. Here, we need the *Hamming distance* measure $d_H(s,t)$ between two strings s and t of same length L. It is defined as

$$|\{\, p \mid s[p] \neq t[p], 1 \leq p \leq L \,\}|,$$

where $s[p]$ denotes the character at position p in string s.

Graphs. The majority of computational problems we will study is based on graphs. An *undirected graph* G is a pair (V,E), where V is a finite set of *vertices* and E is a finite set of *edges* which are *unordered* pairs of vertices. All graphs considered in this work are undirected. Furthermore, all our graphs are *simple* (that is, there is at most one edge between each pair of vertices) and do not contain *self-loops* (that is, edges from a vertex to itself are forbidden). The *degree* of a vertex is the number of edges incident on it. A graph is *d-regular* if every vertex has degree exactly d. The *neighborhood* of a vertex v in graph $G = (V,E)$ is defined as $N(v) := \{\, u \mid \{u,v\} \in E \,\}$. In a d-regular graph each neighborhood has size exactly d. For a graph $G = (V,E)$ and a set $V' \subseteq V$, the subgraph of G *induced* by V' is denoted by $G[V'] = (V', E')$, where $E' := \{\, \{u,v\} \in E \mid u \in V' \wedge v \in V' \,\}$. A graph is *connected* if every pair of vertices is connected by a path. If a graph is not connected then it naturally falls into its *connected components.* Two graphs $G = (V,E)$ and $G' = (V',E')$ are *isomorphic* if there exists a bijection $g : V \to V'$ such that $(u,v) \in E$ if and only if $(g(u), g(v)) \in E'$.

In this work we often deal with a special class of graphs, *planar graphs.* A graph G is called planar if it can be drawn in the plane such that no two edges cross. A particular crossing-free drawing of a graph is called *plane embedding* of that graph and a *plane graph* is a planar graph together with its embedding in the plane. Planar graphs are "sparse" in the sense that the well-known *Euler formula* says that a planar graph with n vertices has at most $3n - 6$ edges—a general graph may contain up to $n(n-1)/2$ edges. The *faces* of a plane graph are the maximal regions of the plane that contain no point used in the embedding. A *triangular face* is a face enclosed by only three edges (the smallest number possible). A plane graph where every such face boundary is a cycle of three edges is called a *triangulation.*

[3] Indeed, due to the inherent difficulties of average-case analysis starting with the "simple" problem to define what a (practical) average case is and continuing with difficult mathematical problems behind, relatively little is generally known about average-case complexity.

Complexity. Efficiency of algorithms from a theoretical computer science point of view means algorithms running in time polynomial in the input size. By way of contrast, there is a vast amount of important computational problems that so far resisted to efficient solutions. These problems are known as *NP*-hard problems and the classic reference to the theory of computational intractability is the book of Garey and Johnson [119]. A typical *NP*-hard problem is the SATISFIABILITY problem: Given a boolean formula in conjunctive normal form, decide whether or not there is a satisfying truth assignment. An *NP*-hard problem is *NP*-complete if it can be solved in polynomial time by a nondeterministic Turing machine[4]—the (complexity) class of all problems solvable by this means is denoted by *NP*. All computational problems considered in this work are *decidable*, that is, there always is an algorithm deciding the given problem in finite time.

Since *NP*-hard decision problems generally need exponential (or worse) running time to be solved exactly (which is impractical in most cases) a less ambitious goal in attacking the corresponding optimization versions of the *NP*-hard problems, as pursued by *approximation algorithms,* is to solve the problem in polynomial time but not necessarily optimal [23, 152, 212, 258]. The hope is that the provided solution is "not too far from the optimum." The quality of the approximation (with respect to the given optimization criterion) is measured by the *approximation ratio*, a number ϵ between 0 and 1 (see the given literature for details). The smaller ϵ is, the better the approximation will be—$\epsilon = 0$ means an optimal solution. A *polynomial-time approximation scheme (PTAS)* for an optimization problem then is an algorithm which, for each $\epsilon > 0$ and each problem instance, returns a solution with approximation ratio ϵ. The polynomial running time of this algorithm, however, crucially depends on $1/\epsilon$. If the polynomial depends *polynomially* on $1/\epsilon$ as well then the approximation scheme is called *fully polynomial (FPTAS)*. We mention in passing that the class *MaxSNP* [214] (also known as *APX*) of optimization problems can be "syntactically" defined together with a reducibility concept. The point is that *MaxSNP-complete* problems are unlikely to have polynomial-time approximation schemes. The optimization version of VERTEX COVER (which plays a key role on this work) is *MaxSNP*-hard. More details and much more advanced material on the hardness of approximation but also positive results can be found in the given literature.

The recent book of Hromkovič [154] provides an extensive survey on various ways (including approximation and exact algorithms) to cope with computational intractability. Related texts can also be found in [22].

[4] Note that polynomial time on a deterministic Turing machine is equivalent to polynomial time on a RAM.

1.3 Parameterized Complexity—a Brief Overview

We briefly sketch general aspects of the theoretical basis of the study of parameterized complexity. For a detailed exposition we refer to the research monograph of Downey and Fellows [88]. The focus of this section, however, lies on the practical relevance of fixed-parameter tractability, and the consideration of structural complexity issues is limited.

1.3.1 Basic Theory

The *NP*-complete Vertex Cover problem is the best studied problem in the field of fixed-parameter algorithms:

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Question:** Is there 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?

Vertex Cover is *fixed-parameter tractable*: There are algorithms solving it in time less than $O(kn + 1.29^k)$ [60, 204, 207]. By way of contrast, consider the also *NP*-complete Clique problem:

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Question:** Is there a subset of vertices $C \subseteq V$ with k or more vertices such that C forms a clique by having all possible edges between the vertices in C?

Clique appears to be *fixed-parameter intractable*: It is *not* known whether it can be solved in time $f(k) \cdot n^{O(1)}$, where f might be an arbitrarily fast growing function only depending on k [88]. More precisely, unless $P = NP$, the well-founded conjecture is that no such algorithm exists. The best known algorithm solving Clique runs in time $O(n^{ck/3})$ [201], where c is the exponent on the time bound for multiplying two integer $n \times n$ matrices (currently best known, $c = 2.38$, see [68]). Note that $O(n^{k+2})$ is the running time for the straightforward algorithm just checking all size k subsets. The decisive point is that k appears in the exponent of n, and there seems to be no way "to shift the combinatorial explosion only into k," independent from n.

The observation that *NP*-complete problems like Vertex Cover and Clique behave completely differently in a "parameterized sense" lies at the very heart of parameterized complexity, a theory pioneered by Downey and Fellows [88]. Subsequently, we will concentrate on the world of fixed-parameter tractable problems as, e.g., exhibited by Vertex Cover. Hence, here we only briefly sketch some very basics from the theory of parameterized intractability in order to provide some background on parameterized complexity theory and the ideas behind it. For many further details and an additional discussion, we once more refer to Downey and Fellows' monograph [88].

We begin with some basic definitions of parameterized complexity theory.

Definition 1.3.1. *A* parameterized problem *is a language $L \subseteq \Sigma^* \times \Sigma^*$, where Σ is a finite alphabet. The second component is called the* parameter *of the problem.*

In basically all examples in this work the parameter is a nonnegative integer. Hence, we will subsequently write $L \subseteq \Sigma^* \times \mathcal{N}$ instead of $L \subseteq \Sigma^* \times \Sigma^*$. In principle. however, the above definition leaves it open to also define more complicated parameters, e.g., substructures one is searching for in a cited structure.[5] The key notion of this work is the concept of fixed-parameter tractability.

Definition 1.3.2. *A parameterized problem L is* fixed-parameter tractable *if the question "$(x_1, x_2) \in L$?" can be decided in running time $f(|x_2|) \cdot |x_1|^{O(1)}$, where f is an arbitrary function on nonnegative integers. The corresponding complexity class is called FPT.*

In the preceding definition, one may assume any standard model of sequential computation such as deterministic Turing machines. For the sake of convenience, in the following, if not stated otherwise, we will always take the parameter, then denoted by k, as a nonnegative integer encoded in unary. Hence, we will simply write $f(k)$ for the running time function only depending on the parameter.

Attempts to prove nontrivial, absolute lower bounds on the computational complexity of problems have made relatively little progress [43]. Hence, it is probably not surprising that up to now there is no proof that *no* $f(k) \cdot n^{O(1)}$ time algorithm for Clique exists. In a more complexity-theoretical language, this can be rephrased by saying that it is unknown whether Clique $\in FPT$. Analogously to classical complexity theory, Downey and Fellows developed some way out of this quandary by providing a completeness program. The completeness theory of parameterized intractability involves significantly more technical effort than the classical one. We very briefly sketch some integral parts of this theory in the following.

To start with a completeness theory, we first need a reducibility concept:

Definition 1.3.3. *Let $L, L' \subseteq \Sigma^* \times \mathcal{N}$ be two parameterized languages. We say that L* reduces *to L' by a standard* parameterized m-reduction *if there are functions $k \mapsto k'$ and $k \mapsto k''$ from $\mathcal{N}$ to $\mathcal{N}$ and a function $(x, k) \mapsto x'$ from $\Sigma^* \times \mathcal{N}$ to Σ^* such that*

1. *$(x, k) \mapsto x'$ is computable in time $k''|x|^c$ for some constant c and*
2. *$(x, k) \in L$ iff $(x', k') \in L'$.*

Notably, most reductions from classical complexity turn out *not* to be parameterized ones [88]. For instance, the well-known reduction from Independent Set to Vertex Cover (see [212]), which is given by letting $G' = G$ and $k' = |V| - k$ for a graph instance $G = (V, E)$, is not a parameterized one.

[5] An example is given by the Graph Minor Order Testing problem, cf. [88].

This is due to the fact that the reduction function of the parameter $k \mapsto k'$ strongly depends on the instance G itself, hence contradicting the definition of a parameterized m-reduction. However, the reductions from INDEPENDENT SET to CLIQUE and vice versa, which are obtained by simply passing the original graph over to the complementary one for $k' = k$, indeed are parameterized ones. Therefore, these problems are of comparable difficulty in terms of parameterized complexity.

Now, the "lowest class of parameterized intractability" can be defined as the class of parameterized languages that are equivalent to the so-called SHORT TURING MACHINE ACCEPTANCE problem (also known as the k-STEP HALTING problem).

> **Input:** A nondeterministic Turing machine M with its transition table, an input word x, and a nonnegative integer k.
> **Question:** Does M accept x in a computation of at most k steps?

This is the parameterized analogue to the TURING MACHINE ACCEPTANCE problem—the basic generic *NP*-complete problem in classical complexity theory. Together with the above introduced reducibility concept, SHORT TURING MACHINE ACCEPTANCE can now be used to define the currently lowest class of parameterized intractability, that is, $W[1]$.

Definition 1.3.4. *The class of all parameterized languages that reduce by a standard parameterized m-reduction to* SHORT TURING MACHINE ACCEPTANCE *is called* $W[1]$. *A problem which lets all other problems in* $W[1]$ *reduce to it is called* $W[1]$-hard*; if, additionally, it is contained in* $W[1]$ *then it is called* $W[1]$-complete.

Note that $W[1]$ is originally defined in a "circuit-based", more technical way, refer to [88] for any details.[6] Clearly, by definition, SHORT TURING MACHINE ACCEPTANCE is $W[1]$-complete. Other problems that are *W[1]-complete* include CLIQUE and INDEPENDENT SET, where the parameter is the size of the relevant vertex set [88]. Note that a parameterized m-reduction from one of these problems to VERTEX COVER would lead to a collapse of the classes $W[1]$ and FPT. Downey and Fellows provide an extensive list of many more $W[1]$-complete problems [88].

As a matter of fact, a whole hierarchy of parameterized intractability can be defined, $W[1]$ only being the lowest level. In general, the classes $W[t]$ are defined based on "logical depth" (i.e., the number of alternations between unbounded fan-in And- and Or-gates) in boolean circuits. For instance, the well-known DOMINATING SET problem on undirected graphs, which is *NP*-complete, is known to be $W[2]$-complete, where the size of the dominating set is the parameter [88].

[6] This is also where the "W" stems from—it refers to the *weft* of boolean circuits, that is, the maximum number of unbounded fan-in gates on any path from the input variables to the output gate of the (decision) circuit.

We mention in passing that Flum and Grohe recently came up with some close connections between parameterized complexity theory and the general logical framework of descriptive complexity theory [112, 114, 138]. They study the parameterized complexity of various model-checking problems through the syntactical structure of the defining sentences. Moreover, they provide a descriptive characterization of the class FPT, as well as an approach of how to characterize classes of intractability by syntactical means.

There exists a very rich structural theory of parameterized complexity somewhat similar to classical complexity. Observe, however, that in some respects parameterized complexity appears to be, in a sense, "orthogonal" to classical complexity: For instance, the problem of computing the V-C dimension from learning theory [37, 215], which is not known (and not believed) to be NP-hard, is $W[1]$-complete [86, 87]. Thus, although in the classical sense it appears to be easier than Vertex Cover (which is NP-complete), the opposite appears to be true in the parameterized sense, because Vertex Cover is in FPT.

From a more practical point of view (and the remainder of this work), it is sufficient to distinguish between $W[1]$-hardness and membership in FPT. Thus, for an algorithm designer not being able to show fixed-parameter tractability of a problem, it may be "sufficient" to give a reduction from, e.g., Clique to the given problem using a standard parameterized m-reduction. This then gives a concrete indication that, unless $P = NP$, the problem is unlikely to allow for an $f(k) \cdot n^{O(1)}$ time algorithm. One piece of circumstantial evidence for this is the result showing that the equality of $W[1]$ and FPT would imply a time $2^{o(n)}$ algorithm for the NP-complete 3-Satisfiability (formulas in conjunctive normal form with clauses consisting of at most three literals) problem [88] (where n denotes the number of variables of the given Boolean formula), which would mean a breakthrough in computational complexity theory.

1.3.2 Interpreting Fixed-Parameter Tractability

The remainder of this work concentrates on the world inside the class FPT of fixed-parameter tractable problems and the potential it carries for improvements and future research. We therefore finish this section by an interpretation of FPT under some application aspects. Note that in the definition of FPT the function $f(k)$ may take unreasonably large values, e.g.,

$$2^{2^{2^{2^{2^{2^k}}}}}.$$

Thus, showing that a problem is a member of the class FPT does not necessarily bring along an efficient algorithm (not even for small k). In fact, many problems that are classified fixed-parameter tractable still wait for such efficient, practical algorithms. In this sense, we strongly have to distinguish two

k	$f(k) = 2^k$	$f(k) = 1.32^k$	$f(k) = 1.29^k$
10	$\approx 10^3$	≈ 16	≈ 13
20	$\approx 10^6$	≈ 258	≈ 163
30	$\approx 10^9$	≈ 4140	≈ 2080
40	$\approx 10^{12}$	$\approx 6.7 \cdot 10^4$	$\approx 2.7 \cdot 10^4$
50	$\approx 10^{15}$	$\approx 1.1 \cdot 10^6$	$\approx 3.4 \cdot 10^5$
75	$\approx 10^{22}$	$\approx 1.1 \cdot 10^9$	$\approx 2.0 \cdot 10^8$
100	$\approx 10^{30}$	$\approx 1.1 \cdot 10^{12}$	$\approx 1.1 \cdot 10^{11}$
500	$\approx 10^{150}$	$\approx 1.9 \cdot 10^{60}$	$\approx 2.0 \cdot 10^{55}$

Table 1.1. Comparing the efficiency of various VERTEX COVER algorithms with respect to the exponential terms given to be found in the literature.

different aspects of fixed-parameter tractability: The theoretical part which consists in classifying parameterized problems along the W-hierarchy (i.e., proving membership in FPT or hardness for $W[1]$) and the algorithmic component of actually finding efficient algorithms for problems inside the class FPT.

The *Graph Minor Theorem* by Robertson and Seymour [88, chapter 7], for instance, provides a great tool for the classification of graph problems. It states that, for a given family of graphs $\mathcal{F}$ which is closed under taking minors, membership of a graph G in $\mathcal{F}$ can be checked by analyzing whether a certain finite "obstruction set" appears as a minor in G. Moreover, the GRAPH MINOR ORDER TESTING problem is in FPT [88], or, more precisely, for a fixed graph G of n vertices there is an algorithm with running time $O(f(|H|) \cdot n^3)$ that decides whether a graph H is a minor or not. As a matter of fact, the set $\mathcal{F}_k$ of vertex covers of size $\leq k$ are closed under taking minors. Hence there exists a finite obstruction set $\mathcal{O}_k$ for $\mathcal{F}_k$. The above method then yields the existence of a fixed-parameter algorithm for VERTEX COVER. However, in general, the function f appearing in the GRAPH MINOR ORDER TESTING algorithm grows fast and the constants hidden in the O-notation are huge. Moreover, finding the obstruction set in the Graph Minor Theorem, in general, is highly non-constructive. Thus, the above mentioned method may only serve as a classification tool.

Taking into account that the theory of fixed-parameter tractability should be understood as an approach to cope with inherently hard problems, it is necessary to aim for practical, efficient fixed-parameter algorithms. In the case of VERTEX COVER, for example, it is fairly easy to come up with an algorithm of running time $O(2^k n)$ using a simple search tree strategy. The base of the exponential term in k was further improved, now below 1.29 (see [60, 204, 205, 207] for a complete exposition). Table 1.1 relates the size of the exponential term for these base values. From this table we can conclude that improvements in the exponential growth of f can lead to significant changes in the running time, and, therefore, deserve investigation.

Finally, to demonstrate the problematic nature of the comparison "fixed-parameter tractable" versus "fixed-parameter intractable," let us compare the functions 2^{2^k} and $n^k = 2^{(k \log n)}$. The first refers to fixed-parameter tractability, the second to intractability. It is easy to verify that assuming input sizes n in the range from 10^3 up to 10^{15}, the value of k where 2^{2^k} starts to exceed n^k is in the small range $\{6, 7, 8, 9\}$.

A striking example in this direction is that of computing the treewidth of graphs (see Section 4.4 for more on that). For constant k, there is a famous result giving a linear time algorithm to compute whether a graph has treewidth at most k [38]. More precisely, the algorithm has running time $2^{\Theta(k^3)} \cdot k^{O(1)} \cdot n$.

As a consequence, the $O(n^{k+1})$ algorithm [17] appears to be "more practical." This shows how careful one has to be with the term fixed-parameter tractable since, in practical cases with reasonable input sizes, a fixed-parameter intractable problem might turn out to have a still more "efficient" solution than a fixed-parameter tractable one.

Keeping these considerations in mind, this work now starts off to try to give convincing arguments that the field of fixed-parameter tractability contains a wealth of interesting questions and answers, relevant for the practice of algorithmic strategies against computationally hard problems.

1.4 Vertex Cover – an Illustrative Example

Vertex Cover is the so far most popular fixed-parameter tractable problem. It is an important problem in combinatorial optimization and graph theory and it is easy to grasp. Many aspects of fixed-parameter algorithms can naturally be developed and illustrated using Vertex Cover as a running example. Hence, Vertex Cover also plays a prominent role in this work. We start with recalling the definition:

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Question:** Is there 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?

Vertex Cover has seen quite a long history in the development of fixed-parameter algorithms [88, page 5]. Surprisingly, a lot of papers published fixed-parameter results (e.g., cf. [215]) on Vertex Cover that are worse than the $O(2^k n)$ time bound that directly follows from the elementary search tree method e.g. described in Mehlhorn's text book on graph algorithms [189, page 216]. The corresponding simple observation, which is also used in the ratio 1/2 approximation algorithm for Vertex Cover, is as follows: Each edge $\{u, v\}$ has to be covered. Hence, u or v has to be in the cover (perhaps both). Thus, building a search tree where we branch as either bringing u or v in the cover, deleting the respective vertex together with its incident edges

and continuing recursively with the remaining graph by choosing the next edge (which is arbitrary) solves the problem. The search tree with depth at most k has at most 2^k nodes, each of which can be processed in linear time using suitable data structures. In recent times, there has been very active research on lowering the size of the search tree [30, 60, 92, 204, 207, 249], the best known result now being better than 1.29^k [60, 204, 207]. The basic idea behind all of these papers is to use two fundamental techniques for developing efficient fixed-parameter algorithms, that is, *bounded search trees* and *reduction to problem kernel.* Both will be explained in greater detail in chapters particularly dedicated to these methods (Chapters 2 and 3). To improve the search tree size, intricate case distinctions with respect to the degree of graph vertices were designed. As to reduction to problem kernel, a very elementary idea is as follows: Assume that you have a graph vertex v of degree $k+1$, that is, $k+1$ edges have endpoint v. Then, to cover all these edges, one has to either bring v or all its neighbors into the vertex cover. In the latter case, however, the vertex cover then would have size at least $k+1$—too much if we are only interested in covers of size at most k. Hence, without branching we *have* to bring v and all other vertices of degree greater than k into the cover. From this observation, one can easily conclude that after doing this preprocessing, the remaining graph may consist of at most k^2+k vertices and at most k^2 edges. A well-known theorem of Nemhauser and Trotter [199] (also see, e.g., [32, 217]) can be used to construct an improved reduction to problem kernel resulting in a graph of only $2k$ vertices [60] as will be discussed in Section 2.4.

In the following subsections, we shed some light on the opportunities fixed-parameter complexity studies offer. Vertex Cover is the illustrating example.

1.4.1 Parameterize

In our discussions of Vertex Cover so far, the parameter k always denoted the size of the vertex cover set to be found. But this is not necessarily the only way how the parameter could have been chosen—it is probably the most natural way, though. Let us discuss some other possible choices for what we term the *parameter*.

As a first choice, consider the "dual" parameterization $n-k$, that is, now the question is whether or not there is a vertex cover of size $n-k$, where n denotes the total number of graph vertices and k again denotes a nonnegative integer given as part of the input. It is a well-known fact that a graph has a vertex cover of size k iff it has an independent set of size $n-k$. Hence, the question whether or not a graph has a vertex cover of size $n-k$ then is equivalent to the question whether or not a graph has an independent set of size k. The latter problem, however, is known to be $W[1]$-complete [88] and, thus, unlikely to be fixed-parameter tractable. In this sense, therefore, one can say that the "parameterization" with $n-k$ is "hopeless" with respect to

efficient fixed-parameter algorithms (the explicit parameter value still being k which then denotes the number of vertices *not* being part of the vertex cover) in our sense. There is also another subtle point brought up by this discussion, namely the distinction between *minimization* problems (such as Vertex Cover) and *maximization* problems (such as Independent Set) which might also influence the choice of the parameterization to be applied to the given problem. In several settings, in order to guarantee a small parameter value (and, in particular, often for maximization problems) it might make sense to choose the parameterization $n - k$ because then k might be small whereas $n - k$ is not.

Another parameterization is the following where we restrict, for the time being, our attention to planar graphs. Vertex Cover is *NP*-complete also for planar graphs [119]. For planar graphs we have the famous four-color theorem [15, 16, 229]—every planar graph has a coloring (that is, neighboring vertices have to have different colors) using only four colors. Moreover, there is a polynomial time algorithm for constructing a four-coloring [228]. From this result, however, it easily follows that every planar graph has a vertex cover of size at most $\lfloor 3n/4 \rfloor$. Hence, the question whether or not a given planar graph possesses a vertex cover of size $\lfloor 3n/4 \rfloor - k$ comes up naturally (again, k is a given nonnegative integer and n is the total number of vertices). To the best of our knowledge, however, it is open whether Vertex Cover with this parameterization is fixed-parameter tractable or whether it is $W[1]$-hard or something in-between. This way of parameterizing problems is known as "parameterizing above (respectively below) guaranteed values" (cf. Subsection 1.5.2 and [186]).

In summary, however, it needs to be emphasized that Vertex Cover has natural and obvious, practically relevant parameterizations. In Section 1.5 we will see that for other problems the choice of what to consider as the parameter may become a more complex issue.

1.4.2 Specialize

Already in the previous subsection we encountered Vertex Cover on *planar* graphs, a still *NP*-complete special case of the general problem on unrestricted graphs. Due to results in approximation algorithms, where Vertex Cover on planar graphs can be better approximated than in the general case [24][7], one might expect that it easier to cope with it on planar graphs. And, indeed, this is the case. There are time $O(c^{\sqrt{k}} + kn)$ algorithms for

[7] There are linear time algorithms giving approximation factor 2 (i.e., approximation ratio 1/2) for the unweighted case [119] (as considered here) as well as for the weighted case [31] of Vertex Cover on general graphs. Both results can be improved to an approximation factor that is asymptotically better: $2 - \log\log|V|/2\log|V|$ [32, 192]. Until now, no further improvements of these bounds have been obtained. By way of contrast, Baker [24] gave a polynomial time approximation scheme (PTAS) for Vertex Cover on planar graphs.

VERTEX COVER on planar graphs [3, 7, 8], where the currently best value for c is $2^{4\sqrt{3}}$. This is an (asymptotic) exponential speedup in comparison with the best known bound $O(1.3^k + kn)$ for general graphs. Here, it is important to notice the huge difference between the exponential base values $c = 2^{4\sqrt{3}} \approx 121.8$ and 1.3, which, at first sight, makes this result a purely theoretical one. Recent experimental studies [4], however, reveal that for the mathematically proven constant c there is probably quite some room for improvement and/or the average-case behavior of the corresponding algorithm is much better. Clearly, this raises the general issue of investigating VERTEX COVER on various special graph classes (cf. [44] for a survey on graph classes); very recent examples of this, building on the works [3, 8] are given in [81, 115].

1.4.3 Generalize

There are many, more general "relatives" of VERTEX COVER. The perhaps closest ones are WEIGHTED VERTEX COVER problems, where we put, for example, nonnegative integer or real weights on the graph vertices. Then, the task is to search for a minimum weight vertex cover set, where we sum up the weights of the vertices in the set. Recent studies also give efficient fixed-parameter search tree algorithms for WEIGHTED VERTEX COVER problems, but, in comparison with unweighted VERTEX COVER, completely new branching strategies had to be designed (cf. Subsection 5.2.1 and [207]). Very recently, a still more general version, so-called CAPACITATED VERTEX COVER with applications in drug design has been considered [142]. This problem still lacks fixed-parameter complexity studies, whereas a polynomial time factor 2 approximation algorithm has been given.

Another way of generalization is to consider a broader graph concept, that is, *hypergraphs*. By way of contrast to standard graphs, here edges may connect more than two vertices. For instance, if we allow three vertices (instead of two) per edge, then VERTEX COVER becomes the so-called 3-HITTING SET problem. Again, developing efficient fixed-parameter algorithms for 3-HITTING SET requires new ideas (cf. Section 3.4 and [208]).

Nishimura *et al.* [209] derived fixed-parameter tractability results that can also be seen as generalizations of the algorithms designed for VERTEX COVER. For instance, they consider classes of graphs which are defined through a base graph class. Then, a graph is in such a graph class if a base graph can be generated from it by deleting at most k vertices. In particular, if the base graph class consists of all edgeless graphs, then this exactly yields the class of graphs with a vertex cover of size at most k. Thus, a recognition algorithm for this graph class yields an algorithm for the parameterized VERTEX COVER problem. Unfortunately, so far these results seem to be of mainly theoretical interest only, because the exponential functions involved (e.g., k^k and worse) in the running times are too big for most practical purposes.

Studying *NP*-complete variants of Vertex Cover on bipartite graphs is motivated by applications in reconfigurable VLSI design [171, 111]. Here, when modeling the application problem, one ends up with the task of minimizing *two* vertex cover sets, and, thus, one has to deal with two parameters, both being nonnegative integers. There also are fixed-parameter algorithms for these modifications of Vertex Cover (cf. Subsection 5.2.2 and [58, 111]).

1.4.4 Count or Enumerate

So far, parameterized complexity has been focusing on the consideration of decision or search problems. Counting problems, being standard in classical computational complexity, have largely been neglected. Consider the graph consisting of n vertices and $n/2$ vertex-disjoint edges. Clearly, an optimal vertex cover of this graph has size $k = n/2$ and there are exactly 2^k many optimal vertex covers. Generally, it may become difficult to exactly count the number of optimal vertex covers of a given graph. The problem is that the common search tree approaches to solve Vertex Cover in a fixed-parameter way have to be extended such that it is made sure that not the same vertex cover set is generated through two different paths through the search tree and thus to avoid that is counted twice. Rossmanith [234] reports on a fixed-parameter search tree strategy for exactly counting vertex covers that runs in time $O(1.47^k k^2 + m + n)$ for a graph with n vertices and m edges. Also see [122] for a non-parameterized counting algorithm for Vertex Cover.

More general considerations of parameterized counting, in particular including hardness results, have recently been obtained by [19, 113, 188]. Specifically, a whole complexity theory of parameterized counting has been initiated.

If one wants to enumerate all optimal vertex covers of a given graph (let an optimal vertex cover have size k) then clearly the trivial 2^k search tree is optimal because, again considering the example graph from above, there are 2^k optimal vertex covers of this graph and 2^k is an upper bound for the number of optimal vertex covers of every graph. Refer to [110] for further observations concerning parameterized enumeration and refer to [96, 122] for expositions explaining the practical usefulness of counting and enumeration problems and also pointing out some concrete computational problems and difficulties.

1.4.5 Lower-Bound

Giving (relative) lower bounds for the computational complexity of problems is a core challenge in theoretical computer science. Unfortunately, it is also a very hard problem with relatively little success so far. Nevertheless, it is worth pursuing this issue also in the context of fixed-parameter algorithms. Besides the "relative lower bounds" given by the mentioned parameterized

completeness program as such (e.g., $W[1]$-hardness of INDEPENDENT SET for parameter k (number of vertices in the independent set) and, equivalently, $W[1]$-hardness of VERTEX COVER with respect to parameter k when it is asked whether there is a vertex cover of size $n - k$), for instance, also the following two questions merit attentiveness:

- Can VERTEX COVER on general graphs be solved in time $(1 + \epsilon)^k \cdot n^{O(1)}$ for arbitrary $\epsilon > 0$ or is there a minimum ϵ-value?
- Can VERTEX COVER on general graphs be solved in time $2^{o(k)} \cdot n^{O(1)}$?

Cai and Juedes [53] negatively answered the second question by stating (among other things) that VERTEX COVER cannot be solved in time $2^{o(k)} \cdot n^{O(1)}$ unless 3-SATISFIABILITY can be solved in time $2^{o(n)}$, where n is the number of variables. Note that the currently best deterministic algorithm for 3-SATISFIABILITY runs in time $1.481^n \cdot n^{O(1)}$ [76]. By way of contrast, the first question appears to be completely open.

1.4.6 Implement

All we discussed up to now are concerns with strong theoretical flavor. The value of all algorithms mentioned with their proven worst-case performance bounds only may indicate their practical usefulness. Fixed-parameter algorithms for VERTEX COVER (enriched with heuristic strategies) and for VERTEX COVER on planar graphs have actually been implemented and tested [79, 234, 248, 4, 85]. The results are encouraging but this field as a whole is still in its infancy, VERTEX COVER being one of the rare examples with some practical experiences. Usually a lot of algorithm engineering is necessary to turn a theoretically efficient algorithm into a practically useful tool. In particular, it appears to be obvious that the theoretically best search tree algorithms for VERTEX COVER [60, 204] which are based on highly complicated case distinctions need to be simplified for practical applications. The administrative (and intellectual) overhead caused by these fine-grained case distinctions does not seem to pay off in practice. Hence, the question of "re-engineering" these case distinctions as much as possible arises where the challenge is to simplify the case distinctions as much as possible and, at the same time, to deteriorate the (worst-case) bounds on the search tree size as little as possible.

1.4.7 Apply

The last subsection, which we consciously separated from the last but one point "Implement it," deals with a very subtle point. It is still comparatively easy in the academic setting to get your algorithms implemented and to test them on some "toy examples," e.g., random graphs. It may become a significantly more time-consuming and challenging task to experiment with "real

data," i.e., graphs derived from real world applications together with solving practically relevant problems. Observe that, especially for exponential time algorithms as we deal with, final "fine tuning" is a must in order to make them running as fast as possible. This fine tuning should be directed towards the final application behind. For pretty general problems such as Vertex Cover, it is to be expected that this is anything but easy to do. Also, then, there should be a serious comparison with possibly existing, different approaches (especially with those of putative "practical people"). The task to really find for a somewhat academic problem such as Vertex Cover (although there are nice textbook examples for applications) a real-life, useful, industrial-level application is a challenge of high importance. This topic deserves more attention than it currently obtains.

1.4.8 Final Remarks

Compared with other parameterized problems, the fixed-parameter complexity of Vertex Cover is well understood. Now, there are efficient fixed-parameter algorithms available and two major challenges here are

1. to give them a really practical meaning as discussed in Subsection 1.4.7, and
2. to further see what improvements in the running time (theoretically as well as practically (heuristically?)) are still possible and what lower bounds apply.

As Vertex Cover developed into a core problem of fixed-parameter algorithmics, it would be good to have some benchmark instances available where new algorithms can be tested and tuned. Also, modified and generalized Vertex Cover problems, in particular those arising in practice, need to be dealt with. The fixed-parameter history of Vertex Cover has not yet come to its end.

1.5 How to Parameterize

Already for Vertex Cover we have seen that there is usually more than one natural way to choose the parameterization of a problem. Still, for Vertex Cover the case was relatively clear. As we will see now, in general it is not always straightforward to find the "right" parameterization for a problem—as a matter of fact, several reasonable, equally valid parameters to choose may exist and it often depends on the application behind or additional knowledge about the problem which parameterization is to be preferred. In what follows, we try to collect a kind of a check list of questions to ask when confronted with a new problem and the task to develop an exact, fixed-parameter algorithm to solve it.

1.5.1 Parameter Really Small?

Recalling our running example Vertex Cover, the parameter "size of the vertex cover set" appeared as natural choice, and, for general graphs, it seems plausible in many cases to assume that the size of the vertex cover set is significantly smaller than the total number of graph vertices. Hence, this can be considered as a useful parameterization of Vertex Cover. The situation changes when we turn our attention to the special case of Vertex Cover on planar graphs. Observing the fact that planar graphs with n vertices have at most $3n - 6$ edges, it is no longer clear that the vertex cover sizes for planar graphs are "small" compared to the total number of graph vertices. And, indeed, in experiments with combinatorial random planar graphs we observed that minimum size vertex covers very often contained about half of all graph vertices [4].

A second example, drawn from computational biology and genome rearrangements, is given by the Breakpoint Median problem (cf. Subsection 5.1.2 and [236]):

> **Input:** Signed permutations $\pi_1, \pi_2, \ldots, \pi_k$ on n elements and a nonnegative integer d.
> **Question:** Is there a signed permutation $\hat{\pi}$ such that $\sum_{i=1}^{k} d_{bp}(\pi_i, \hat{\pi}) \leq d$?

Herein, $d_{bp}(\pi_i, \hat{\pi})$ denotes the *breakpoint distance* between permutations π_i and $\hat{\pi}$, see Subsection 5.1.2 for definitions. Breakpoint Median is *NP*-complete, and remains so in the case of only three input permutations [46, 218]. Hence, choosing k as parameter does not make sense. The problem is fixed-parameter tractable with respect to the distance parameter d. More precisely, there is an algorithm solving Breakpoint Median in time $O(2.15^d \cdot kn)$, which is practical when d is not too large (as demonstrated by experiments), a reasonable assumption in applications [135]. Notably, with increasing k, the base of the exponential term becomes smaller and smaller [135]. By way of contrast, however, with increasing k also the value of distance parameter d should increase, because we are using a "sum metrics." It is clear that with large enough k and, thus, many summation terms, it is no longer reasonable to assume small values for distance parameter d. One way to deal with this may be to to consider a "normalized parameter" d/k instead of d, thus making the maximally allowed distance value dependent on the number of input permutations. It is not clear whether or not Breakpoint Median is fixed-parameter tractable with respect to parameter d/k. From a practical point of view, however, the fact that the exponential base also depends on k and decreases with increasing k still makes the above mentioned algorithm valuable. In summary, we observe that increasing parameter values with increasing input size have to be taken into account. Normalization might become an important (but difficult) issue in future fixed-parameter complexity studies.

1.5.2 Guaranteed Parameter Value?

This issue is closely related to the previous point. Once more reconsider Vertex Cover on planar graphs. In Subsection 1.4.1 we noted that no minimum vertex cover can contain more than $\lfloor 3n/4 \rfloor$ of all n graph vertices due to the four-color theorem. For the "dual problem" of Vertex Cover, Independent Set, on planar graphs this means that every maximum independent set contains at least $\lceil n/4 \rceil$ vertices. We have the *guaranteed value* $\lceil n/4 \rceil$. Hence, the at first sight natural parameter "size of the independent set" is not to be considered as really small. There is an alternative parameterization, which makes sense: Find an independent set of size $\lceil n/4 \rceil + k$. Unfortunately, the fixed-parameter complexity of this problem is open. This alternative problem formulation for Independent Set on planar graphs is the perhaps best example for "parameterizing above guaranteed values," introduced by Mahajan and Raman [186]. They focused on the Maximum Satisfiability and the Maximum Cut problems and give similar, somewhat "weaker"[8] guaranteed values for these problems.

Two further problems with guaranteed values again appear in the computational biology context. The Minimum Quartet Inconsistency problem appears in the construction of evolutionary trees (cf. Subsection 5.1.1 and [133]):

> **Input:** A set S of n taxa and a set Q_S of $\binom{n}{4}$ quartet topologies such that there is exactly one topology for *every* quartet corresponding to S and a nonnegative integer k.
> **Question:** Is there an evolutionary tree T where the leaves are bijectively labeled by the elements from S such that the set of quartet topologies induced by T differs from Q_S in at most k quartet topologies?

Here, a *quartet* is a size four subset $\{a, b, c, d\}$ of the set of taxa and the *quartet topology* for $\{a, b, c, d\}$ induced by T simply is the four leaves subtree of T for $\{a, b, c, d\}$ [133]. Minimum Quartet Inconsistency is *NP*-complete [35, 157]. Steel [246] pointed out that the quartet cleaning algorithm by Berry *et al.* [35] finds the optimal solution for instances with $k < (n-3)/2$. Thus, we have the guaranteed value $(n-3)/2$.

The Betweenness problem [211, 65] consists of a finite set of n elements (or *points*) $S = \{x_1, \ldots, x_n\}$, and a finite set of m constraints. Each constraint consists of a triplet $(x_i, x_j, x_k) \in S \times S \times S$. A candidate solution to Betweenness is a total order $<$ on its points. A total order $x_{i_1} < x_{i_2} < \ldots < x_{i_n}$ satisfies the constraint (x_i, x_j, x_k) if either $x_i < x_j < x_k$ or $x_k < x_j < x_i$.

[8] The problem there is that it is not always clear what should be taken as a guaranteed value. For example, in case of Maximum Satisfiability by a simple argument it is known that at least half of all clauses can be satisfied. Thus, this can be taken as a guaranteed value (as it is done in [186]). There exist, however, larger guaranteed bounds (cf. [186]).

That is, each constraint forces the second variable x_j to be between the two other variables x_i and x_k, but does not specify the relative order of x_i and x_k. The decision version of BETWEENNESS is to decide if all constraints can be simultaneously satisfied by a total order of the variables. The optimization version of BETWEENNESS is to find a total ordering satisfying the maximum number of constraints. Opatrny [211] showed that the decision version of BETWEENNESS is *NP*–complete. This problem arises when analyzing certain mapping problems in molecular biology. For example, it occurs when trying to order markers on a chromosome, given the results of a radiation hybrid experiment [72, 124]. It is easy to find an assignment satisfying $\lceil m/3 \rceil$ out of the m constraints. What is the situation with respect to satisfying $\lceil m/3 \rceil + k$ constraints? Is the corresponding problem with parameter k fixed-parameter tractable?

In summary, guaranteed bounds for parameter values should—whenever available—be taken into account when designing fixed-parameter algorithms. The particular difficulty herein is that to prove these bounds can be very hard, and, additionally, it is not always clear whether these bounds are optimal and, therefore, better guaranteed bounds might exist.

1.5.3 More Than One Reasonable Parameterization?

Many problems naturally offer a whole selection of possible parameterizations and some parameterizations may make the design of a fixed-parameter algorithm "easy" and some may make it "hard." Moreover, different application settings and different side conditions arising in practice may require different parameterizations.

Firstly, we turn to VERTEX COVER but now the weighted case (cf. Subsection 1.4.3). With positive weights on the vertices, we asked for a minimum weight vertex cover, the parameter k denoting the total weight of the vertex cover set. In most interesting cases, there are fixed-parameter algorithms solving this problem [207]. The situation changes, however, when k denotes the number of vertices of a vertex cover of minimum weight. It is open whether WEIGHTED VERTEX COVER is fixed-parameter tractable with respect to this parameterization.

A second, in flavor significantly different example is given by the *NP*-complete CLOSEST STRING (or, equivalently, CONSENSUS STRING or CENTER STRING) problem (cf. Section 3.3 and [137]):[9]

> **Input:** k strings $s_1, s_2, \ldots, s_k$ over alphabet Σ of length L each, and a nonnegative integer d.
> **Question:** Is there a string s such that $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$?

[9] From a linguistic point of view, a "closest" string would only mean a string with minimum Hamming distance to the given strings. Beyond that, we use the term "closest" string here for a string which has Hamming distance at most d to all given strings.

Here, $d_H(s, s_i)$ denotes the Hamming distance between strings s and s_i. Consider the two most natural parameters of Closest String: the maximum Hamming distance d allowed and the number k of given input strings. Under the natural assumption that either d or k is (very) small (in particular, in biological applications it is appropriate to assume small d, e.g., $d < 10$ [100, 174]), it is important to ask whether efficient polynomial or even better linear time algorithms are possible when d or k are constants. Put in slightly more general terms, this is the question for the fixed-parameter tractability of these problems. Closest String can be solved in time $O(d^d \cdot kd + kL)$, yielding a linear time search tree algorithm for constant d (cf. Section 3.3 and [137]). Using an *integer linear program* formulation, we can observe that the problem is fixed-parameter tractable with respect to k—the exponential term in k is huge, though (cf. Section 4.1 and [137]). As we see, there are two parameters with completely different fixed-parameter algorithms—the application behind has to decide which one is to be preferred. In this particular case, so far, the parameterization with d seems to be the first choice in most cases [136, 137].

1.5.4 Several Parameters at the Same Time?

If one has to design and analyze a fixed-parameter algorithm involving more than one parameter at the same time, things might become easier as well as they might become more difficult. Two examples, one for each case, follow.

A close relative of Vertex Cover is Constraint Bipartite Vertex Cover (cf. Subsection 5.2.2 and [111]):

> **Input:** A bipartite graph $G = (V_1, V_2, E)$ and two nonnegative integers k_1 and k_2.
> **Question:** Are there two subsets $C_1 \subseteq V_1$ and $C_2 \subseteq V_2$ of sizes $|C_1| \leq k_1$ and $|C_2| \leq k_2$ such that each edge in E has at least one endpoint in $C_1 \cup C_2$?

The existence of *two* parameters and two vertex sets makes this problem quite different from the original Vertex Cover problem. Thus, whereas the classical Vertex Cover (with only one parameter!) restricted to bipartite graphs is solvable in polynomial time (because it is equivalent to a polynomial time solvable maximal matching problem), by a reduction from Clique it has been shown that Constraint Bipartite Vertex Cover is *NP*-complete [171]. The fact that the problem in a sense requires the "minimization with respect to two parameters" seemingly makes the problem harder. It can be solved in time $O(1.40^{k_1+k_2} + (k_1 + k_2)n)$ but it is conjectured that, due to its different combinatorial structure in comparison with Vertex Cover, it should be very hard to get an exponential base close to the one there (which is 1.29) [111].

On the contrary, looking back to Closest String before, it is clear that, in principle, it is easier to design an algorithm with running time $f_1(k, d) \cdot$

$n^{O(1)}$ than to give one with running time only $f_2(k) \cdot n^{O(1)}$ or $f_3(d) \cdot n^{O(1)}$ with arbitrary functions f_1, f_2, and f_3.[10]

1.5.5 Implicit Parameters?

For graph problems, there is a famous "implicit" parameter: *treewidth* [230]. The essential point here is that, intuitively speaking, treewidth measures how tree-like a given graph is (see Section 4.4 for details and definitions). Thus, if for a graph there is a tree decomposition with small width k, then many otherwise hard graph problems can be solved efficiently. More precisely, on a graph with a tree decomposition of width k Vertex Cover can be solved in time $O(2^k n)$ and the also *NP*-complete Dominating Set problem can be solved in time $O(4^k n)$ [12], where n is the number of nodes of the tree decomposition (cf. Section 4.5 and [3, 12]). The important thing here to note is that in the above algorithms the parameter "treewidth" is not given explicitly as part of the input, but it is "implicitly hidden" in the given graph (and does not depend on the concrete problem (such as Vertex Cover) we want to solve on it). For graphs, numerous other implicit parameters are known, see, e.g., [44]. Observe, however, that it is often computationally expensive to determine these implicit parameters. As to treewidth and tree decompositions, Bodlaender [38] gave a celebrated linear time algorithm when the treewidth k is a fixed constant—with a hidden constant factor exponential in k (i.e., c^{k^3} for some constant c) that still seems too large for practical purposes. That is why also heuristic approaches for constructing tree decompositions are in use, see [168] for an up-to-date account. Bodlaender's algorithm is a fixed-parameter algorithm for the *NP*-complete Treewidth problem. It is a matter of current research to improve this fixed-parameter algorithm, i.e., to improve the term exponential in k.

As a second example, recall the Betweenness problem from Subsection 1.5.2. A computational task of practical significance in this context is to find a total ordering of the x_i that maximizes the number of satisfied constraints. Indeed, Betweenness is central in the software package RHMAPPER [243, 250]. At the heart of this package is a method for producing the order of *framework markers* based on betweenness constraints (obtained from a statistical analysis of the biological data). Slonim *et. al.* [243, 250] successfully employ two greedy heuristics for solving the betweenness problem. These heuristics perform quite well on real biological data, where presumably the number of bad constraints is small. However, Chor and Sudan [65] have shown that the problem is *MaxSNP*-complete. Therefore, there is some $\epsilon > 0$ such that finding a total order which satisfies at least $m(1 - \epsilon)$ of the constraints (even if they are all satisfiable) is *NP*–hard. It is suspected that the success of

[10] Interestingly, to the best of the author's knowledge, for Closest String only a size $k \cdot d$ *problem kernel* (cf. Chapter 2 and Lemma 3.3.1, Section 3.3) but no problem kernel with size only depending on d or k alone is known.

the greedy heuristics depends on additional structure, a "hidden parameter" that should be investigated.

1.5.6 Final Remarks

In conclusion, parameters implicitly hidden in the input may provide an alternative approach to cope with generally hard computational problems. As we have seen, there are numerous possibilities but also decisions to be made when attacking a problem the fixed-parameter way. This wealth of opportunities concerning the parameterization, in some sense, also makes things more complicated. That is, generally one probably may not have *the* fixed-parameter algorithm for a problem because, at the same time, several alternative parameterizations of the same problem may make sense, being incomparable to each others. This stands in contrast to approximation algorithms where the approximation ratio is uniquely defined by the optimization goal whereas the value to optimize usually is *one* possible parameterization of the given problem (e.g., cf. Closest String in Subsection 1.5.3 where the distance value d is the optimization goal but k, the number of input strings, yields an equally valid parameterization). Perhaps, this also makes it difficult to obtain a clean, unified (complexity) theory covering all that. Thus, in fixed-parameter complexity it may get harder to keep track of what is going on in the broad sense. But to cope with computational intractability is not an easy game to play. A flexible response as offered by parameterized complexity is a worthwhile opportunity to take into consideration.

2. Problem Kernels—Data Reduction by Preprocessing

If a computationally hard problem has to be solved in practice, one of the first things usually done is to try to perform a reduction of the size of the input data. Many input instances have the property that they consist of some parts that are relatively easy to cope with and other parts that form the "really hard" core of the problem. Hence, before starting a cost-intensive algorithm to solve the difficult problem, a preprocessing phase (which, as a rule, makes use of problem-specific properties) is executed in order to shrink the given input data as much (and as fast) as possible. Weihe [262, 263] gave a striking example when dealing with the *NP*-complete Red/Blue Dominating Set problem appearing in context of the European railroad network. In a preprocessing phase, he applied two simple data reduction rules again and again until no further application was possible. The impressive result of his empirical study was that each of his real-world instances was broken into very small pieces such that for each of these a simple brute-force approach was sufficient to solve the hard problems efficiently and optimally. Observe the "universal importance" of data reduction by preprocessing. It is not only an ubiquitous topic for the design of efficient fixed-parameter algorithms, but it is of same importance for basically *any* method (such as approximation or purely heuristic algorithms) that tries to cope with hard problems. Refer to [49, 181] for related considerations concerning "off-line preprocessing" of intractable problems (with a special emphasis on computational problems from artificial intelligence and related fields).

We start with two easy examples for such an efficient preprocessing. Firstly, consider the classic *NP*-complete Satisfiability problem, where one is given a Boolean formula F in conjunctive normal form and the task is to decide whether or not F has a satisfying truth assignment. Clearly, if there are clauses consisting of only one literal (i.e., a negated or a non-negated Boolean variable) then to satisfy F one has to satisfy these "unit-clauses" by setting the value of the corresponding variable accordingly. There is no choice here. This (and other simple rules such as, e.g., the well-known pure literal rule, cf. Section 3.5) can be done in a simple preprocessing phase, and it may shrink the original input formula considerably, resulting in the combinatorially hard core formula.

Secondly, let us return to our running example VERTEX COVER: It clearly is permissible to remove isolated vertices, i.e., vertices with no adjacent edges. Moreover, if (as usual) one is looking for only *one* optimal vertex cover and *not all* of them, then vertices with only one adjacent edge and, thus, one adjacent vertex, can easily be dealt with by putting the neighboring vertex into the cover. This is correct because in order to cover the corresponding edge one of the both endpoints *has* to be in the vertex cover. Finally, in the fixed-parameter setting, where we ask for a vertex cover of size at most k, we can further do the following. If there is a vertex of degree at least $k+1$, that is, a vertex with more than k adjacent edges, then, if a vertex cover of size k exists this particular vertex has to be part of it. Otherwise, to cover all its adjacent edges would require all its at least $k+1$ neighbors, a contradiction. This is known as Buss' *reduction to problem kernel* for VERTEX COVER (cf. [48, 88]). One can easily see that after performing the above rules, in order to have a VERTEX COVER of size at most k, the remaining graph can have at most k^2+k vertices and at most $2k^2$ edges. For the time being, however, our sole point of interest here is that all three above rules (concerning isolated vertices, vertices of degree one and of degree at least $k+1$) can be applied in an easy and efficient polynomial time preprocessing phase. Thus, we may obtain a data reduction by preprocessing. In Section 2.4, we will see that VERTEX COVER even allows for a much more sophisticated and stronger data reduction, based on a theorem of Nemhauser and Trotter [199].

In summary, data reduction by preprocessing is a topic with practical importance that cannot be overrated and it belongs as an important key technique in *every* algorithm designer's tool-box. One might say that it is still under-represented in theoretical studies and the conjecture is that it will become a growing field of research on its own. In the context of fixed-parameter algorithms, this basically comes down to what is called *reduction to problem kernel*. We formally introduce this notion in the following section.

2.1 Formal Definition

In its most general form needed in this work, reduction to problem kernel, also simply referred to as *kernelization*, can be defined as follows.

Definition 2.1.1. *Let $\mathcal{L}$ be a parameterized problem, i.e., $\mathcal{L}$ consists of pairs (I,k), where problem instance I is asked to have a solution of size k (the parameter).* Reduction to problem kernel *then means to replace instance (I,k) by a "reduced" instance (I',k') (called* problem kernel*) such that*

$$k' \leq c \cdot k, \qquad |I'| \leq g(k)$$

with a constant c,[1] *some function g only depending on k, and*

[1] Usually, $c \leq 1$. In general, it would even be allowed that $k' = f(k)$ for some arbitrary function f. For our purposes, however, we fixed that k and k' are

$$(I, k) \in \mathcal{L} \text{ iff } (I', k') \in \mathcal{L}.$$

Furthermore, we require that the reduction from (I, k) *to* (I', k') *is computable in polynomial time* $T_K(|I|, k)$.[2] *The* size *of the problem kernel is bounded by* $g(k)$.

Often, the best one can hope for is that a problem kernel has size linear in k, a so-called *linear problem kernel*. For instance, using a theorem of Nemhauser and Trotter [199], (also cf. [32, 217]), Chen *et al.* [60] recently observed a problem kernel of size $2k$ for Vertex Cover (see Section 2.4).

Clearly, the aforementioned kernelization for Vertex Cover due to Buss fits into the given definition. To get further acquainted with (also the pitfalls) of this concept, now let us have a brief look at a somewhat strange kind of problem kernelization. Consider Independent Set on planar graphs.

> **Input:** A planar graph $G = (V, E)$ and a nonnegative integer k.
> **Question:** Does G have an independent set $V' \subseteq V$ with $|V'| \geq k$, that is, a set of at least k vertices that are pairwisely nonadjacent?

Independent Set restricted to planar graphs has a problem kernel consisting of only $4k$ vertices: Due to the four-color theorem [15, 16, 229] and the corresponding polynomial-time coloring algorithm by Robertson *et al.* [228] one can color the vertices of a given planar graph such that no two neighboring vertices possess the same color. Hence, each "color class" forms an independent set of the graph, and, since we only have four color classes, one of them has to contain at least one fourth of all vertices. Thus, the reduction to problem kernel preprocessing may simply proceed as follows: If for the given input instance (a planar graph with n vertices and parameter k) it holds that $k \leq \lceil n/4 \rceil$, then answer "yes" and produce an independent set using the polynomial time coloring algorithm of Robertson *et al.* If $k > \lceil n/4 \rceil$, then $n < 4k$ and, voilà, we have a size $4k$ problem kernel. On the one hand, in a way, this kind of kernelization is not satisfactory because in the second case we did not reduce the size of the input graph at all, but simply made indirectly the observation that it has to have a big (!) independent set that contains at least one quarter of all graph vertices. This contradicts the common assumption of parameters being "small" and turns the attention to the then more natural parameterization above the guaranteed value $\lceil n/4 \rceil$, see Subsection 1.5.2. On the other hand, this example also shows that there may be deep theory behind the construction of problem kernels, here the famous four-color theorem [15, 16, 229] and the corresponding intricate coloring algorithm [228].

linearly related. We are not aware of a concrete, natural parameterized problem with problem kernel where this is not the case.

[2] Again, one could allow for a more general definition here (i.e., allowing even a time $f(k) \cdot n^{O(1)}$ for some arbitrary function f only depending on k), but we are not aware of a non-polynomial time problem kernelization.

Finally, let us mention in passing that in parameterized complexity theory it has become a commonplace that "every fixed-parameter tractable problem is kernelizable." Firstly, note that it is obvious that if there is a reduction to problem kernel for a decidable parameterized problem, then it is fixed-parameter tractable: Simply perform a brute-force search algorithm on the remaining problem kernel. The opposite direction is less obvious, but also not hard: Assume that the given fixed-parameter algorithm has running time $f(k)\cdot n^d$ for some constant d. The idea is to run this algorithm on the problem for at most n^{d+1} steps and then to consider the two cases that either the algorithm has stopped within that time or it has not stopped. In the first case, we directly obtain a kernelization algorithm running in polynomial time n^{d+1}, which simply outputs either a trivial "no"- or a trivial "yes"-instance. In the second case, we can argue that $n < f(k)$ and thus our problem kernel is the original input instance itself. Clearly, this observation is of no practical use. As a matter of personal experience, one might say that, as a rule of thumb, it mostly seems easier to come up with a simple fixed-parameter algorithm based on bounded search trees (see Section 3) than to come up with a practical kernelization.

2.2 Maximum Satisfiability

To start with, we present a very simple reduction to problem kernel for the *NP*-complete Maximum Satisfiability (MaxSat) problem:

> **Input:** A boolean formula in conjunctive normal form consisting of K clauses and a nonnegative integer k.
> **Question:** Is there a truth assignment satisfying at least k clauses.

We represent the boolean values true and false by 1 and 0, respectively. A *truth assignment* I can be defined as a set of literals that contains no pairs of complementary literals. Then for a variable x we have $I(x) = 1$ iff $x \in I$ and $I(x) = 0$ iff $\bar{x} \in I$. We only deal with propositional formulas in conjunctive normal form. These are often represented in *clause form*, i.e., as a set of clauses, where a clause is a set of literals. We will represent formulas as *multi-sets* of sets since a formula might contain some identical clauses. For the satisfiability problem multiple clauses can be eliminated but this is of course no longer true if we are interested in the number of satisfiable clauses. The formula

$$(x \vee y \vee \bar{z}) \wedge (x \vee y \vee \bar{z}) \wedge (\bar{x} \vee z) \wedge (\bar{y} \vee z)$$

will be represented as

$$\{\{x, y, \bar{z}\}, \{x, y, \bar{z}\}, \{\bar{x}, z\}, \{\bar{y}, z\}\}.$$

Note that the outer curly brackets denote a multi-set and the inner curly brackets denote sets of literals. The *length of a clause* is its cardinality, and the *length of a formula* is the sum of the lengths of its clauses.

Suppose that we are given an input instance for MaxSat. The first simple observation is that if $k \leq \lceil K/2 \rceil$ then the desired truth assignment trivially exists: Take a random truth assignment. If it satisfies at least k clauses then we are done. Otherwise, "flipping" each bit in this truth assignment to its opposite value yields a new truth assignment that now fulfills at least k (more precisely, at least $\lceil K/2 \rceil$) clauses.

Hence, from now on we can assume that $k > \lceil K/2 \rceil$ which implies $K < 2k$. The next observation, due to Mahajan and Raman [186], gives a quadratic size problem kernel: Partition the clauses of the given formula F into long clauses (i.e., clauses containing k or more literals) and into short clauses (i.e., clauses containing less than k literals). Thus, $F = F_l \wedge F_s$, where F_l contains all long clauses and F_s contains all short clauses. Let L be the number of long clauses. If $L \geq k$ then again at least k clauses can be satisfied by picking (in the worst case) in each long clause another variable and setting its value accordingly such that the corresponding clause gets satisfied. If $L < k$ then we proceed as follows.

An important point is that now it is sufficient to concentrate attention on the new instance $(F_s, k - L)$ which is our problem kernel due to the following. First, note that (F, k) is a yes-instance of MaxSat iff $(F_s, k - L)$ is a yes-instance of MaxSat. This holds since the same reasoning as above shows that the "remaining" L large clauses can always be satisfied. This is true because to satisfy $k - L$ clauses at most $k - L$ variables (at most one variable per satisfied clause) are used and thus for the L large clauses at least $k - (k - L) = L$ variables remain to be freely set. But now the rest is easy. We have that there are $K - L \leq K$ small clauses, each containing at most k literals, and we have that $K < 2k$ as explained before. Hence, the total number of literals in F_s is bounded by $2k \cdot k = 2k^2$. This means a quadratic size problem kernel for MaxSat with respect to parameter k.

Proposition 2.2.1. *There is a problem kernel of size $O(k^2)$ for* MaxSat *and it can be found in linear time.*

Proof. The method and its correctness immediately follow from the considerations above. Concerning the running time, it is easy to verify that the determination of the small and large clauses and their respective numbers as well as the determination of an assignment satisfying all large clauses can easily be done in linear time. □

We remark that the reduction to problem kernel for MaxSat has in some sense a somewhat unsatisfactory character in a similar way as the one for Independent Set on planar graphs (see Section 2.1) had. The point is that again it is made use of a structural property that guarantees that at least half of all clauses are always satisfiable. Hence, similar to there parameterizing above guaranteed values probably would be more appropriate here (cf. Subsection 1.5.2 and [186]).

2.3 3-Hitting Set

Another simple problem kernelization is given for the 3-Hitting Set (3HS) problem, the generalization of Vertex Cover to "hypergraphs with size-three edges":

> **Input:** A collection C of subsets of size three of a finite set S and a nonnegative integer k.
> **Question:** Is there a subset $S' \subseteq S$ with $|S'| \leq k$ which allows S' contain at least one element from each subset in C?

Vertex Cover is the same as 2-Hitting Set. Following [208], we show how to reduce the original instance to a new one consisting of only $O(k^3)$ elements. Similar to Buss' reduction to problem kernel for Vertex Cover the idea is that we *must* put "high degree elements" into the hitting set.

In the following we assume that elements of S are integers between 1 and n.

Theorem 2.3.1. *There is a problem kernel of size $O(k^3)$ for* 3-Hitting Set *and it can be found in time $O(n)$.*

Proof. Firstly, let us consider two fixed elements $x, y \in S$:
Claim 1: For an instance (C, k) we can find an instance (C', k) in linear time so that $(C, k) \in$ 3HS iff $(C', k) \in$ 3HS and there can be at most k size three subsets in the collection C' that contain both x and y.
Claim 1 is seen as follows. Assume that there are more than k subsets containing x and y. Since each set appears only once in C this implies that there are more than k different "third" elements in the corresponding sets. Hence, to cover these more than k different sets with at most k elements from the base set S, we *have* to bring at least one of x and y into our hitting set S'. This means, however, that all sets containing both x and y can be replaced by the single set $\{x, y\}$. This proves Claim 1.

Next, we consider the case where there is only one fixed element $x \in S$:
Claim 2: For an instance (C, k) we can find an instance (C', k') in linear time so that $(C, k) \in$ 3HS iff $(C', k') \in$ 3HS, $k' \leq k$, and there can be at most k^2 size three subsets in the collection C that contain x.
Claim 2 is seen as follows. Assume that there are more than k^2 subsets containing x. From Claim 1 we can assume that x can occur in a subset together with another element y at most k times. Hence, if there were more than k^2 subsets containing x, these could not be covered by some $S' \subseteq S$ with $|S'| \leq k$ without taking x. Thus, x must be in S' and the corresponding sets can be deleted. This proves Claim 2.

Now, from Claim 2 we can conclude that each element x from S can occur in at most k^2 subsets in C. (Otherwise, x *had* to be in the hitting set S'.) Clearly, because $|S'| \leq k$ this means (provided that C has a hitting set of size $\leq k$) that C can consist of at most $k \cdot k^2 = k^3$ size three subsets. This is the size of the problem kernel.

Finally, we remark that we can easily count in how many sets each element occurs and throw away in linear time all elements (and their corresponding subsets) occurring in more than k^2 subsets. □

The above kernelization still is comparatively simple. The next two sections indicate that designing good kernelization algorithms may turn into a very challenging task. In particular, it is an interesting open question to investigate whether the subsequent kernelization due to Nemhauser and Trotter can be generalized to 3-Hitting Set in order to improve Theorem 2.3 and to perhaps attain a linear size problem kernel.

2.4 Vertex Cover

We already discussed the $O(k^2)$ problem kernel due to Buss, which is based on a simple observation concerning high-degree vertices (cf. the beginning of this chapter). Now, we present a much more sophisticated kernelization based on a theorem of Nemhauser and Trotter [199]. It was developed in the context of approximation algorithms and Chen *et al.* [60] were the first to point out its usefulness when designing fixed-parameter algorithms for Vertex Cover. We basically follow Bar-Yehuda and Even [32] in proving the Nemhauser-Trotter theorem [199], a core result for the kernelization field. Also see [164] for a new alternative proof.

Theorem 2.4.1. (Nemhauser-Trotter) *For an n-vertex graph $G = (V, E)$ with m edges, two disjoint sets $C_0 \subseteq V$ and $V_0 \subseteq V$ can be computed in time $O(\sqrt{n} \cdot m)$, such that the following three properties hold.*

1. *Let $D \subseteq V_0$ be a vertex cover of the subgraph $G[V_0]$. Then $C := D \cup C_0$ is a vertex cover of G.*
2. *There is a minimum vertex cover S of G with $C_0 \subseteq S$.*
3. *The subgraph $G[V_0]$ has a minimum vertex cover of size at least $|V_0|/2$.*

Proof. The algorithm given in Fig. 2.1 computes the sets C_0 and V_0.

To prove the validity of the three claims of the theorem, we still need the following definition.

$$\begin{aligned} I_0 &:= \{x \mid x \notin C_B \text{ and } x' \notin C_B\} \\ &= V - (V_0 \cup C_0). \end{aligned}$$

In the following, step by step we prove the three statements of the theorem.

$D \cup C_0$ is a vertex cover of G: For $\{x, y\} \in E$ we have to show that $x \in C$ or $y \in C$.

Case 1: If $x \in I_0$, i.e., $x, x' \notin C_B$, then it must hold $y, y' \in C_B$ and, thus, $y \in C_0$.

Case 2: The case $y \in I_0$ is completely analogous to Case 1.

Input: $G = (V, E)$
Output: C_0 and V_0.
Phase 1:
Define the bipartite graph $B = (V, V', E_B)$
where $E_B := \{\{x, y'\} \mid \{x, y\} \in E\}$
and V' is a "copy" of V.
Phase 2:
Let C_B be an optimal vertex cover of B
which can be computed by a maximum matching
in time $O(\sqrt{n}m)$.
$C_0 := \{x \mid x \in C_B \text{ and } x' \in C_B\}$.
$V_0 := \{x \mid \text{either } x \in C_B \text{ or } x' \in C_B\}$.

Fig. 2.1. An algorithm to compute the sets C_0 and V_0 of Theorem 2.4.1.

Case 3: $x \in C_0$ or $y \in C_0$ is trivial.
Case 4: If $x, y \in V_0$ then $x \in D$ or $y \in D$.

There is an optimal vertex cover S of G with $C_0 \subseteq S$: Define $S_V := S \cap V_0$, $S_C := S \cap C_0$, $S_I := S \cap I_0$ and $\bar{S}_I := I_0 - S_I$. By S'_C we denote a copy of S_C.

Claim: $C'_B := (V - \bar{S}_I) \cup S'_C$ is a vertex cover of B.

Using the Claim (which we prove afterwards), we show $|C_0| \leq |S - S_V|$ which implies $|C_0 \cup S_V| \leq |S|$. With the first point of the theorem we obtain the optimality of $C_0 \cup S_V$.

$$\begin{aligned}
|V_0| + 2|C_0| &= |V_0 \cup C_0 \cup C'_0| \\
&= |C_B| \quad \text{[Def. of } V_0 \text{ and } C_0\text{]} \\
&\leq |C'_B| \quad \text{[Optimality of } C_B\text{]} \\
&= |V - \bar{S}_I| + |S'_C| \quad \text{[Claim]} \\
&= |V_0 \cup C_0 \cup I_0 - (I_0 - S_I)| + |S'_C| \\
&= |V_0| + |C_0| + |S_I| + |S_C| \\
\Rightarrow \quad |C_0| &\leq |S_I| + |S_C| \\
&= |S_I| + |S_C| + |S_V| - |S_V| \\
&= |S| - |S_V|.
\end{aligned}$$

It remains to show the claim. Let $\{x, y'\} \in E_B$. We have to show that $x \in C'_B$ or $y' \in C'_B$.

Case 1: If $x \notin \bar{S}_I$ then $x \in C_{B_1}$ according to the definition of C'_B.
Case 2: If $x \in \bar{S}_I$ then $x \in I_0$ and, therefore, $x \notin C_0$ and $x \notin S$. Then, $y \in S$ and $y \in C_0$ (according to Case 1 in the proof of the first claim of the theorem), thus, $y \in S \cap C_0 = S_C$ and $y' \in S'_C$.

The graph induced by V_0 has a minimum vertex cover of size at least $|V_0|/2$:
Assume that S_0 is a minimum vertex cover of $G[V_0]$. According to the first claim of the theorem $C_0 \cup S_0$ is a vertex cover of G. According to the

definition of the bipartite graph B the set $C_0 \cup C_0' \cup S_0 \cup S_0'$ is a vertex cover of B. Hence:

$$\begin{aligned} |V_0| + 2|C_0| &= |C_B| \quad [\text{Def. of } V_0 \text{ and } C_0] \\ &\leq |C_0 \cup C_0' \cup S_0 \cup S_0'| \quad [C_B \text{ optimal}] \\ &= 2|C_0| + 2|S_0|. \end{aligned}$$

This implies $|V_0| \leq 2|S_0|$.

□

As observed by Chen *et al.* [60], Theorem 2.4.1 is the key to a linear size problem kernel for Vertex Cover which can be found efficiently.

Theorem 2.4.2. *Let $(G = (V, E), k)$ be an input instance of* Vertex Cover. *In time $O(k \cdot |V| + k^3)$ one can compute a reduced instance $(G' = (V', E'), k')$ with $|V'| \leq 2k$ and $k' \leq k$ such that G admits a vertex cover of size k iff G' admits a vertex cover of size k'.*

Proof. Firstly, use Buss' reduction to problem kernel to get a reduced instance containing at most k^2+k vertices and parameter $k'' \leq k$. Then, Phase 2 of the algorithm described in the proof of Theorem 2.4.1—which basically consists of a maximum matching computation in a bipartite graph then containing $O(k^2)$ vertices and $O(k^2)$ edges—can be done in time $O(k^3)$ (cf. [69]). From the maximum matching in linear time we get the set C_0 of vertices that have to be in the vertex cover (thus, $k' := k'' - |C_0|$) and the set V_0 that induces the subgraph $G[V_0]$, the problem kernel G'. Observe that due to Theorem 2.4.1 we directly know that if $|V_0| > 2k'$ then there is no vertex cover of size k of the original graph G. Otherwise, the remaining vertices for a minimum vertex cover of G can be found by searching for a minimum vertex cover of G'. □

There are several important observations to be made with respect to Theorem 2.4.2. Firstly, according to the current state of knowledge, a size $2k$ problem kernel for Vertex Cover is the best one could hope for because a problem kernel of size $(2-\varepsilon)k$ with constant $\varepsilon > 0$ would imply a factor $2-\varepsilon$ polynomial-time approximation algorithm for Vertex Cover. This, however, is a long-standing open question and an answer to it would mean a major breakthrough in approximation algorithms for Vertex Cover [152].

Secondly, in contrast to the simpler Buss kernelization, after performing the Nemhauser-Trotter reduction to problem kernel, in general we cannot guarantee to find *all* vertex covers up to size k. Theorem 2.4.1 only leads to (at least) one particular minimum vertex cover, excluding others from further consideration.

Thirdly, it is important to note that Theorem 2.4.1 can be generalized to finding minimum *weight* vertex covers, where vertices have a positive real weight (see [199, 32] for details). This is useful for fixed-parameter algorithms for Weighted Vertex Cover, see Subsection 5.2.1.

We only briefly mention that Chen *et al.* [60] used another interesting technique called "folding" (also called "struction" in the literature) to solve Vertex Cover. The point is that all degree-two vertices in a graph together with their two neighbors can be melted into one super-vertex. Thus, the "combinatorially explosive" part of the search for a (minimum) vertex cover can then be restricted to graphs of minimum degree three (see [60] for details).

Finally, it is worth noting that implementing the Nemhauser-Trotter kernelization and experimenting with random (planar) graphs showed the impressive power of this data reduction at least for planar random graphs [4, 85]. Often data reduction around 60% and more (concerning both vertices and edges) could be achieved. Moreover, the preprocessing usually detected a very high percentage (around 50–60%) of vertices that can be guaranteed to belong to a minimum vertex cover. These experiments need to be extended in order to get a better picture of the benefits of Vertex Cover kernelization in different contexts.

2.5 Dominating Set on Planar Graphs

In the previous subsection, we became acquainted with a linear size problem kernel for Vertex Cover. We know of so far only few problem kernels of linear size—a further example is the one recently developed for Dominating Set on planar graphs.

> **Input:** A planar graph $G = (V, E)$ and a nonnegative integer k.
> **Question:** Is there a choice of at most k vertices $V' \subseteq V$ such that for every vertex $v \in V$ there is either at least one vertex in V' that is a neighbor of v or $v \in V'$ (or both)?

The kernelization consists of two main parts—the algorithmic side with the actual reduction rules and the mathematical side with the proof of correctness and the analysis of the problem kernel size. Because of the significant technical machinery involved, we will essentially focus on the algorithmic side and refer to [2, 6] for more details. We follow parts of [6].

Two reduction rules are used to prove the linear size problem kernel for Dominating Set. Both reduction rules are based on the same principle: Explore the local structure of the graph and try to replace it by a simpler structure. In what follows, the minimum k such that graph G has a size k dominating set is called the *domination number* of G, denoted by $\gamma(G)$.

2.5.1 The Neighborhood of a Single Vertex

Consider a vertex $v \in V$ of the given graph $G = (V, E)$ and partition the vertices of the neighborhood $N(v)$ of v into three different sets $N_1(v)$, $N_2(v)$, and $N_3(v)$ depending on what neighborhood structure these vertices have. More precisely, setting $N[v] := N(v) \cup \{v\}$, we define

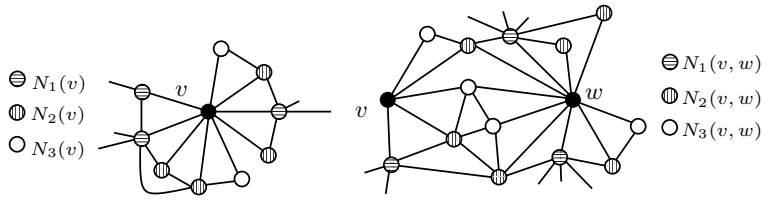

Fig. 2.2. The left-hand side shows the partitioning of the neighborhood of a single vertex v. The right-hand side shows the partitioning of a neighborhood $N(v,w)$ of two vertices v and w. Since, in the left-hand figure, $N_3(v) \neq \emptyset$, reduction Rule 1 applies. In the right-hand figure, since $N_3(v,w)$ cannot be dominated by a single vertex at all, Case 2 of Rule 2 applies.

$$N_1(v) := \{u \in N(v) \mid N(u) \setminus N[v] \neq \emptyset\},$$
$$N_2(v) := \{u \in N(v) \setminus N_1(v) \mid N(u) \cap N_1(v) \neq \emptyset\},$$
$$N_3(v) := N(v) \setminus (N_1(v) \cup N_2(v)).$$

An example which illustrates the partitioning of $N(v)$ into the subsets $N_1(v)$, $N_2(v)$, and $N_3(v)$ can be seen in the left-hand diagram of Fig. 2.2.

Based on the above definitions we give our first reduction rule.

Rule 1 *If $N_3(v) \neq \emptyset$ for some vertex v, then*

- *remove $N_2(v)$ and $N_3(v)$ from G and*
- *add a new vertex v' with the edge $\{v, v'\}$.*

Lemma 2.5.1. *Let $G = (V, E)$ be a graph and let $G' = (V', E')$ be the resulting graph after having applied Rule 1 to G. Then $\gamma(G) = \gamma(G')$.*

Proof. Consider a vertex $v \in V$ such that $N_3(v) \neq \emptyset$. The vertices in $N_3(v)$ can only be dominated by either v or by vertices in $N_2(v) \cup N_3(v)$. But, clearly, $N(w) \subseteq N(v)$ for every $w \in N_2(v) \cup N_3(v)$. This shows that an optimal way to dominate $N_3(v)$ is given by taking v into the dominating set. This is simulated by the "gadget" $\{v, v'\}$ in G'. It is safe to remove $N_2(v) \cup N_3(v)$, since these vertices need not to be used in an optimal dominating set. Hence, $\gamma(G') = \gamma(G)$. □

Lemma 2.5.2. *Rule 1 can be carried out in time $O(n)$ for planar graphs and in time $O(n^3)$ for general graphs.*

Proof. We first discuss the planar case. To carry out Rule 1, for each vertex v of the given planar graph G we have to determine the neighbor sets $N_1(v)$, $N_2(v)$, and $N_3(v)$. By definition of these sets, one easily observes that it is sufficient to consider the subgraph G that is induced by all vertices that are connected to v by a path of length at most two. To do so, we employ a search tree of depth two, rooted at v. We perform two phases.

In phase 1, constructing the search tree we determine the vertices from $N_1(v)$. Each vertex of the first level (i.e., distance one from the root v) of the search tree that has a neighbor at the second level of the search tree belongs to $N_1(v)$. Observe that it is enough to stop the expansion of a vertex from the first level as soon as its *first* neighbor in the second level is encountered. Hence, denoting the degree of v by $\deg(v)$, phase 1 takes time $O(\deg(v))$ because there clearly are at most $2 \cdot \deg(v)$ tree edges and at most $O(\deg(v))$ non-tree edges to be explored. The latter holds true since these non-tree edges all belong to the subgraph of G induced by $N[v]$. Since this graph is clearly planar and $|N[v]| = \deg(v) + 1$, the claim follows.

In phase 2, it remains to determine the sets $N_2(v)$ and $N_3(v)$. To get $N_2(v)$, one basically has to go through all vertices from the first level of the above search tree that are not already marked as being in $N_1(v)$ but have at least one neighbor in $N_1(v)$. All this can be done within the planar graph induced by $N[v]$, using the already marked $N_1(v)$-vertices, in time $O(\deg(v))$. Finally, $N_3(v)$ simply consists of vertices from the first level that are neither marked being in $N_1(v)$ nor marked being in $N_2(v)$. In summary, this shows that for vertex v the sets $N_1(v)$, $N_2(v)$, and $N_3(v)$ can be constructed in time $O(\deg(v))$.

Once having determined these three sets, the sizes of which all are bounded by $\deg(v)$, it is clear that the possible removal of vertices from $N_2(v)$ and $N_3(v)$ and the addition of a vertex and an edge as required by Rule 1 all can be done in time $O(\deg(v))$. Finally, it remains to analyze the overall complexity of this procedure when going through all n vertices of $G = (V, E)$. But this is easy. The running time can be bounded by $\sum_{v \in V} O(\deg(v))$. Since G is planar, this sum is bounded by $O(n)$, i.e., the whole reduction takes linear time.

For general graphs, the method described above leads to a worst-case cubic time implementation of Rule 1. Here, one ends up with the sum

$$\sum_{v \in V} O((\deg(v))^2) = O(n^3).$$

Note that the size of the graph that is induced by the neighborhood $N[v]$ again is relevant for the time needed to determine the sets $N_1(v)$, $N_2(v)$, and $N_3(v)$. For general graphs, this neighborhood may contain $O(\deg(v)^2)$ many vertices. □

2.5.2 The Neighborhood of a Pair of Vertices

Similar to Rule 1, we explore the set $N(v, w) := N(v) \cup N(w)$ of two vertices $v, w \in V$. Analogously, we now partition $N(v, w)$ into three disjoint subsets $N_1(v, w)$, $N_2(v, w)$, and $N_3(v, w)$. Setting $N[v, w] := N[v] \cup N[w]$, we define

$$N_1(v,w) := \{u \in N(v,w) \mid N(u) \setminus N[v,w] \neq \emptyset\},$$
$$N_2(v,w) := \{u \in N(v,w) \setminus N_1(v,w) \mid N(u) \cap N_1(v,w) \neq \emptyset\},$$
$$N_3(v,w) := N(v,w) \setminus (N_1(v,w) \cup N_2(v,w)).$$

The right-hand diagram of Fig. 2.2 shows an example which illustrates the partitioning of $N(v,w)$ into the subsets $N_1(v,w)$, $N_2(v,w)$, and $N_3(v,w)$.

Our second reduction rule—compared to Rule 1—is slightly more complicated.

Rule 2 *Consider $v, w \in V$ $(v \neq w)$ and suppose that $N_3(v,w) \neq \emptyset$. Suppose that $N_3(v,w)$ cannot be dominated by a single vertex from $N_2(v,w) \cup N_3(v,w)$.*

Case 1 If $N_3(v,w)$ can be dominated by a single vertex from $\{v,w\}$:
- *(1.1) If $N_3(v,w) \subseteq N(v)$ as well as $N_3(v,w) \subseteq N(w)$:*
  - *remove $N_3(v,w)$ and $N_2(v,w) \cap N(v) \cap N(w)$ from G and*
  - *add two new vertices z, z' and edges $\{v,z\}, \{w,z\}, \{v,z'\}, \{w,z'\}$.*
- *(1.2) If $N_3(v,w) \subseteq N(v)$ but not $N_3(v,w) \subseteq N(w)$:*
  - *remove $N_3(v,w)$ and $N_2(v,w) \cap N(v)$ from G and*
  - *add a new vertex v' and the edge $\{v,v'\}$ to G.*
- *(1.3) If $N_3(v,w) \subseteq N(w)$ but not $N_3(v,w) \subseteq N(v)$:*
  - *remove $N_3(v,w)$ and $N_2(v,w) \cap N(w)$ from G and*
  - *add a new vertex w' and the edge $\{w,w'\}$ to G.*

Case 2 If $N_3(v,w)$ cannot be dominated by a single vertex from $\{v,w\}$:
- *remove $N_3(v,w)$ and $N_2(v,w)$ from G and*
- *add two new vertices v', w' and edges $\{v,v'\}$, $\{w,w'\}$.*

Lemma 2.5.3. *Let $G = (V,E)$ be a graph and let $G' = (V',E')$ be the resulting graph after having applied Rule 2 to G. Then $\gamma(G) = \gamma(G')$.*

Proof. Similar to the proof of Lemma 2.5.1, vertices from $N_3(v,w)$ can only be dominated by vertices from $M := \{v,w\} \cup N_2(v,w) \cup N_3(v,w)$. All cases in Rule 2 are based on the fact that $N_3(v,w)$ needs to be dominated. All rules only apply if there is not a *single* vertex in $N_2(v,w) \cup N_3(v,w)$ which dominates $N_3(v,w)$.

We first of all discuss the correctness of Case (1.2) (and similarly the symmetric Case (1.3)): If v dominates $N_3(v,w)$ (and w does not), then it is better to take v into the dominating set—and at the same time still leave the option of taking vertex w—than to take any combination of two vertices $\{x,y\}$ from the set $M \setminus \{v\}$. It may be that we still have to take w to a minimum dominating set, but in any case $\{v,w\}$ dominates at least as many vertices as $\{x,y\}$. The "gadget" $\{v,v'\}$ simulates the effect of taking v. It is safe to remove $N := (N_2(v,w) \cap N(v)) \cup N_3(v,w)$ since, by taking v into the dominating set, all vertices in N are already dominated and since, as discussed above, it is always better to take $\{v,w\}$ into a minimum dominating set than to take v and any other of the vertices from N.

In the situation of Case (1.1), we can dominate $N_3(v, w)$ by both v or w. Since we cannot decide at this point which of these vertices should be chosen to be in the dominating set, we use the "gadget" with vertices v' and w' which simulates a choice between v or w, as can be seen easily. In any case, however, it is better to take one of the vertices v and w (maybe both) than taking any two of the vertices from $M \setminus \{v, w\}$. The argument for this is similar to the one for Case (1.2). The removal of $N_3(v, w) \cup (N_2(v, w) \cap N(v) \cap N(w))$ is safe by a similar argument than the one that justified the removal of N in Case (1.2).

In Case 2, we need at least two vertices to dominate $N_3(v, w)$. Since $N(v, w) \supseteq N(x, y)$ for all pairs $x, y \in M$ it is best to take v and w into the dominating set, simulated by the gadgets $\{v, v'\}$ and $\{w, w'\}$. As in the previous cases removing $N_3(v, w) \cup N_2(v, w)$ is safe since these vertices are already dominated and since these vertices need not be used for an optimal dominating set. □

Lemma 2.5.4. *Rule 2 can be carried out in time $O(n^2)$ for planar graphs and in time $O(n^4)$ for general graphs.*

Proof. To prove the time bounds for Rule 2, basically the same ideas as for Rule 1 apply (cf. proof of Lemma 2.5.2). Instead of a depth two search tree, one now has to argue on a search tree where the levels indicate the minimum of the distances to vertex v and w. Hence, we associate the vertices v and w to the root of this search tree. The first level consists of all vertices that lie in $N(v, w)$ (i.e., at distance one from either of the vertices v or w). Determining the subset $N_3(v, w)$ means to check whether some vertex on the first level has a neighbor on the second level. We do the same kind of construction as in Lemma 2.5.2. The running time again is determined by the size of the subgraph induces by the vertices that correspond to the root and the first level of this search tree, i.e., by $G[N[v, w]]$ in this case. For planar graphs, we have $|G[N[v, w]]| = O(\deg(v) + \deg(w))$. Hence, we get $\sum_{v,w \in V} O(\deg(v) + \deg(w))$ as an upper bound on the overall running time in the case of planar graphs. This is upper-bounded by

$$O(\sum_{v \in V}(n \cdot \deg(v) + \sum_{w \in V} \deg(w))) = O(n^2).$$

In case of general graphs, we have $|G[N[v, w]]| = O\left((\deg(v) + \deg(w))^2\right)$, which trivially yields the upper bound

$$\sum_{v,w \in V} O((\deg(v) + \deg(w))^2) = O(n^4)$$

for the overall running time. □

We remark that the running times given in Lemmas 2.5.2 and 2.5.4 are pure worst-case estimates and turn out to be much lower in experimental studies. The problem kernels will be obtained from "reduced" graphs.

Definition 2.5.1. *Let $G = (V, E)$ be a graph such that both the application of Rule 1 and the application of Rule 2 leave the graph unchanged. Then we say that G is* reduced *with respect to these rules.*

Observing that the (successful) application of any reduction rule always "shrinks" the given graph implies that there can only be $O(n)$ successful applications of reduction rules. This directly leads to the following.[3]

Lemma 2.5.5. *A graph G can be transformed into a reduced graph G' with $\gamma(G) = \gamma(G')$ in time $O(n^3)$ in the planar case and in time $O(n^5)$ in the general case.* □

The kernelization procedure implied by the above reduction rules allows to prove the following main result. Herein, $\gamma(G)$ denotes the size of an optimal dominating set of graph G.

Theorem 2.5.1. *For a planar graph $G = (V, E)$ which is reduced with respect to Rules 1 and 2, we get $|V| \leq 335\,\gamma(G)$, i.e., the* Dominating Set *problem on planar graphs admits a linear problem kernel.* □

To present the proof of correctness of Theorem 2.5.1 and the corresponding concrete upper bound $335\gamma(G)$ goes beyond the scope of this work. We refer to [2, 6] for a complete exposition.

Instead, we briefly report on the efficiency of the given reduction rules in practice. The performance of the preprocessing was measured on a set of combinatorial random planar graphs of various sizes. More precisely, eight sample sets of 100 random planar graphs each were created, containing instances with 100, 500, 750, 1000, 1500, 2000, 3000, and 4000 vertices. The preprocessing was, at least on the given random sample sets, to be very effective. As a general rule of thumb, one may say that, in all of the cases,

- more than 79% of the vertices and
- more than 88% of the edges

were removed from the graph. Moreover, the reduction rules determined a very high percentage (for all cases approximately 89%) of the vertices of an optimal dominating set. The overall running time for the reduction ranged from less than one second (for small graph instances with 100 vertices) to around 30 seconds (for larger graph instances with 4000 vertices) on a standard Linux PC. Moreover, it is possible to enrich the given reduction rules by further heuristics derived from the bounded search tree algorithm for Dominating Set on planar graphs (cf. Section 3.6 and [2, 5]). In this extended setting, the running times for the data reduction went down to less than half a second (for graphs of 100 vertices) and less than eight seconds (for graphs of 4000 vertices) in average. Most interestingly, the combination of these rules removed, in average,

[3] Observe that the polynomial-time bounds for the reduction rules given here are real worst-case bounds (which may not even be tight) and, in practice, the algorithms implementing these rules appear to be much faster.

- more than 99.7% of the vertices and
- more than 99.8% of the edges

of the original graph. A similarly high percentage of the vertices that belong to an optimal dominating set could be detected.

2.6 Concluding Discussion

The design and analysis of good kernelization algorithms clearly is among the most important and practically most relevant contributions to algorithm development in general. The reason for that is the ubiquitous need for efficient preprocessing procedures almost everywhere in the algorithmics for hard problems. In addition, as the Nemhauser and Trotter problem kernel for Vertex Cover exhibits, there are close connections to the theory of approximation algorithms, which should be further pursued in future investigations. With respect to fixed-parameter studies, it is worth mentioning that the *linear* size problem kernels as presented for Dominating Set on planar graphs and for Vertex Cover do imply exponential speedup for fixed-parameter algorithms for hard problems on planar graphs [7, 8] (also cf. Section 4.5 and Subsection 5.2.4). All in all, a good problem kernelization for a combinatorially hard parameterized problem is among the best and most meaningful things from a practical as well as a theoretical side that a designer of fixed-parameter algorithms can achieve.

Other problems with kernelization algorithms (clearly this is an incomplete list) to be found in the literature are Constraint Bipartite Vertex Cover [111] (also see Chen and Kanj [58] for a linear size problem kernel for a closely related problem), and Max Leaf Spanning Tree [108]. Many other known fixed-parameter tractable problems still lack non-trivial reductions to problem kernel, for instance, Minimum Quartet Inconsistency (cf. Subsection 5.1.1 and [133]) or Breakpoint Median (cf. Subsection 5.1.2 and [135]).

3. Search Trees—the Power of Systematics

A standard way to explore the huge search space of a computationally hard problem in quest for an optimal solution is to perform a systematic search. This can be organized in a tree-like fashion. In the fixed-parameter context, the depths of these search trees usually are bounded by some numbers depending on the parameter values. Bounded search trees lie at the heart of most efficient fixed-parameter algorithms today.

3.1 The Basic Idea

The basic idea behind a systematic search by way of bounded search trees is as follows: In polynomial time find a "small subset" of the input instance such that at least one element of this subset is part of an optimal solution to the problem. For instance, in case of Vertex Cover this "small subset" is a two-element set consisting of the two endpoints of an edge—one of these two vertices has to be part of the vertex cover. This leads to the previously mentioned search tree of size 2^k for Vertex Cover, where the parameter k denotes the size of the vertex cover. In case of 3-Hitting Set, we have the same observation with size-three sets. Hence, we obtain a size 3^k search tree here, where parameter k denotes the size of the hitting set. The "art" of constructing search trees lies in detecting more clever (and usually more complicated) ways of "defining" these small subsets. Both Vertex Cover and 3-Hitting Set do possess a sophisticated search tree machinery. The problematic nature of search trees, as we will see in the course of this section, may come in by the fact that many small size search trees are based on using numerous case distinctions. These case distinctions, on the one hand, require a complicated analysis and correctness proof, and, on the other hand, when implemented, they may cause administrative overhead. Thus, the theoretically best (i.e., smallest) search tree may not always be the practically best one and implementation and experiments have to decide on that.

Before we come to several examples for more or less intricate bounded search trees used in fixed-parameter algorithmics, let us first come to a very simple search tree, which, to the best of our knowledge, is the smallest in size known for that particular problem. Consider the Independent Set problem:

Input: A graph $G = (V, E)$ and a nonnegative integer k.
Question: Does G have an independent set $V' \subseteq V$ with $|V'| \geq k$, that is, a set of at least k vertices that are pairwisely nonadjacent?

For general graphs, INDEPENDENT SET is $W[1]$-complete [88]—there is no hope for fixed-parameter tractability. If INDEPENDENT SET is restricted to the class of planar graphs (where it remains *NP*-complete) the situation changes. There is an important property of planar graphs that we can make use of, namely, that in every planar graph there is at least one vertex of degree five or smaller. Using this knowledge, our search strategy for a size k independent set on planar graphs is as follows. Pick a vertex v in G which has minimum degree (bounded by five) and branch into at most six cases. Either put v into the vertex cover or one of its as most five neighbors. In each branching case, delete the corresponding vertex together with all its adjacent edges and vertices from G. Thus, obtain a smaller graph G' in each case, and recursively search for an independent set of size $k-1$ in each G'. This is correct because from the set $\{ v \} \cup N(v)$, i.e., the closed neighborhood of v, *exactly* one vertex has to be in an optimal independent set. Since the parameter in each branch decreases by one, we thus obtain a search tree of size at most 6^k. Using a suitable edge list representation of G's n vertices and m edges, picking a vertex and generating G' can easily be done in linear time $O(n+m)$. Altogether, we have:

Theorem 3.1.1. INDEPENDENT SET *on planar graphs can be solved in time* $O(6^k \cdot (n+m))$*, where* k *is the size of the independent set we search for and* n *and* m *are the numbers of vertices and edges, respectively.* □

Sometimes, such a tempting simple argument as the one for INDEPENDENT SET above can go wrong. Consider the DOMINATING SET problem on planar graphs.

Input: A planar graph $G = (V, E)$ and a nonnegative integer k.
Question: Is there a choice of at most k vertices $V' \subseteq V$ such that for every vertex $v \in V$ there is either at least one vertex in V' that is a neighbor of v or $v \in V'$ (or both)?

For general graphs, in a parameterized sense, DOMINATING SET is even "harder" than INDEPENDENT SET, that is, it is $W[2]$-complete [88]. Can we use a similar argument as for INDEPENDENT SET in order to show fixed-parameter tractability of DOMINATING SET on planar graphs? Unfortunately, it is much harder to prove the existence of a bounded search tree here. Namely, the problem is the following. Assume that we want to argue as we did for INDEPENDENT SET, that is, choosing a vertex of minimum degree, then branching on it by recursively solving the problem in each branch. Clearly, a minimum degree vertex v or one of its neighbors can be chosen to be in an optimal dominating set. By way of contrast to INDEPENDENT SET, the point now is that we cannot say any longer that exactly one vertex of the closed neighborhood of v has to be in an optimal dominating set. It could be more than

one. For instance, removing v from G, we can only delete its adjacent edges, but not its adjacent vertices, because, although all its neighbors then being already dominated, they still are suitable candidates for an optimal dominating set. To circumvent this problem, in Section 3.6 we will formulate a more general, "annotated" version of DOMINATING SET, where there are two kinds of vertices in our graph. The technical effort for solving this problem increases significantly (see Section 3.6). Notably, the difficulty mainly comes from the mathematical analysis of the correctness, the presented algorithm still is relatively easy and also easy to implement [176].

3.2 Analyzing Search Tree Sizes

To analyze the sizes of the presented search trees for VERTEX COVER, 3-HITTING SET, and INDEPENDENT SET on planar graphs was straightforward. There, we branched into either two, three, or six cases, in each of which the parameter value could be decreased by one. This is due to the fact that in each case we selected exactly one element for inclusion into the set to be constructed. Thus, we had very regular branchings and since the depth of the search tree can be bounded by k then, we end up with search tree sizes 2^k, 3^k, and 6^k in the respective problems. The improvements on 2^k and 3^k known for VERTEX COVER and 3-HITTING SET, however, heavily rely on more complicated branchings with numerous case distinctions. And, even more importantly, often more than one element is selected for inclusion into the desired set. In addition, the numbers of selected elements may differ in different branches. This leads to more complicated recursions and to estimate the worst-case sizes of the corresponding search trees requires some mathematical tools. Fortunately, these tools are available and they are easy to use.

Search tree algorithms work in a recursive manner. The number of recursion calls is the number of nodes in the according tree. This number is governed by homogeneous, linear recurrences with constant coefficients. It is well-known how to solve them and the asymptotic solution is determined by the roots of the characteristic polynomial. We use the notation of Kullmann and Luckhardt [169, 170]. If the algorithm solves a problem of size n and calls itself recursively for problems of sizes $n - d_1, \ldots, n - d_i$ then $(d_1, \ldots, d_i)$ is called the *branching vector* of this recursion. It corresponds to the recurrence

$$T_n = T_{n-d_1} + \cdots + T_{n-d_i}. \tag{3.1}$$

Observe that in recurrence (3.1) we actually give a recurrence to estimate the number of leaves of a search tree. This is sufficient because for non-degenerate trees (i.e., all inner nodes have at least two children), as we always consider here, the leaves make at least half of the total number of tree nodes. In addition, also note that we assume that $T_0 = T_1 = \ldots = T_{d_i-1} = 1$ for the

termination of the recursion. The characteristic polynomial of this recurrence is

$$z^d = z^{d-d_1} + \cdots + z^{d-d_i}, \tag{3.2}$$

where $d = \max\{d_1, \ldots, d_i\}$. If α is a root of (3.2) with maximum absolute value then T_n is α^n up to a polynomial factor. We call $|\alpha|$ the *branching number* that corresponds to the branching vector $(d_1, \ldots, d_i)$. (In our case α is always real, since $1/\alpha$ is the dominant singularity of the series $T_n z^n$. The dominant singularity of a series with non-negative coefficients is always a positive, real number.) Moreover, if α is a single root, then even $T_n = O(\alpha^n)$ and all branching numbers that will occur in this work are single roots.

In the examples to follow, the size of the search tree is therefore $O(\alpha^k)$, where k is the parameter and α is the biggest branching number that will occur. For instance, in an intricate search tree algorithm for Vertex Cover [203, 204], among others, the branching vectors $(3, 5, 7)$, $(4, 5, 8, 9, 9)$, and $(3, 5, 8, 8)$ occur. Here, it is no longer obvious which the branching with the largest branching number is. Solving the corresponding recurrences (which can be done using any standard computer algebra system), we obtain the respective (approximate) branching numbers 1.273739, 1.290649, and 1.291743. The last number gives the worst case and the corresponding search tree algorithm (which has several more branchings) indeed has the worst-case bound 1.291743^k on its search tree size.

One further thing to learn from the above is that all the bounds for search tree sizes we give are *worst-case* estimates. They are based on the determination of worst-case branching vectors and branching numbers. It is well conceivable that the average-case sizes of the constructed search trees are significantly smaller in many cases. Since average-case complexity analysis is a very elusive matter, the normal way to find out the average-case and, thus, practical behavior of a search tree algorithm is through implementation and experiments. Experience tells us that, as a rule, additional heuristic improvements (affecting search tree size and running time, but not the quality of the solution) can be incorporated and the resulting algorithm may perform much better than to be expected from the sole consideration of the proven worst-case bounds. Finally, Chen *et al.* [60] proposed the iterative branching method as a means of a further improved complexity analysis and branching strategy. Roughly speaking, the idea is to iteratively pick "special" branches in a skillful way in order to keep some kind of invariant concerning "nice" branching situations which lead to good branching vectors. Refer to [60] for any details with respect to an application to Vertex Cover.

3.3 Closest String

The first example for a "state of the art" search tree is for a problem with applications in computational molecular biology. We follow parts of [137] in deriving a bounded search tree for Closest String (also cf. Subsection 1.5.3).

> **Input:** k strings $s_1, s_2, \ldots, s_k$ over alphabet Σ of length L each, and a nonnegative integer d.
> **Question:** Is there a string s such that $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$?

Here, $d_H(s, s_i)$ denotes the Hamming distance between strings s and s_i. We present a fixed-parameter algorithm with respect to parameter d. To start with, we need to introduce some notation and some easy observations. Given a set of k strings of length L, we can think of these strings as a $k \times L$ character matrix. The *columns* of a Closest String instance are the columns of this matrix. With the following observation by Evans and Wareham [100], we find that it is sufficient to solve instances containing less than kd columns. We call a column *dirty* iff it contains at least two different symbols from alphabet Σ. Clearly, "all the work" in solving Closest String concentrates on the dirty columns of the input instance.

Lemma 3.3.1. *Given a* Closest String *instance with k strings of length L and integer d. If the corresponding $k \times L$ matrix has more than kd dirty columns, then there is no solution to this instance.* □

Lemma 3.3.1 gives a simple reduction to problem kernel. Notably, it is not with respect to the parameter d alone, but needs to incorporate parameters k *and* d. A useful kernelization only with respect to d currently is not known.

In Fig. 3.1, we outline a recursive algorithm solving Closest String. For the correctness of the algorithm, we need the following simple observation.

Lemma 3.3.2. *Given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$ and a nonnegative integer d. If there are $i, j \in \{1, \ldots, k\}$ with $d_H(s_i, s_j) > 2d$ then there is no string s with $\max_{i=1,\ldots,k} d_H(s, s_i) \leq d$.*

Proof. The Hamming distance satisfies the triangle inequality. If $d_H(s_i, s_j) > 2d$ and we are given an arbitrary string s, we, therefore, know that $d_H(s, s_i) + d_H(s, s_j) > 2d$. It follows that $d_H(s, s_i) > d$ or $d_H(s, s_j) > d$ (or both). □

Theorem 3.3.1. Closest String *can be solved in time $O(kL + kd \cdot d^d)$.*

Proof. Fig. 3.1 contains the recursive procedure *CSd* which, after a "successfull" reduction to problem kernel (i.e., according to Lemma 3.3.2 a solution is possible) is invoked by the call $CSd(s_1, d)$. Referring to this by "Algorithm CS-D," we subsequently analyze its running time and prove that it correctly solves Closest String.

Algorithm CS-D, recursive procedure $CSd(s, \Delta d)$**:**
Global variables: Set of strings $S = \{s_1, s_2, \ldots, s_k\}$, nonnegative integer d.
Input: Candidate string s and integer Δd.
Output: A string $\hat{s}$ with $\max_{i=1,\ldots,k} d_H(\hat{s}, s_i) \leq d$ and $d_H(\hat{s}, s) \leq \Delta d$, if it exists, and "not found," otherwise.
Method:
(D0) **if** $(\Delta d < 0)$ **then return** "not found";
(D1) **if** $(d_H(s, s_i) > d + \Delta d)$ for some $i \in \{1, \ldots, k\}$ **then return** "not found";
(D2) **if** $(d_H(s, s_i) \leq d)$ for all $i = 1, \ldots, k$ **then return** s;
(D3) choose $i \in \{1, \ldots, k\}$ such that $d_H(s, s_i) > d$:
    $P := \{\, p \mid s[p] \neq s_i[p] \,\}$;
    choose any $P' \subseteq P$ with $|P'| = d + 1$;
    for all $p \in P'$ **do**
        $s' := s$;
        $s'[p] := s_i[p]$;
        $s_{ret} := CSd(s', \Delta d - 1)$;
        if $s_{ret} \neq$"not found" **then return** s_{ret};
(D4) **return** "not found"

Fig. 3.1. Algorithm CS-D. Inputs are a Closest String instance consisting of a set of strings $S = \{s_1, s_2, \ldots, s_k\}$ of length L, and an integer d. First, we perform a preprocessing performing the reduction to problem kernel as shown in Lemma 3.3.1: We select the dirty columns. If there are more than kd many then we reject the instance. If there are at most kd many then we invoke the recursion with $CSd(s_1, d)$.

1. *Running time.*
Prior to the recursion, we perform the reduction to problem kernel as described in Lemma 3.3.1. This preprocessing, reducing the size of the input instance to kd, can be done in time $O(kL)$. Now, we consider the recursive part of the algorithm. Parameter Δd is initialized to d. Every recursive call decreases Δd by one. The algorithm stops when $\Delta d < 0$. Therefore, the algorithm builds a search tree of height at most d. In one step of the recursion, the algorithm chooses, given the current candidate string s, a string s_i such that $d_H(s, s_i) > d$. It creates a subcase for $d + 1$ of the positions in which s and s_i disagree (there are more than d but at most $2d$ such positions). This yields an upper bound of $(d + 1)^d$ on the search tree size. Every step of the recursion only needs linear time $O(kd)$. Before starting the recursion, we build a table containing the distances of the candidate s to all other given strings in time $O(kd)$. Using this table, instructions (D1) and (D2) can be done in time $O(k)$. In instruction (D3), we need time $O(k)$ to select the s_i for branching and time $O(kd)$ to find the positions in which s and s_i differ. For $d+1$ of the differing positions we modify the candidate, update the table of distances, and call the procedure recursively. Since we changed only one position, we can update the table of distances in time $O(k)$.

2. *Correctness.*
We show that Algorithm CS-D finds a string s with $\max_{i=1,\ldots,k} d_H(s, s_i) \leq d$, if one exists. Here, we explicitly show only the correctness of the first recursive step; the correctness of the algorithm then follows with an inductive application of the argument.

In the situation that s_1 satisfies $\max_{i=1,\ldots,k} d_H(s_1, s_i) \leq d$, we immediately find a solution, namely s_1. If s_1 is not a solution but there exists a closest string s for this instance with distance value d, then there is a string s_i, $i = 2, \ldots, k$, such that $d_H(s_1, s_i) > d$. For branching, we consider the positions where s_1 and s_i differ, i.e., $P := \{\, p \mid s_1[p] \neq s_i[p] \,\}$. Algorithm CS-D successively creates subcases for $d + 1$ positions p from P in order to create a new candidate by altering the respective position p from $s_1[p]$ to $s_i[p]$. Such a "move" is correct if we choose a position p from $P_1 := \{\, p \mid s_1[p] \neq s[p] = s_i[p] \,\}$. Now, we show that (at least) one of our $d + 1$ moves is a correct one. We observe that $P = P_1 \cup P_2$ for $P_2 := \{\, p \mid s[p] \neq s_i[p] \,\}$. Since $d_H(s, s_i) \leq d$ we know that $|P_2| \leq d$. Therefore, at least one of our $d + 1$ subcases will try a position from P_1. An inductive application of this argument shows that Algorithm CS-D finds a closest string for this instance, if one exists. Note that we allow to alter the candidate s in only those positions p in which $s[p] = s_1[p]$. Having started with s_1, every positions p with $s[p] \neq s_1[p]$ has been previously altered. It does not make sense, however, to alter the candidate twice in one position. Regarding instruction (D1), we can analogously to Lemma 3.3.2 observe that it is correct to omit those branches where the candidate string s satisfies $d_H(s, s_i) > d + \Delta d$ for some string s_i of the given strings $s_1, \ldots, s_k$.[1]: Assume that there is a solution s'. Solution s' can differ from s_i in at most d positions. Due to the triangle inequality, s' would differ from s in more than Δd positions, contradicting the assumption that s' is a solution. □

With Algorithm CS-D, we can find a solution if one exists. We find *all* solutions if the given distance parameter d is optimal. We do not necessarily find all solutions to a given instance when d is not optimal. Using binary search, however, we can find the optimal distance value at most d at the cost of a constant time factor.

It is open to give a good bounded search tree yielding fixed-parameter tractability with respect to parameter k. In Section 4.1, however, we will see that a deep result from *integer linear programming* theory implies that Closest String is fixed-parameter tractable with respect to parameter k.

[1] If there are two strings s_i, s_j with $d_H(s_i, s_j) = 2d$ then we can use a special strategy: We know that a solution has to differ from both s_i and s_j in d positions. We can search a solution by trying all ways to partition the set of positions p with $s_i[p] \neq s_j[p]$ into two sets of size d. In the candidate, we give to one set of positions the characters of s_i, to the second set the characters of s_j. This strategy has running time $O(kL + kd \cdot 2^{2d})$.

3.4 3-Hitting Set

The next example is a bounded search tree for 3-Hitting Set (3HS). A special reason for choosing this example was that it offers a complicated, but still easier to overview search tree machinery than the best known Vertex Cover algorithms do. In addition, the determination of the size of the search tree is a little more involved than usual. We follow [208]. Recall from Section 2.3 that we already know that 3HS has a problem kernel of size $O(k^3)$.

> **Input:** A collection C of subsets of size three of a finite set S and a nonnegative integer k.
> **Question:** Is there a subset $S' \subseteq S$ with $|S'| \leq k$ which allows S' contain at least one element from each subset in C?

We assume that no three element subset occurs more than once within the collection. By n we denote the length of the encoding of the input. Equally, 3HS can be seen as a vertex cover problem for *hypergraphs:* Interpret the elements of S as vertices and interpret the size three subsets as hyperedges. Thus, a hyperedge now joins three vertices instead of two. As a result, 3HS requires the *covering* of all these three element sets (hyperedges) by elements (vertices) completely analogously to Vertex Cover. From this point of view it is natural to speak of the *degree* of elements in S. It simply means the number of subsets in which it occurs. Moreover, we call a given collection of subsets *d-regular* if each element x has exactly degree d. Finally, we call an element $y \in S$ *dominated* by an element $x \in S$ if each subset containing y also contains x.

A special property of the subsequent mathematical analysis of the search tree size is that in contrast to previous estimations of search tree sizes in parameterized complexity, we use a *system* of recurrences. For example, we may have recurrences as

$$T_k = 1 + T_{k-1} + T_{k-2} + B_{k-1}$$

and

$$B_k = 1 + B_{k-1} + T_{k-1}$$

with $T_1 = B_1 = 1$ and $T_2 = 2$, where we start with T_k. Since for nondegenerate trees (inner nodes have at least two children) the number of leaves is at least half of all tree nodes, to get an asymptotic solution for the recurrences we may drop the additive term "1+." Solving these simplified recurrences, we obtain the branching number α which tells us that $T_k = O(\alpha^k)$, thereby giving an upper bound for the search tree size. Note that we will have to study several cases for our algorithm, each yielding some recurrence(s).

3.4.1 The Algorithm

In what follows, the fundamental aim is to undertake a case distinction concerning the degree of elements x in the base set S. However, there are also

some special cases, which are always considered first. For example, if there is a singleton $\{x\}$ in our collection, we clearly have to take x in our hitting set. A simple but important concept is that of *domination*. An element x is dominated by an element y if whenever x occurs in a set of the collection, then y will occur in this set as well. In this case, we can delete x from the sets without repercussion. Lastly, before coming to degree considerations, a case of special importance is when we have subsets with two elements in our collection. In this case, which can easily be dealt with, the size of the search tree is at most B_k, whereas when there is no such subset (i.e., all subsets have size three), its size is at most T_k.

Summarizing, the algorithmic structure of the search tree is as follows. Observe that the subsequent order of the steps is important. In each step, the algorithm always executes the applicable step with the lowest possible number:

1. Deal with simple cases, that is, one element subsets, elements occurring in only one set, or dominated elements.
2. Deal with subsets of size two.
3. Deal with elements of degree three.
4. Deal with elements of degree at least four.
5. Deal with the case that the collection of subsets is two-regular.

It is easily verified that the above case distinction covers all cases that may occur. As a rule, each case leads to some recursive calls. The worst case leads to the bound $O(2.27^k)$ on the search tree size.

Simple Cases

There exist some simple cases that we always consider first. Firstly, assume that there is a singleton, say $\{x\}$. Then we clearly have to take x into the hitting set without any branching of the recursion.

Secondly, assume that there is an element $x \in S$ that occurs in only one set of size three, e.g., $\{x, a, b\}$. Then it suffices to consider the covering of $\{a, b\}$, leading to the recursive call B_k. Thus, we find the recurrences as shown for subsets of size two.

Thirdly, if an element y is dominated by an element x it never occurs in a set without x occurring in the same set. This implies, however, that it would not make sense to take y and not to take x into the hitting set. As a consequence, we can simply throw away all occurrences of y, thus obtaining two-element-sets instead of three element ones.

In the cases handled in the following, it will be of key importance to rely on the absence of dominated elements, degree 1 elements, and sets of size one in the given instance.

Subsets of Size Two

We distinguish whether the size of all sets is at least three and whether there is at least one set with size two. In this subsection we assume the latter. An upper bound on the number of leaves in a branching tree whose root has this property will be called B_k. First of all, note that we can discard from further consideration the case when all sets have size two, because we can simply apply the much better results for 2HS, i.e., Vertex Cover. Hence, in the following subcases, at least one subset of size three occurs.

Let us first handle the special case where there are two sets

$$\{x, y\} \text{ and } \{x, a, b\},$$

where x only occurs in these two sets and b may be missing from the second set. We can branch according to y: If y is in the hitting set then x occurs only in $\{x, a, b\}$ and x can be deleted from this subset. This must happen because $y \notin \{a, b\}$ (otherwise x would be dominated by y). The corresponding subtree has at most B_{k-1} leaves. If y is not in the hitting set then x is. Since there are no elements occurring in only one subset y occurs in some other set from which it is deleted, leaving a set of size at most two. Hence, this subtree has at most B_{k-1} leaves, as well. Altogether, the corresponding upper bound for this case is $2B_{k-1}$.

We continue with the case that x occurs in at least two sets of size two and one set of size three, that is

$$\{x, y_1\},\ \{x, y_2\},\ \{x, a, b\}.$$

If x is in the hitting set then we get a T_{k-1} branch. If x is not in the hitting set, y_1 and y_2 have to be in the hitting set and (since, without loss of generality, $y_1 \neq y_2$) we trivially get a T_{k-2} branch. The corresponding upper bound reads $T_{k-1} + T_{k-2}$.

Next, we consider the case of three sets

$$\{x, y\},\ \{x, a, b\},\ \{x, a, c\},$$

where x may occur in other sets as well. If x is in the hitting set, we get a T_{k-1} branch. Otherwise, y must be in the hitting set. Among others, the sets $\{a, b\}$ and $\{a, c\}$ remain. Now we branch according to a. If a is in the hitting set then we obtain a T_{k-2} branch and, otherwise, b and c must be in the hitting set, yielding a T_{k-3} branch. (Note that $b \neq c$.) The corresponding upper bound is $T_{k-1} + T_{k-2} + T_{k-3}$.

Finally, the remaining case is three sets

$$\{x, y\},\ \{x, a, b\},\ \{x, c, d\},$$

where x may again occur elsewhere but $\{a, b\} \cap \{c, d\} = \emptyset$. If x is in the hitting, set we get a T_{k-1} branch. Otherwise, we branch according to a. Note

that now, y must be in the hitting set. If a is in the hitting set, we still have the set $\{c, d\}$ and, hence, a B_{k-2} branch. If a is not, then b is in the hitting set and we are also left with $\{c, d\}$: Another B_{k-2} branch. The corresponding upper bound reads $T_{k-1} + 2B_{k-2}$.

Degree Three

Here, we assume that there is an element x that occurs in exactly three sets

$$\{x, a_1, a_2\}, \{x, b_1, b_2\}, \{x, c_1, c_2\}.$$

We can assume that all sets have size three because if they did not the previous considerations would apply.

We consider two subcases: Firstly, that there is another element besides x that occurs in at least two of the three sets (A.) and, secondly, that there is no such element (B.).

A. Let us assume $a_1 = b_1$. Then $a_1 \neq c_1$ and $a_1 \neq c_2$ because, otherwise, x would be dominated by $a_1 = b_1$. We make three branches: Either c_1 and a_1 are in the hitting set or c_1 is and a_1 is not or, finally, c_1 is not in the hitting set.

If c_1 and a_1 are in the hitting set then we get a T_{k-2} branch because the size of the hitting set grows by two.

If only c_1 is in the hitting set but a_1 is not then the elimination of $\{x, c_1, c_2\}$ will leave $\{x, a_1, a_2\}, \{x, a_1, b_2\}$ as the only sets that contain x. Hence, x is dominated by a_1 and, since we assume that a_1 is not in the hitting set, x won't be either. This leaves the two singletons $\{a_2\}$ and $\{b_2\}$. They are, in fact, *two* sets, since $a_2 \neq b_2$ (otherwise x would not have occurred in three sets, but only in two). We can include a_2 and b_2 together with c_1 in the hitting set and, consequently, obtain a T_{k-3} branch.

Finally, if c_1 is *not* in the hitting set then $\{x, c_2\}$ remains after c_1 has been eliminated, yielding at least a B_k branch.

Summarizing, the upper bound reads $T_k \leq T_{k-2} + T_{k-3} + B_k$.

B. Now we may assume that $a_1, a_2, b_1, b_2, c_1, c_2$ are pairwisely different. We branch on x. By a T_{k-1} branch, we deal with the case where x is in the hitting set. Otherwise we can eliminate x, leaving $\{a_1, a_2\}, \{b_1, b_2\}, \{c_1, c_2\}$. Now we branch according to whether a_1 or a_2 is in the hitting set and according to whether b_1 or b_2 is in the hitting set. Hence, we get four branches, each putting two elements in the hitting set and leaving the two element set $\{c_1, c_2\}$. Therefore, the corresponding recurrence reads $T_k \leq T_{k-1} + 4B_{k-2}$.

Degree at Least Four

Here, we assume that there is an element x that occurs in at least fours sets

$$\{x, a_1, a_2\}, \{x, b_1, b_2\}, \{x, c_1, c_2\}, \{x, d_1, d_2\}.$$

Subsequently, we distinguish between two cases.

Firstly, assume that a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2 are not all pairwise different. Without loss of generality, assume that $a_1 = b_1$. (Of course, $a_1 \neq a_2$, $b_1 \neq b_2$, $c_1 \neq c_2$, and $d_1 \neq d_2$.) Clearly, $a_1 = b_1$ implies that $a_2 \neq b_2$ because, otherwise, we would have two times the same size three subset in our given collection. We claim the upper bound $T_{k-1} + B_{k-1} + T_{k-2}$ for this case. We branch on x as follows. If x is part of the hitting set, all four sets above are covered. We obtain a T_{k-1} branch. If we do not include x in the hitting set, then we will, in particular, obtain the sets $\{a_1, a_2\}$ and $\{a_1, b_2\}$ which are to be covered. Upon branching on a_1, we end up with the following situation. If a_1 is in the hitting set, both of these sets are covered and there must be an additional two-element set, without loss of generality, $\{c_1, c_2\}$, remaining. This is due to the following fact: Since due to the preceding considerations we may assume that a_1 is not dominated by x and vice versa (see "Simple Cases") there must be a set containing x and not containing a_1. Without loss of generality, let this set be $\{x, c_1, c_2\}$. Hence, since we decided not to take x in the hitting set, $\{c_1, c_2\}$ remains to be covered. Thus, we can continue with a B_{k-1} branch here. Eventually, if a_1 is not in the hitting set then only $\{a_2\}$ and $\{b_2\}$ remain to be covered and, clearly, we have to take both, leading to a T_{k-2} branch.

Clearly, here and in the following case, the situation (and, consequently, the branching number) improves if we have degree greater than four.

Now let us turn to the second case, that is, assuming that a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2 are pairwise distinct. In this case, we claim the upper bound $T_{k-1} + 8B_{k-3}$. Again, we branch on x. Bringing x into the hitting set leads to a T_{k-1} branch. If x is not in the hitting set, we have to cover the four sets

$$\{a_1, a_2\}, \{b_1, b_2\}, \{c_1, c_2\}, \{d_1, d_2\}.$$

Since these contain eight distinct elements we can branch in the manner of a binary tree of height 3, yielding eight possibilities, each putting three elements in the hitting set and, without loss of generality, leaving in each branch the two element set $\{d_1, d_2\}$. Hence, we get eight B_{k-3} branches.

The Collection Is Two-Regular

Finally, we end up with two-regular collections, that is, each element $x \in S$ occurs in exactly two subsets of the given collection. Hence, we have

$$\{x, a, b\}, \{x, c, d\}.$$

We may assume that a, b, c, d all are pairwise distinct, because if, e.g., $a = c$, then x would be dominated by a. Therefore, we branch according to a. If a is in the hitting set, then $\{x, a, b\}$ is covered and $\{x, c, d\}$ will be replaced by $\{c, d\}$, because x is dominated after $\{x, a, b\}$ has been removed. This leads to

a B_{k-1} branch. If a is not in the hitting set then we will branch additionally according to x. If x is in the hitting set then both of the above sets are covered. However, since we assume two-regularity we also know that a has to appear in some other set $\{a, e, f\}$. Since this now is the only remaining occurrence of a, this set is replaced by $\{e, f\}$. Consequently, we find the recursive call B_{k-1}. Finally, suppose that neither a nor x are in the hitting set. Then b has to be in the hitting set. Furthermore, $\{x, c, d\}$ is replaced by $\{c, d\}$, yielding a B_{k-1} branch. Hence, the size of this subtree is at most $3B_{k-1}$ in the case of two-regular collections.

Summarizing the Various Parts

Altogether, we have the following result.

Theorem 3.4.1. 3-Hitting Set *has a search tree of size* $O(2.27^k)$ *and each search tree node can be processed in time linear in the input size.*

Proof. We described a search tree whose size is bounded from above by the given recurrences, which yield search tree size $O(2.27^k)$. Since for each node of the search tree we have to process the collection of subsets (i.e., throwing out subsets or deleting elements from them) in time linear in the input size, we obtain linear time complexity for the processing of one search tree node. □

3.4.2 *d*-Hitting Set for General *d*

Finally, we present a more general algorithm that works for the generalization of 3-Hitting Set to subsets of sizes larger than three. It is quite efficient, but, of course, it is outperformed by the above algorithm for the most important case $d = 3$.

The trivial algorithm that tries all d possibilities for a set of d elements has running time $O(d^k + n)$. Our algorithm is better, having running time $O(\alpha^k + n)$, where

$$\alpha = \frac{d-1}{2} + \frac{d-1}{2}\sqrt{1 + \frac{4}{(d-1)^2}} = d - 1 + \frac{1}{d-1} + O(d^{-3}) = d - 1 + O(d^{-1}).$$

The algorithm proceeds as follows.

1. Eliminate all dominating elements.
2. Choose some set $s = \{x_1, x_2, \ldots, x_d\}$.
3. Branch according to the following possibilities:
   a) Choose x_1 for the hitting set, or
   b) choose that x_1 is not in the hitting set but x_i is for $i = 2, \ldots, d$.

d	3	4	5	6	7	8	9	10	20	50	100
T_k	2.41^k	3.30^k	4.23^k	5.19^k	6.16^k	7.14^k	8.12^k	9.11^k	19.05^k	49.02^k	99.01^k

Table 3.1. Search tree sizes for d-Hitting Set.

That makes d branches in total. If T_k is the number of leaves in a branching tree, then the first branch has at most T_{k-1} leaves. Let B_k be the number of leaves in a branching tree where there is at least one set of size $d-1$ or smaller. For each $i = 2, \ldots, d$, there is some set s' in the given collection such that $x_1 \in s'$, but $x_i \notin s'$. Therefore, the size of s' is at most $d-1$ after excluding x_1 from and including x_i in the hitting set. Altogether we get $T_k \leq T_{k-1} + (d-1)B_{k-1}$.

If there is already a set with at most $d-1$ elements, we can play the same game and get $B_k \leq T_{k-1} + (d-2)B_{k-1}$. The branching number of this recursion is α from above.

Table 3.1 shows the resulting running times for several values of d. Note that even for $d = 3$ the result is better than the previously best special algorithm for 3-Hitting Set (cf. [92]) and that α is always *smaller* than $d - 1 + (d-1)^{-1}$.

Observe, however, that the general Hitting Set problem (unbounded subset size) is $W[2]$-complete [88], so there is little chance of showing fixed-parameter tractability for the general problem with unbounded value of d.

3.5 Maximum Satisfiability

Many bounded search tree algorithms (including those for Vertex Cover and 3-Hitting Set) achieve small search tree size by distinguishing between quite a number of "branching cases." These algorithms are solely governed by branching (or "splitting") rules. Now, by investigating the Maximum Satisfiability (MaxSat) problem we will give a example where besides branching also so-called *transformation* rules play a major role. Moreover, MaxSat is a good example for the limitations and pitfalls of the fixed-parameter approach, as we will discuss in the end of this relatively lengthy section. We follow parts of [206].

MaxSat is the following problem:

> **Input:** A boolean formula in conjunctive normal form consisting of K clauses and a nonnegative integer k.
> **Question:** Is there a truth assignment satisfying at least k clauses.

Like the satisfiability problem itself, MaxSat plays an important role in computer science since it is the basis for solutions of major problems in AI and combinatorial optimization [27, 147]. It has also been a subject of the second DIMACS challenge [159]. It has been termed "a paradigmatic problem for the

"algorithmic engineering" and scientific testing and tuning effort" [26]. According to Crescenzi and Kann [73], MaxSat is among the 15 most popular problems in combinatorial optimization. MaxSat cannot be solved in polynomial time unless $P = NP$ since it generalizes the Satisfiability problem.

3.5.1 Basic Definitions

We also refer to Section 2.2 for some very basic definitions. A subformula, i.e., a subset of clauses, is called *closed* if it is a minimal subset of clauses such that no variable in this subset also occurs outside this subset in the rest of the formula. A clause that contains the same variable positively and negatively, e.g., $\{x, \bar{x}, y, \bar{z}\}$, is satisfied by every assignment. We will not allow for such clauses but we assume that such clauses are always replaced by a special clause $\top$ which denotes a clause that is always satisfied. We call a clause containing r literals simply an *r-clause.* A formula in *2CNF* is one consisting of 1- and 2-clauses. We assume that 0-clauses do not appear in our formula since they clearly are not satisfiable. Let l be a literal occurring in a formula F. We call it an (i, j)-*literal* if the variable corresponding to l occurs exactly i times positively and exactly j times negatively, respectively. In analogy, we get (i^+, j)-, (i, j^+)-, and (i^+, j^+)-*literals* by replacing "exactly" with "at least" at the appropriate positions and get (i^-, j)-, (i, j^-)- and (i^-, j^-)-*literals* by replacing "exactly" with "at most." We denote the number of occurrences of a literal l in a formula F by $\#_l(F)$.

For a literal l and a formula F, let $F[l]$ be the formula originating from F by replacing all clauses containing l by $\top$ and removing $\bar{l}$ from all clauses where it occurs. To estimate the time complexity of our algorithms, the following notions are useful: $S(F)$ denotes the number of $\top$-clauses in F, and $maxsat(F)$ denotes the maximum number of simultaneously satisfiable clauses in F. We say two formulas F and G are *equisatisfiable* if $maxsat(F) = maxsat(G)$. A formula that contains only $\top$ as its clauses is called *final*. Obviously, there is exactly one final formula in the equivalence class of equisatisfiable formulas, assuming that 0-clauses are deleted from our formula as soon as they exist.

Definition 3.5.1. *A formula is called* nearly monotone *if negative literals occur only in 1-clauses. It is called a* simple formula *if it is nearly monotone and each pair of variables occurs together in one clause at most.*

Definition 3.5.2. *For a variable x, we say $\tilde{x}$ occurs in a clause C if $x \in C$ or $\bar{x} \in C$.*

For example, $\tilde{x}$ occurs in $\{\bar{x}, y, z\}$ and in $\{x, y, z\}$ but x occurs only in $\{x, y, z\}$ and $\bar{x}$ only in $\{\bar{x}, y, z\}$. As a rule, we will use x, y, z to denote variables and l to denote a literal.

3.5.2 Transformation and Branching Rules

In the following, we present algorithms that solve MaxSat by mapping a formula to the unique, equisatisfiable, final formula. We distinguish two possibilities: If a formula is replaced by another formula then we speak of a *transformation rule*; if one formula is replaced by several other formulas then we speak of a *branching rule*. The resulting formula or formulas are then solved recursively, a technique that goes back to the Davis–Putnam procedure [78]. It must be emphasized here that the purpose of the transformation rules as presented here is not to obtain a reduction to problem kernel (cf. Section 2.2) but they serve to simplify the subsequently described branching strategy and its mathematical analysis.

Transformation Rules

A transformation rule $\frac{F}{F'}$ replaces F by F', where F' and F are equisatisfiable but F' is simpler. We will use the following transformation rules, whose correctness is easily confirmed. We mention in passing that many rules that apply for the conceptually easier Satisfiability problem do not (directly) apply for MaxSat, thus requiring new techniques for MaxSat.

Pure Literal Rule.

$$\frac{F}{F[x]} \text{ if } x \text{ is a } (1^+, 0)\text{-literal.}$$

The correctness of the pure literal rule is easy to prove. Obviously, there is an optimal assignment I that fulfills $I(x) = 1$.

Complementary Unit-Clause Rule.

$$\frac{F}{\{\top\} \cup G} \text{ if } F = \{\{\bar{x}\}, \{x\}\} \cup G.$$

For *every* assignment $maxsat(F) = maxsat(G) + 1$.

Dominating Unit-Clause Rule.

$$\frac{F}{F[l]} \text{ if } \bar{l} \text{ occurs in } i \text{ clauses, and } l \text{ occurs at least } i \text{ times in 1-clauses.}$$

Resolution Rule.

$$\frac{\{\{\bar{x}\} \cup K_1, \{x\} \cup K_2\} \cup G}{\{\top, K_1 \cup K_2\} \cup G} \text{ if } G \text{ does not contain } \tilde{x}.$$

Small Subformula Rule. Let $F = \{\{x', y', \ldots\}, \{x'', y'', \ldots\}, \{x''', y''', \ldots\}\} \cup G$, where G contains neither $\tilde{x}$ nor $\tilde{y}$ and $x', x'', x''' \in \{x, \bar{x}\}$ and $y', y'', y''' \in \{y, \bar{y}\}$. Then

$$\frac{F}{\{\top, \top, \top\} \cup G},$$

since there is always an assignment to x and y only that already satisfies $\{\{x', y', \ldots\}, \{x'', y'', \ldots\}, \{x''', y''', \ldots\}\}$.

Star Rule. A formula $\{\{\bar{x}_1\}, \{\bar{x}_2\}, \ldots, \{\bar{x}_r\}, \{x_1, x_2, \ldots, x_r\}, \{x_1, x_2, \ldots, x_r\}\}$ is called an *r-star.* Let F be an r-star. Then

$$\frac{F}{\{\top, \ldots, \top\}},$$

where the "$\top$-multi-set" contains $r + 1$ many $\top$'s.

Definition 3.5.3. *A formula is* reduced *if no transformation rule is applicable, each variable occurs at least as often positively as negatively, and it contains no empty clauses. Using the above transformation rules, Reduce(F) denotes the corresponding reduced, equisatisfiable formula.*

When reducing a formula, it can be necessary to rename literals. Observe that in the rest of the paper many arguments will rely on the fact that we are dealing with a reduced formula. Particularly note that a variable in a reduced formula occurs at least three times.

Branching Rules

The branching rules are based on partitioning the search space, i.e., dividing the set of all possible assignments into several parts, finding an optimal assignment within each part, and then taking the best of them. Careful splits enable us to simplify the formula in some of the branches. Take, for example, the formula

$$\{\{x, y\}, \{\bar{x}, y\}, \{x, \bar{y}\}, \{\bar{x}, \bar{y}\}\}$$

and split the set of all assignments into those with $x = 0$ and those with $x = 1$. If $x = 0$, the formula becomes

$$\{\{0, y\}, \{1, y\}, \{0, \bar{y}\}, \{1, \bar{y}\}\}$$

which simplifies to

$$\{\{y\}, \top, \{\bar{y}\}, \top\}.$$

We assume in the following that the elimination of 0 or 1 in clauses is done automatically whenever it occurs; a 0 is removed from its clause and a clause that contains 1 is replaced by $\top$. Finally, we can simplify $\{\{y\}, \{\bar{y}\}, \top, \top\}$ with the complementary unit-clause rule to get $\{\top, \top, \top\}$. The result is $|\{\top, \top, \top\}| = 3$ for assignments with $x = 1$. Similarly, we get the result 3 for assignments with $x = 0$, so the result is "3 satisfiable clauses" which is obviously correct.

If we remove m clauses in which l occurs from F to get $F[l]$ then obviously $S(F[l]) = S(F) + m$ if we look *only* at assignments where $l = 1$. In general, however, we can at least say that

$$S(F[l]) \geq S(F) + m$$

Input: A formula F
Output: An equisatisfiable formula $A(F) = \{\top, \ldots, \top\}$
Method:
$F \leftarrow \mathit{Reduce}(F)$;
if F is final **then return** F
else
    let $\tilde{x}$ be a variable that occurs in F;
    return $\max\{A(F[x]), A(F[\bar{x}])\}$
fi

Fig. 3.2. Algorithm A to compute a final, equisatisfiable formula. Note that $\max\{A(F[x]), A(F[\bar{x}])\}$ is the multi-set with the maximum number of $\top$'s.

since an assignment where $l = 0$ could be better than all assignments where $l = 1$. Thus, a simple algorithm to compute $\mathit{maxsat}(F)$ is easily developed and can be found in Fig. 3.2.

In the rest of this subsection, we describe our set of branching rules. We distinguish between three basic cases, the first being easy: Either there is a variable occurring at least five times in the given formula or all variables occur three or four times in the formula and either there is one occurring exactly three times or *all* variables occur exactly four times. Clearly, this gives a complete case distinction.

There is a Variable that Occurs at Least Five Times. F1 Let F be reduced.

$$\frac{F}{F[x], F[\bar{x}]} \text{ if } \tilde{x} \text{ occurs at least five times in } F.$$

We get $S(F[x]) \geq S(F) + a$ and $S(F[\bar{x}]) \geq S(F) + b$ with $a, b \geq 1$ and $a + b = 5$.

Each Variable Occurs Three or Four Times and Some Variable Occurs Exactly Three Times. In the following we present seven branching rules **T1**–**T7** and an analysis with respect to $S(F)$. These rules are applicable if F is reduced and all literals in F are $(2,1)$, $(3,1)$, or $(2,2)$-literals. Moreover, there must be at least one $(2,1)$-literal x. In what follows, we firstly describe our set of rules and secondly show that it really handles all possible cases.

T1 $\dfrac{F}{F[l'], F[\bar{l'}]}$ if $F = \{\{\bar{x}, l', \ldots\}, \{x, \ldots\}, \{x, \ldots\}, \{l'', \ldots\}, \{l''', \ldots\}, \ldots\}$ and $l', l'', l''' \in \{y, \bar{y}\}$.

First, assume that $l' = y$. Then, in $F[l']$ we get the first clause satisfied and since y is a $(2,1)$-literal or better, at least one of the last two clauses is also satisfied. Additionally, x then becomes a pure literal and the second and third clauses can be satisfied setting $x = 1$ by the pure literal rule. In

$F[\bar{l}']$, trivially, at least one clause is directly satisfied. Altogether, we have $S(\mathit{Reduce}(F[l'])) \geq S(F) + 4$ and $S(F[\bar{l}']) \geq S(F) + 1$. Second, assume that $l' = \bar{y}$. Arguing in an analogous way as before, we obtain $S(\mathit{Reduce}(F[l'])) \geq S(F) + 3$ and $S(F[\bar{l}']) \geq S(F) + 2$.

T2 $\dfrac{F}{F[l], F[\bar{l}]}$ if $F = \{\{\bar{x}, \bar{l}, \ldots\}, \{x, l, \ldots\}, \{x, \ldots\}, \{l, \ldots\}, \ldots\}$.

Clearly, $S(\mathit{Reduce}(F[l])) \geq S(F) + 3$ since two clauses containing l are satisfied and then another clause is satisfied by the resolution rule. We also get $S(\mathit{Reduce}(F[\bar{l}])) \geq S(F)+3$: Since at least one clause containing $\bar{l}$ is satisfied, x becomes a pure literal, thus satisfying one or two more clauses because of the pure literal rule.

T3 $\dfrac{F}{F[x], F[\bar{x}]}$ if $F = \{\{\bar{x}, y, \ldots\}, \{x, y, \ldots\}, \{x, \ldots\}, \{\bar{y}, \ldots\}, \ldots\}$ and y is a $(2, 1)$-literal.

Now $S(\mathit{Reduce}(F[\bar{x}])) \geq S(F)+2$, because of one directly satisfied clause and the resolution rule on $\{x, y, \ldots\}$ and $\{\bar{y}, \ldots\}$; $S(\mathit{Reduce}(F[x])) \geq S(F) + 3$ because of two directly satisfied clauses and the resolution rule on $\{\bar{x}, y, \ldots\}$ and $\{\bar{y}, \ldots\}$.

T4 $\dfrac{F}{F[y], F[\bar{y}]}$ if $F = \{\{\bar{x}, y, \ldots\}, \{x, \bar{y}, \ldots\}, \{x, \ldots\}, \{y, \ldots\}, \ldots\}$ and y is a $(2, 1)$-literal.

Then $S(\mathit{Reduce}(F[\bar{y}])) \geq S(F) + 2$, since $\{x, \bar{y}, \ldots\}$ is directly satisfied and we get one more clause from the resolution rule on $\{\bar{x}, y, \ldots\}$ and $\{x, \ldots\}$. Also, $S(\mathit{Reduce}(F[y])) \geq S(F)+4$ since two clauses are directly satisfied and x becomes a pure literal in two clauses.

T5 $\dfrac{F}{F[x], F[\bar{x}]}$ if $F = \{\{\bar{x}\}, \{x, y, \ldots\}, \{x, z\}, \{y, \ldots\}, \{\bar{y}\}, \{\bar{z}\}, \ldots\}$ and y and z are $(2, 1)$-literals.

Then $S(\mathit{Reduce}(F[x])) \geq S(F) + 4$ because of two directly satisfied clauses and the resolution rule on y satisfying another clause. Then z becomes a $(1^-, 1)$-literal and resolution, pure literal rule, or complementary unit-clause rule are applicable. Clearly, $S(F[\bar{x}]) = S(F) + 1$. Observe that we did not fix the third occurrence of z—for example, $\tilde{z}$ might occur in the clause $\{x, y, \ldots\}$.

T6 $\dfrac{F}{F[l], F[\bar{l}]}$ if $F = \{\{\bar{x}, \ldots\}, \{x, l, \ldots\}, \{x, \ldots\}, \ldots\}$, l is a $(3, 1)$- or $(2, 2)$-literal, and l does not occur together with $\tilde{x}$ in three clauses.

We have $S(F[l]) \geq S(F)+a$, $S(F[\bar{l}]) \geq S(F)+b$, and $a+b = 4$ with $a, b \geq 1$. In $F[l]$, however, $\tilde{x}$ occurs either 1 or 2 times. Hence, $S(\mathit{Reduce}(F[l])) - S(F) \geq a + 1$.

T7 $\dfrac{F}{F[l], F[\bar{l}\,]}$ if $F = \{\{\bar{x}, y, \ldots\}, \{x, y, \ldots\}, \{x, y, \ldots\}, \{\bar{y}, \ldots\}, \ldots\}$ and there is a literal l that occurs in a clause with $\tilde{y}$ and $\tilde{l}$ occurs also in a clause with no $\tilde{y}$.

If l occurs in two clauses together with $\tilde{y}$ then these two clauses are directly satisfied in $F[l]$ and two others by the pure literal rule (first x and then y becomes pure or vice versa). If l occurs in a clause with no $\tilde{y}$ and in a clause with $\tilde{y}$ then these two clauses are directly satisfied by $l = 1$ and at least two others following transformation rules. Altogether, $S(\mathit{Reduce}(F[l]) \geq S(F)+4$, if l occurs in at least two clauses of F.

If, however, l occurs only in one clause of F then $S(\mathit{Reduce}(F[l])) \geq S(F)+3$ but now $\bar{l}$ occurs in at least two clauses and consequently $S(F[\bar{l}\,]) \geq S(F)+2$.

We get $S(\mathit{Reduce}(F[l])) \geq S(F)+4$ and $S(\mathit{Reduce}(F[\bar{l}])) \geq S(F)+1$ or $S(\mathit{Reduce}(F[l])) \geq S(F)+3$ and $S(\mathit{Reduce}(F[\bar{l}])) \geq S(F)+2$.

Lemma 3.5.1. *Let F be a reduced formula with no closed subformulas and each variable occur in three or four clauses. Moreover, let there be at least one variable that occurs in exactly three clauses. Then one of the rules **T1-T7** is applicable.*

Proof. 1. *There are only $(2,1)$-literals in F.*
Firstly, assuming that F is not nearly monotone (cf. Definition 3.5.1) we can conclude that there is some variable x that occurs negatively in a clause together with at least one other literal, say l. If $\tilde{l}$ occurs together with $\tilde{x}$ in no other clause then **T1** applies. Otherwise, $\tilde{x}$ and $\tilde{l}$ occur together in at least two clauses. Three joint occurrences are not possible since F is reduced and that case is covered by the small subformula rule (Subsection 3.5.2). Depending on the combination of positive and negative occurrences, **T2**, **T3**, or **T4** apply, as they cover all combinations.

If, however, F happens to be nearly monotone, **T5** applies: Pick any variable x. Then pick a variable y that occurs together with x in exactly one clause. Such a y exists since, otherwise, x would be part of a star or in a unit-clause. Then pick some arbitrary variable z from the other clause that contains x, but not y.

2. *There is also some $(3,1)$- or $(2,2)$-literal in F.*
Find a $(2,1)$-literal x and a $(3,1)$- or $(2,2)$-literal y such that $\tilde{x}$ and $\tilde{y}$ occur in the same clause. (Such a pair is available since, otherwise, there would be a closed subformula that contains only $(2,1)$-literals.) Let N be the number of clauses where $\tilde{x}$ and $\tilde{y}$ occur together. If $N = 1$ then **T1** or **T6** apply: If $\tilde{y}$ occurs together with x then **T6** applies where $\tilde{y}$ plays the role of l (the precondition of **T6** is fulfilled even if $l = \bar{y}$ since then l is nevertheless a $(2,1)$-, $(3,1)$-, or $(2,2)$-literal, and if $\tilde{y}$ occurs together with $\bar{x}$ then **T1** applies where $\tilde{y}$ plays the role of l' (note that here y can be a $(2,1)$-, $(3,1)$-, or $(2,2)$-literal,

too). If $N = 2$ then **T6** applies. Observe that in **T6** $\tilde{y}$ may occur together with $\bar{x}$ as well as together with x. If $N = 3$ then **T6** or **T7** apply (the existence of l in the side condition of **T7** can be assumed since otherwise there would be a small closed subformula). □

All Variables Occur Exactly Four Times. Now, we assume that F is reduced, it contains no closed subformulas, and that each variable occurs exactly four times.

D1 $\dfrac{F}{F[l], F[\bar{l}]}$ if $F = \{\{\bar{x}, l, \ldots\}, \{x, \ldots\}, \{x, \ldots\}, \{x, \ldots\}, \ldots\}$.

In $F[l]$, the clause $\{\bar{x}, l, \ldots\}$ is satisfied. Moreover, $F[l]$ contains the pure literal x and we can apply the pure literal rule. Clearly, it follows that $S(\mathit{Reduce}(F[l])) \geq S(F) + 4$ and $S(F[\bar{l}]) \geq S(F) + 1$.

D2 $\dfrac{F}{F[x], F[\bar{x}, y]}$ if $F = \{\{\bar{x}\}, \{x, y\}, \{x, \ldots\}, \{x, \ldots\}, \ldots\}$.

There is always an optimal assignment I with $I(x) = 1$ or with $I(x) = 0$ and $I(y) = 1$: Let I' be an optimal assignment with $I'(x) = 0$ and $I'(y) = 0$. Let I be the assignment that coincides with I', except that $I(x) = 1$. Obviously, I satisfies at least as many clauses as I' and is therefore also optimal. Hence, it suffices to examine $F[x]$ and $F[\bar{x}, y]$. We get $S(F[x]) \geq S(F) + 3$ and $S(F[\bar{x}, y]) \geq S(F) + 3$.

D3 $\dfrac{F}{F[x], F[\bar{x}]}$ if $F = \{\{\bar{x}\}, \{x, l, \ldots\}, \{x, \ldots\}, \{x, \ldots\}, \ldots\}$ and $\tilde{l}$ occurs in 1 or 2 clauses that do not contain $\tilde{x}$.

In $F[x]$, three clauses containing x are satisfied and l is a $(1, 1)$-, $(1^+, 0)$-, or $(0, 1^+)$-literal. Some transformation rule satisfies at least one other clause. We get $S(\mathit{Reduce}(F[x])) \geq S(F) + 4$ and, of course, $S(F[\bar{x}]) \geq S(F) + 1$.

D4 $\dfrac{F}{F[x], F[\bar{x}]}$ if $F = \{\{\bar{x}, \ldots\}, \{\bar{x}, \ldots\}, \{x, \ldots\}, \{x, \ldots\}, \ldots\}$.

Obviously, $S(F[x]) \geq S(F) + 2$ and $S(F[\bar{x}]) \geq S(F) + 2$.

D5 $\dfrac{F}{F[y], F[\bar{y}, z], F[\bar{y}, \bar{z}]}$ if $F = \{\{\bar{x}\}, \{x, y, z\}, \{x, \ldots\}, \{x, \ldots\}, \{\bar{y}\}, \{y, \ldots\}, \{y, \ldots\}, \{\bar{z}\}, \{z, \ldots\}, \{z, \ldots\}, \ldots\}$.

Obviously, $S(F[y]) = S(F) + 3$ and $S(F[\bar{y}, z]) = S(F) + 4$. Finally, we get $S(\mathit{Reduce}(F[\bar{y}, \bar{z}])) \geq S(F) + 5$ since $F[\bar{y}, \bar{z}]$ contains a subformula $\{\{\bar{x}\}, \{x\}, \{x, \ldots\}, \{x, \ldots\}\}$ and, applying the complementary unit-clause rule followed by the pure literal rule, satisfies three clauses.

D6 $\dfrac{F}{F[\bar{x}], F[x,\bar{y}], F[x,y,\bar{z}_1,\bar{z}_2,\ldots,\bar{z}_6]}$

if $F = \{\{\bar{x}\},\{x,y,\ldots\},\{x,\ldots\},\{x,\ldots\},$
$\{y,z_1,z_2,z_3,\ldots\},\{y,z_4,z_5,z_6,\ldots\},\{\bar{y}\},\ldots\}$,
F is simple, and each positive clause has size at least 4.

We have to prove the following claim: If there is an optimal assignment I for F with $I(x) = 1$ then there is an optimal assignment I' with $I'(x) = 1$ and $I'(y) = 0$ unless $I(z_1) = \ldots = I(z_6) = 0$. Let us assume that I is indeed an optimal assignment with $I(x) = 1$, but $I(z_1) = \cdots = I(z_6) = 0$ does not hold; without loss of generality let us assume $I(z_1) = 1$. Now define I' as I, but $I'(y) = 0$. When changing from I to I', the clause $\{y, z_4, z_5, z_6, \ldots\}$ may no longer be satisfied. The number of satisfied clauses, however, does not decrease since now $\{\bar{y}\}$ is satisfied. The status of all other clauses does not change. We get $S(F[\bar{x}]) \geq S(F) + 1$, $S(F[x,\bar{y}]) \geq S(F) + 4$, and $S(F[x,y,\bar{z}_1,\bar{z}_2,\bar{z}_3,\bar{z}_4,\bar{z}_5,\bar{z}_6]) \geq S(F) + 11$.

Lemma 3.5.2. *Let F be a reduced formula. Let each variable occur in exactly four clauses and let there be no closed subformula. Then one of the rules **D1-D6** is applicable.*

Proof. If there is at least one $(2,2)$-literal then **D4** applies. Therefore, in the following we can assume that F contains only $(3,1)$-literals.

Let us first assume that F is not nearly monotone. Then **D1** applies.

Next, let us assume that F is nearly monotone, but not simple. Then **D3** applies.

Finally, let F be simple. If a variable x occurs in a clause of size 2 (resp. 3), then **D2** (resp. **D5**) applies. Otherwise, all variables occur positively only in clauses of size at least 4 and **D6** applies. □

The following lemma shows that the relatively inefficient rule **D4** can always be followed by something efficient.

Lemma 3.5.3. *Let F be a reduced formula such that each variable occurs in exactly four clauses and there is some $(2,2)$-literal x. Let F contain no closed subformulas. Then $S(Reduce(F[x])) > S(F[x])$ or $Reduce(F[x])$ contains a $(2,1)$-literal. The same applies for $F[\bar{x}]$.*

Proof. Let $\tilde{y}$ occur together with x in any one clause and also in any one other clause that does not contain x. Then $\tilde{y}$ occurs in $F[x]$ between one and three times. If it occurs one or two times then the pure literal or resolution rule is applicable to $F[x]$. If y occurs three times in $F[x]$ then it is a $(2,1)$-literal. Then it remains a $(2,1)$-literal in $Reduce(F[x])$ unless a reduction that increases $S(F[x])$ was carried out. Analogously, prove the same for $F[\bar{x}]$. □

Input: A formula F
Output: An equisatisfiable formula $B(F) = \{\top, \ldots, \top\}$
Method:
$F \leftarrow \mathit{Reduce}(F)$;
if F is final **then return** F
else if $F = F_1 \oplus F_2 \oplus \cdots \oplus F_m$ **then return** $B(F_1) \cup B(F_2) \cup \cdots \cup B(F_m)$
else if F has less than 6 unresolved clauses **then return** $A(F)$
else
choose an applicable rule $\dfrac{F}{F_1, \ldots, F_r} \in \{\textbf{F1,T1–T7,D1–D6}\}$,
where **D4** is chosen only if no other rule is applicable;
return $\max\{B(F_1), \ldots, B(F_r)\}$
fi

Fig. 3.3. Algorithm B. Note that $F_1 \oplus F_2 \oplus \cdots \oplus F_m$ denotes the decomposition of F into closed subformulas and $\max\{B(F_1), \ldots, B(F_r)\}$ is the multi-set with the maximum number of $\top$'s. Among the applicable rules some rule with minimum branching number is chosen.

3.5.3 The Algorithm and Its Analysis

One key to an efficient algorithm for MaxSat is a suitable data structure to represent formulas in conjunctive normal form. For the high-level description of transformation and branching rules, we used the representation as a multi-set of sets of literals. The actual implementation of the algorithm will use a refinement of this representation. We represent literals as natural numbers. A positive literal x_i is represented as the number i and the negative literal $\bar{x}_i$ by $-i$. A clause is represented as a list of literals and a formula as a list of clauses. The $\top$-clause is represented by a special symbol. Moreover, for each variable there is an additional list of pointers that point to each occurrence of the variable in the formula.

Algorithm B constructs an equisatisfiable final formula $\{\top, \ldots, \top\}$ from a formula F by using transformation and branching rules (see Fig. 3.3).

Lemma 3.5.4. *A formula F can be decomposed into its closed subformulas in linear time.*

Proof. Simply find the connected components in the graph whose nodes are all variables and edges connect variables that occur in the same clause. □

Lemma 3.5.5. *A formula F can be transformed into an equisatisfiable, reduced formula F' with $S(F') \geq S(F)$ in time $O(|F| + |F|(S(F') - S(F)))$.*

Proof. First check if the formula is a star, subsequently check for each variable in constant time if a transformation rule applies to it and if yes, apply it in linear time.

A technique to achieve these bounds easily is a dictionary that can be constructed from a formula in linear time and that can process queries such

Cases	*Branching vector*	*Branching number*
F1, T1, T5, T6, T7, D1, D3	$(4, 1)$	1.39
F1, T1, T3, T7	$(3, 2)$	1.33
T2, D2	$(3, 3)$	1.26
T4	$(4, 2)$	1.28
D4	$(2, 2)$	1.42
D5	$(5, 4, 3)$	1.33
D6	$(11, 4, 1)$	1.40

Table 3.2. Lower bounds on branching vectors for Algorithm B referring to the number of $\top$-clauses $S(F)$.

as "Give me a variable x such that x occurs in C_1, $\tilde{x}$ occurs in C_2 and $\bar{x}$ occurs in C_2, if such a variable exists." It is sufficient to have a dictionary for queries that involve at most four clauses and that answers with variables that occur at most four times in the formula.

For example, one can check in constant time whether the small subformula rule applies to a variable x: Check that $\tilde{x}$ occurs three times. Find the clauses C_1, C_2, and C_3 that contain $\tilde{x}$. Then, using a dictionary, ask the query "Give me two variables that occur exactly in C_1, C_2, and C_3." The rule is applicable iff such a pair exists. □

Algorithm B generates a search tree whose nodes are labeled by formulas that are recursively processed. The children of an inner node F are computed by a transformation rule (one child) or a branching rule (more than one child). The *value* of a node F is $S(F)$. The values of all children of F are bigger than $S(F)$. If the children of F were computed according to a rule

$$\frac{F}{F_1, F_2, \ldots, F_r}$$

then $(S(F_1) - S(F), \ldots, S(F_r) - S(F))$ is the branching vector of this node.

Lemma 3.5.6. *The branching tree of Algorithm B has $O(1.40^k)$ nodes, where k is the number of satisfiable clauses in F.*

Proof. In Table 3.2 we list all branching vectors and numbers corresponding to the branching rules given in Subsection 3.5.2. All branching numbers are smaller than 1.40 which is the branching number of the branching vector $(1, 4, 11)$ (rule **D6**) except the branching number $\sqrt{2} \approx 1.42$ which belongs to nodes that are split according to rule **D4** and whose branching vector is $(2, 2)$.

By Lemma 3.5.3, however, the children of nodes with branching vector $(2, 2)$ have a branching vector of at least $(1, 4)$ or $(2, 3)$, since they are split by any one rule **T1**–**T7**. The combined branching number of the nodes and its children is therefore at most $(3, 6, 3, 6)$, $(3, 6, 4, 5)$, or $(4, 5, 4, 5)$. The corresponding branching numbers are 1.40, 1.39, and 1.37. The largest branch-

ing number remains 1.40 and consequently the size of the branching tree is $O(1.40^k)$. □

Theorem 3.5.1. *The running time of Algorithm B is $O(|F| \cdot 1.40^k)$ and (for a slight modification in the algorithm) $O(|F| \cdot 1.39^K)$, where $|F|$ is the length of the given formula F, k is the number of satisfiable clauses in F and K is the number of clauses in F.*

Proof. The size of each formula in the branching tree does not exceed $|F|$, since transformation and branching rules never generate longer formulas. Reducing the formula (Lemma 3.5.5), selecting and applying a branching rule, or decomposing the formula into minimal subformulas, take time $O(|F|)$.

The size of the tree is at most 1.40^k (Lemma 3.5.6). This proves the time bound $O(|F| \cdot 1.3995^k)$ for Algorithm B.

Let $\mu(F')$ be K minus the number of clauses in F' that are not $\top$. Then, obviously $\mu(F') \geq S(F')$. Hence, the branching numbers in the tree with respect to $\mu(F')$ are at least as big as those with respect to $S(F')$. In the following we analyze Algorithm B with respect to $\mu(F')$ and get in this way a bound on the size of the branching tree in terms of K.

Except for **D4** and **D6** all branching numbers for $S(F')$ and thus for $\mu(F')$ are at most 1.39. The branching vector for **D6** is $(1, 5, 13)$ with respect to $\mu(F')$ (yielding a branching number of 1.34). Thus, it remains to deal with **D4**. This problem occurs only if all variables occur four times.

If there are only $(2, 2)$-literals left we *do* apply **D4**. If a formula in one of the two branches is reducible, then we have at least a $(3, 2)$ or $(2, 3)$-branch. Otherwise, it is only a $(2, 2)$-branch. However, by Lemma 3.5.3 the formulas in *both* branches contain $(2, 1)$-literals. If both formulas do not contain $(2, 2)$-literals, then **D4** will never again be used and this single use plays no role asymptotically.

Let us assume we branch on the $(2, 2)$-literal x and $F[x]$ still contains $(2, 2)$-literals. Then there is a $(2, 2)$-literal y and a $(2, 1)$-literal z such that $\tilde{y}$ and $\tilde{z}$ occur together in the same clause in $F[x]$ (unless there are closed subformulas). Then, there are between one and two clauses in $F[x, y]$ or $F[x, \bar{y}]$ that contain $\tilde{z}$ and therefore one of the two formulas is reducible. In total, if we branch according to $F[\bar{x}]$, $F[x, y]$, and $F[x, \bar{y}]$, we get a branching vector of $(2, 5, 4)$ or $(2, 4, 5)$. The corresponding branching number is 1.39. This settles the case that there are only $(2, 2)$-literals in the formula.

If there are also $(3, 1)$-literals then either **D1** is applicable or all $(3, 1)$-literals occur negatively only in unit-clauses. If that is the case and there are also $(2, 2)$-literals then there is also a clause that contains a $(3, 1)$-literal x and some $(2, 2)$-literal l (otherwise there would exist a closed subformula). Then, rule **D3** can be applied. If, however, no $(2, 2)$-literal exists then **D4** is not applicable and therefore some other rule must be applicable. The branching numbers of all rules except **D4** are, however, at most 1.39. □

Assuming a parameter value $k < K$, the following corollary is of interest.

Corollary 3.5.1. *To determine an assignment satisfying at least k clauses of a boolean formula F in CNF takes $O(k^2 \cdot 1.40^k + |F|)$ steps.*

Proof. Proposition 2.2.1 (Section 2.2) gives a size $O(k^2)$ problem kernel for MaxSat which can be computed in linear time. Hence, running time $O(|F| \cdot \gamma^k)$ can be improved to running time $O(k^2\gamma^k + |F|)$. □

Corollary 3.5.1 improves Theorem 7 of Mahajan and Raman [186] by decreasing the exponential factor from $\phi^k \approx 1.62^k$ to 1.40^k. Analogously, the running time for Max$q$Sat is improved to $O(qk \cdot 1.40^k + |F|)$. Building up on the above rules and increasing the number of case distinctions, Bansal and Raman [29] reported an improvement of the upper bounds: They stated that MaxSat can be solved in $O(1.35^K|F|)$ or $O(1.39^k k^2 + |F|)$ time. Very recently, Chen and Kanj [59] stated a further refinement of these methods, leading to the upper bounds $O(1.35^k|F|)$ and $O(1.37^k k^2 + |F|)$, respectively.

3.5.4 Final Remarks

If we do not demand that the solution be exact rather only approximately correct then it is possible to solve MaxSat in polynomial time. There is a deterministic, polynomial time approximation algorithm for MaxSat with approximation factor 0.7845 [20], indicating that further improvements are possible when making use of a conjecture of Zwick [266]. However, a polynomial-time approximation algorithm with an approximation factor arbitrarily close to 1 will not exist unless $P = NP$ [18]. Dantsin *et al.* show how to improve the MaxSat approximation factor of 0.770 arbitrarily close to 1 using an exponential time algorithm [75]. More precisely, they describe, given a polynomial time α-approximation algorithm, how to construct an $(\alpha + \epsilon)$-approximation algorithm running in time exponential in the number of clauses, where the exponent depends on the ϵ. The idea is to combine a search tree with an approximation algorithm.

The most immediate parameterized version of MaxSat is to determine whether at least k clauses of a CNF formula F with K clauses can be satisfied. The fixed-parameter tractability of MaxSat implies that every problem in the optimization class *MaxSNP* [214] is also fixed-parameter tractable [51]. Mahajan and Raman [186] introduced a more meaningful parameterization, asking whether at least $\lceil K/2 \rceil + k$ clauses of a CNF formula F can be satisfied. This is what we know as a parameterization above a guaranteed value (cf. Subsection 1.5.2 and Section 2.2). The guaranteed value $\lceil K/2 \rceil$, however, can further be lifted (see [186] for details). The fixed-parameter algorithm presented above can also be plugged into this framework.

Finally, the special case Max2Sat deserves particular attention. Firstly, it gives an example where search tree algorithms have been implemented and tested empirically with encouraging results [128, 132]. Secondly, for Max2Sat "parameterization" so far has found its limitations. An upper

bound of the order $2^{K_2/5}$ for Max2Sat, where K_2 is the total number of 2-clauses, was proven [131]. For MaxSat, Chen and Kanj [59] give the parameterized bound $2^{k/2.15}$ which is better than their "unparameterized" bound $2^{K/2.36}$ when $k < 0.92K$, where K is the total number of clauses. In [132], the parameterized bound $2^{k/2.73}$ for Max2Sat has been proven. However, the above "unparameterized" bound $2^{K_2/5}$ is better for all reasonable values of k: the parameterized bound is better only when $k < 0.55K_2$, while an assignment satisfying $0.5K + 0.25K_2 \geq 0.75K_2$ clauses can be found in a polynomial time [186, 264]. As $\lceil K/2 \rceil$ clauses can be easily satisfied, Mahajan and Raman [186] propose to ask in the parameterized version of the problem for an assignment satisfying $\lceil K/2 + k' \rceil$ clauses. Taking the parameterized bound shown in [132] and plugging it into the results by Mahajan and Raman, we can translate it into a bound with respect to this new parameter k'; in time $2^{6k'/2.73} = 2^{k'/0.45}$ one can find an assignment to the variables that satisfies at least $\lceil K/2 + k' \rceil$ clauses or one can determine that no such assignment exists. However, for $k' \leq \lceil K_2/4 \rceil$, this question still can be handled in polynomial time. Comparing for $k' > \lceil K_2/4 \rceil$ the bound $2^{k'/0.45}$ to the Max2Sat bound shown we see, again, that the parameterized bound is worse for every parameter value. It would be interesting, however, to consider, for a given k'', the parameterized complexity of the question whether there is an assignment satisfying $\lceil K/2 + K_2/4 \rceil + k''$ clauses.

3.6 Dominating Set on Planar Graphs

We finish the series of search tree examples with Dominating Set on planar graphs. This problem stands for cases where the bounded search tree algorithm does *not* employ many case distinctions and thus is comparatively easy. By way of contrast, the proof of correctness of the considered branching and analyzing the corresponding search tree size is hard. We follow parts of [5].

Recall from Section 3.1 the problems we face with handling Dominating Set instead of Independent Set on planar graphs. The difficulties described there lead us to the study of a more general version of Dominating Set, i.e., Annotated Dominating Set:

> **Input:** A graph $G = (B \uplus W, E)$ with its vertices either colored black or white and a nonnegative integer k.
> **Question:** Is there a choice of at most k vertices $V' \subseteq V = B \uplus W$ such that, for every vertex $u \in B$, there is a vertex $u' \in N[u] \cap V'$? In other words, is there a set of at most k vertices (which may be either black or white) that dominates the set of black vertices?

Then, in each step of the search tree, we would like to branch according to a low degree black vertex. Restricting Annotated Dominating Set to planar graphs we can guarantee the existence of a vertex $u \in B \uplus W$ with $\deg(u) \leq 5$. However, as long as *not all* vertices have degree bounded by five

this vertex needs not necessarily be black. Hence, a direct $O(6^k n)$ search tree algorithm for (Annotated) Dominating Set seems out of reach for planar graphs.

In what follows, we sketch a fixed-parameter algorithm for (Annotated) Dominating Set on planar graphs with running time $O(8^k n^2)$. For that purpose, in analogy to MaxSat (cf. Subsection 3.5.2) we provide a set of transformation rules and, then, use a bounded search tree in which we are constantly simplifying the instance according to the transformation rules. The branching in the search tree will be done with respect to low degree vertices. More precisely, we always choose a black vertex with minimum degree seven and branch on it by either putting itself or one of its at most seven neighbors into the dominating set. This yields eight cases to branch into and the search tree size 8^k follows. The central technical obstacle herein that has to be surmounted is to prove that whenever we want to do a branching on a minimum degree black vertex then always a degree at most seven black vertex must exist. To this end, we introduce a set of easy transformation rules in order to continuously generate a "reduced" graph which possesses such a degree seven black vertex. Then, by fairly technical means based on "Euler arguments" for planar graphs it is possible to prove that planar reduced black-and-white graphs always contain a vertex with maximum degree seven. The technically demanding proof of this property is omitted and can be found in [2, 5].

Without loss of generality, we consider connected planar graphs, i.e., connected graphs that admit crossing-free embeddings in the plane.

3.6.1 Transformation Rules

We consider the following transformation rules for simplifying the Annotated Dominating Set problem on planar graphs. In developing the search tree, we will always assume that we are branching from a reduced instance (thus, we are constantly simplifying the instance according to the transformation rules). When a vertex u is placed in the dominating set D by a transformation rule then the target size k for D is reduced to $k-1$ and the neighbors of u become colored white.

T1 Delete edges between white vertices.
T2 Delete a degree one white vertex.
T3 If there is a degree one black vertex w with neighbor u (either black or white) then delete w, place u in the dominating set, and lower k to $k-1$.
T4 If there is a white vertex u of degree 2, with two black neighbors u_1 and u_2 connected by an edge $\{u_1, u_2\}$ then delete u.
T5 If there is a white vertex u of degree 2, with black neighbors u_1, u_3, and there is a black vertex u_2 and edges $\{u_1, u_2\}$ and $\{u_2, u_3\}$ in G then delete u.

T6 If there is a white vertex u of degree 2, with black neighbors u_1, u_3, and there is a white vertex u_2 and edges $\{u_1, u_2\}$ and $\{u_2, u_3\}$ in G then delete u.

T7 If there is a white vertex u of degree 3, with black neighbors u_1, u_2, u_3 for which the edges $\{u_1, u_2\}$ and $\{u_2, u_3\}$ are present in G (and possibly also $\{u_1, u_3\}$) then delete u.

Let us call a set of simplifying transformation rules of a certain problem *sound* if whenever (G, k) is some problem instance and instance (G', k') is obtained from (G, k) by applying one of the transformation rules then (G, k) has a solution iff (G', k') has a solution. The following is easily shown by a simple case analysis.

Lemma 3.6.1. *The transformation rules are sound.* □

Suppose that G is a *reduced* graph, that is, none of the above transformation rules can be applied.

Lemma 3.6.2. *Let $G = (B \uplus W, E)$ be a plane black and white graph. If G is reduced, then the white vertices form an independent set and every triangular face of $G[B]$ is empty.*

Proof. The result easily follows from the transformation rules **T1**, **T2**, **T4**, and **T7**. □

Lemma 3.6.3. *Applying transformation rules* **T1**–**T7***, a given black and white graph $G = (B \uplus W, E)$ can be transformed into a reduced graph $G' = (B' \uplus W', E')$ in time $O(n^2)$, where n is the number of vertices in G.*

Proof. The result is easy to see if we perform the transformation in the following order: First apply rule **T1**, and then, visit every white vertex, checking whether rules **T4**–**T7** can be applied. Finally, carry out rules **T2** and **T3**. □

3.6.2 Main Result and Final Remarks

Based on the above transformation rules, the following technical, highly nontrivial lemma can be established (for the proof see [2, 5]):

Lemma 3.6.4. *If $G = (B \uplus W, E)$ is a planar black and white graph that is reduced then there exists a black vertex $u \in B$ with degree at most seven.* □

Interestingly, there exists an infinite set of plane reduced black and white graphs with the property that all black vertices have degree seven (see [2, 5]). Hence, in this limited sense, the upper bound provided in Lemma 3.6.4 is optimal since these examples giving matching lower bounds. Note, however, that it is completely open to prove (if, after all, possible) the existence of a family of graphs which keeps this property after *each* branching that is

performed in the course of the algorithm working on the graph. Moreover, it is open whether extending the given reduction rules may lead to reduced graphs with smaller maximum "black degree." In this way, we obtain the main result.

Theorem 3.6.1. (Annotated) Dominating Set *on planar graphs can be solved in time* $O(8^k n^2)$.

Proof. Use Lemma 3.6.4 for the construction of a search tree as described in the beginning of the section. Note that performing the transformation in each node of the search tree, by Lemma 3.6.3, can be done in time $O(n^2)$. □

We remark that slightly changing the above reduction rules and doing a more refined analysis of the quadratic time factor n^2 can be improved to a linear one [145] (also cf. [2]).

The above sketched transformation rules lead to the first search tree algorithm with correct bound on the search tree size for Dominating Set on planar graphs. It improves on the original, flawed theorem stating an exponential term 11^k [87, 88] which is now lowered to 8^k. Unfortunately, the proof of correctness has become fairly technical. Since the "optimality" of Lemma 3.6.4 only holds with respect to the particular set of transformation rules given above, it remains open to improve Lemma 3.6.4 by adding further, more involved transformation rules. Moreover, a generalization of the above considerations is possible and yields analogous bounded search tree algorithms for Dominating Set on graphs of bounded genus [97]. Finally, we stress that the above algorithm is fairly easy to implement [176]—specifically, in comparison with the complicated case distinctions employed by the algorithms in Section 3.4 and 3.5 which might need some "re-engineering" when applied in practice (cf. discussion in Section 3.8).

3.7 Interleaving Search Trees and Kernelization

We now have seen several different problem types where the bounded search tree paradigm applies. In Chapter 2, before we encountered several examples for the reduction to problem kernel paradigm. One may say that these two paradigms form the ground pillars of "feasible fixed-parameter tractability." One obvious way to combine these two methods is, as already indicated, firstly, to do a preprocessing of the given input instance by performing a reduction to problem kernel, and, secondly, to systematically process the generated problem kernel using bounded search trees. What we show next is that to do a kernelization repeatedly during the course of the search tree algorithm may further accelerate the solution finding process for the given problem. We follow [205] in the subsequent presentation.

3.7.1 Basic Methodology

In the following, we will deal with a large class of fixed-parameter algorithms. Let us summarize the conditions that these algorithms have to undergo: They have to be fixed-parameter algorithms that work in two stages, *reduction to problem kernel* and *bounded search tree.* Reduction to problem kernel takes $P(|I|)$ steps and results in an instance of size at most $q(k)$, where both P and q are polynomially bounded. The expansion of a node in the search tree takes $R(|I|)$ steps, which must also be bounded by some polynomial, the search tree size being $O(\alpha^k)$. The overall time complexity of the algorithm is then

$$O(P(|I|) + R(q(k))\alpha^k),$$

where (I, k) is the instance to be solved. In the following we show how to modify the second stage of the algorithm in order to improve the time complexity to

$$O(P(|I|) + \alpha^k).$$

Generally, we now use the following algorithmic steps to expand a node (I, k) in the search tree:

if $|I| > c \cdot q(k)$ **then** replace (I, k) with $\mathcal{R}(I, k)$ **fi**;
replace (I, k) with $(I_1, k - d_1), (I_2, k - d_2), \ldots, (I_i, k - d_i)$

Here $c \geq 1$ is a constant that can be chosen with the aim of further optimizing the running time. There is a tradeoff in choosing c: The optimal choice depends on the implementation of the algorithm but in the end it affects only the constant factor in the overall time complexity. Therefore we neglect optimizing c here.

A closer look shows that we in fact seem to *increase* the time needed to expand a node in the search tree. This is generally speaking true: Sometimes we apply reduction to problem kernel prior to branching into recursive calls. However, these additional kernelizations also *decrease* the instance size in the middle of the search tree. Since the time for branching is bounded polynomially in the *instance size*, this also helps to *decrease* the time to expand a node. It proves to be the case that, while we waste time near the root of the search tree, we gain much more time near the leaves. Note that the technique of *interleaving* reduction to problem kernel and bounded search trees was already used for developing efficient fixed-parameter algorithms for Vertex Cover [92, 249] (also called *rekernelization* there). There, however, it was used to reduce the number of case distinctions in the search tree; it was not considered with the aim of removing the factor $R(q(k))$ as we do.

In order to analyze the running time of the above approach mathematically, we describe the time to expand a node (I, k) and all its descendants by

a recurrence. Let T_k denote an upper bound on the *time* to process (I, k). The following recurrence holds for T_k:

$$T_k = T_{k-d_1} + T_{k-d_2} + \cdots + T_{k-d_i} + O(P(q(k)) + R(q(k)))$$

The time to expand (I, k) itself is at most $O(P(q(k)) + R(q(k)))$ because $|I| = O(q(k))$ since $|I| > c \cdot q(k)$ is constantly prevented. In order to solve this non-homogeneous linear recurrence we need a special solution. To get its general solution we add the general solution of the corresponding homogeneous recurrence $T_k = T_{k-d_1} + T_{k-d_2} + \cdots + T_{k-d_i}$. However, we already know that all solutions of this homogeneous recurrence are bounded by $O(\alpha^k)$. Consequently, we only need to find a small special solution of the non-homogeneous recurrence. In our case the inhomogeneity is a polynomial. Therefore, there exists a special solution that is also a polynomial in k. It is easy to construct such a special solution explicitly. There is always a polynomial solution that has the same degree as the inhomogeneity p. (If r is a polynomial special solution then $r(k) - \sum_{j=1}^{i} r(k - d_j) = p(k)$ and the highest degree monomials on the left side cannot cancel each other.) All solutions of T_k are therefore bounded by $O(\alpha^k)$.

In order to illustrate this, let us consider the following recurrence.

$$T_k = 2T_{k-1} + C \cdot k^2 + D \cdot k + E,$$

where C, D and E are constants that depend on the implementation of the algorithm. The initial conditions are simple, say, $T_0 = 0$. The reflected characteristic polynomial is $1 - 2z$ and its unique root is $1/2$. The general solution of the homogeneous recurrence is $\lambda 2^k$ for $\lambda \in \mathcal{R}$. Since it is a recurrence of first order, the dimension of its space of solutions is one, too.

A special solution is $T_k = -Ck^2 - (4C + D)k - (6C + 2D + E)$. The general solution is then $\lambda 2^k - Ck^2 - (4C + D)k - (6C + 2D + E)$ and the solution for $T_0 = 0$ is $T_k = (6C + 2D + E) \cdot 2^k - Ck^2 - (4C + D)k - (6C + 2D + E)$.

3.7.2 Interleaving is Necessary

Next, we show that an improved analysis alone cannot achieve the speedup of the last section. That is, the interleaving of reduction to problem kernel and the bounded search tree really is necessary to get the claimed improvements. Without modification, the algorithms in general have a running time of $\Omega(P(|I|) + R(f(k))\alpha^k)$. As an example, we can use Vertex Cover and we assume that we use the trivial size 2^k search tree algorithm.

Look at Fig. 3.4 for a definition of a family of instances of a Vertex Cover defined for odd k. There is no solution of size at most k, since the optimal vertex cover has size $\frac{5}{2}k - \frac{3}{2}$ (in the head $k - 2$ vertices and half the vertices of the tail). The graph contains exactly $(k - 1)(k - 2) + 1$ vertices in the head and $3k + 1$ vertices in the tail (altogether $k^2 + 4$). Reduction to

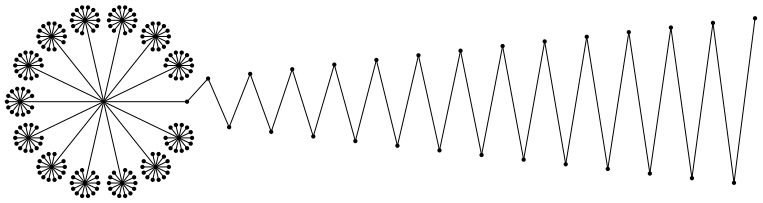

Fig. 3.4. An instance of Vertex Cover. The following graph is the $k = 15$ member of a family of instances (G_k, k) for Vertex Cover. The graph G_k consists of a tree with degree $k-1$ and depth 2 to which a path with $3k+1$ vertices is attached (called the *tail*). It is easy to see that the smallest vertex cover for G_k has size $\frac{5}{2}k - \frac{3}{2}$ and therefore the whole family has no members in Vertex Cover.

problem kernel does not affect this graph since the degree of every vertex is at most k although its size is very near the maximum possible $k(k+1)$. Now, assume that the unmodified algorithm chooses edges from right to left. This leads to a search tree of size 2^k, the largest possible. While the algorithm examines this graph, it removes vertices and edges but the *head* remains unchanged. Consequently, instances have size $\Omega(k^2)$ during *each* branching step. The overall time complexity therefore is the worst possible—$\Omega(k^2 2^k)$. Of course, a better time complexity can also be achieved by changing the order of choosing edges. Nevertheless, the time bound is $\Theta(k^2 2^k)$ in the worst case.

After the modification the running time is decreased tremendously. After the *second* edge is removed and k being decreased by two, the whole head is removed from the graph.

3.7.3 Applications and Final Remarks

The presented interleaving technique applies in numerous settings and it already led to speedups in the solutions of (Weighted) Vertex Cover [60, 204, 207], 3-Hitting Set [208], Maximum Satisfiability [29, 59, 206], Constraint Bipartite Vertex Cover [111] and a closely related problem [58], k-Leaf Spanning Tree [108], and several others. It thus belongs into the tool-box for the development of efficient fixed-parameter algorithms. In this context, it is important to note that the achieved improvements when replacing $O(\alpha^k \cdot q(k) + p(n))$ by $O(\alpha^k + p(n))$ are *not* due to asymptotic tricks, but that $q(k)$ can be replaced by a *small* constant. And the improvement really matters. Simply compare a time $O(2.27^k \cdot k^3 + n)$ (without interleaving, employing the search tree of size 2.27^k (Section 3.4) and the problem kernel of size $O(k^3)$ (Section 2.3)) with a time $O(2.27^k + n)$ (with interleaving) algorithm for 3-Hitting Set.

Finally, applying the interleaving technique still needs carefulness. It would have been tempting to apply it to Dominating Set on planar graphs,

for which we know a linear size problem kernel (see Section 2.5) and a size 8^k search tree (see Section 3.6). We cannot (directly) apply interleaving because the search tree works with black-and-white graphs, whereas the problem kernelization only was designed for non-colored graphs. It seems possible, however, that the reduction to problem kernel as sketched in Section 2.5 can be extended to black-and-white graphs. This, however, remains to be shown in future work.

In summary, as a rule, the potential of improvement due to interleaving increases the larger the problem kernel of the underlying parameterized problem is. It probably always pays off in practice when perhaps not applied at every search tree node but in a regular manner after some branchings. In this way, the additional administrative overhead can be compensated. The best tradeoff, in the end, has to be determined empirically.

3.8 Concluding Discussion

Perhaps the main conclusion from the preceding sections is that the development of bounded search tree fixed-parameter algorithms can mean a highly nontrivial task. Sometimes the more challenging part is to prove the correctness and the search tree size of the proposed method (cf. Sections 3.3 and 3.6) and sometimes the more challenging part is to design and and overview intricate case distinctions leading to good worst-case estimates for the search tree size (cf. Sections 3.4 and 3.5). The latter type of search trees—there are several more examples of these (e.g., [10, 60, 111, 204, 207])—leads to the following question that should be pursued in future research. Given some complicated branching strategy employed by a bounded search tree algorithm, how can one achieve the same or "nearly the same" bounds on the search tree size but with a significantly simplified case distinction? One may consider this as a kind of "re-engineering" of case distinctions and it might turn into a fruitful research topic with particular importance for the practical side of search tree algorithms.

Other points in considering the practical sides of search tree algorithms are as follows. Search tree algorithms ...

- ... can often be further accelerated by incorporating standard heuristic techniques such as branch and bound (cf., e.g., [133])
- ... are easily run on parallel machines because of the straightforward load balancing which is directly implied by the construction process of search trees (cf. [79]).
- ... can easily be combined with approximation algorithms by stopping the recursive search "near" the leaves and running an approximation algorithm instead (cf. [75]).
- ... in practice frequently have few cases of their case distinction that occur very often and the remaining cases occur very seldomly [234].

- ... are still the most important technique to cope with the really hard kernel of a problem.

Dealing with bounded search tree algorithms, it might sometimes appear as unsatisfactory that there are no clear lower bounds on the search tree size. For instance, the upper bounds on the search tree sizes for Vertex Cover have been continuously improved in a series of papers [30, 92, 204, 249, 60]—and the same holds for Maximum Satisfiability [51, 186, 206, 29, 59]. Thus, the impression could be that, at the cost of further extending (which usually means further complicating) the case distinctions there will always be some (tiny) progress achievable. Note, however, that it is not always only a matter of refining case distinctions in order to get the search tree size down—sometimes elegant, practically relevant techniques such as the Nemhauser-Trotter problem kernel reduction for Vertex Cover came along with these efforts (cf. [60]). Generally speaking, the research community has to decide whether there is enough innovation in a newly proposed search tree—elegance matters. Unfortunately, to prove, for instance, a lower bound for the search tree size of Vertex Cover of size say 1.2^k or 1.1^k seems out of reach of current possibilities.

4. Further Algorithmic Techniques

So far, we have concentrated on the currently two "main paradigms" for the development of efficient fixed-parameter algorithms—reduction to problem kernel and bounded search trees. There are, however, several more tools and techniques to derive fixed-parameter tractability and which carry or already have shown the potential for practical applicability. Here, we focus on integer linear programming (Section 4.1), dynamic programming (Sections 4.2 and 4.5), color-coding and hashing (Section 4.3) and tree decompositions of graphs (Section 4.4). In particular, we omitted the famous and powerful machinery of graph minor theory and related topics and also the elegant theory of monadic second order logic—partly justified by the seemingly limited scope of practical applicability of these and partly also due to the extensive requirements in presenting these techniques; refer to the monograph [88] for more on these theoretically highly interesting topics.

4.1 Integer Linear Programming

In spite of the enormous significance that integer linear programming generally has for approximation algorithms and combinatorial optimization (see [200, 213, 237] for some surveys) it has nearly been neglected in the context of fixed-parameter algorithms. One first link will be described now and it will be illustrated using the aforementioned Closest String problem.

There is a famous result of Lenstra [177][1] that applies to fixed-parameter algorithms (also see [161, 172] for more details). Lenstra's result basically says that integer linear programs (ILP's for short) with a constant number of variables can be solved in linear time. More precisely, with Kannan's [161] improvements we have the following theorem. It refers to the integer program feasibility problem where one has to decide on the existence of (not necessarily optimal) solutions fulfilling all constraints given by linear inequalities.

Theorem 4.1.1. (Lenstra) *The integer programming feasibility problem can be solved with $O(p^{9p/2}L)$ arithmetic operations in integers of $O(p^{2p}L)$ bits in*

[1] It won the Fulkerson Prize 1985 as an outstanding paper in the area of discrete mathematics.

size, where p is the number of ILP variables and L is the number of bits in the input. □

Note that this fixed-parameter result also needs space exponential in the parameter p.

Giving a detailed exposition of integer linear programming and, in particular, Lenstra's result is beyond the scope of this work. By way of contrast, we here consider this theory and Theorem 4.1.1 more or less as a black box and we apply it to one (and so far seemingly the only one published) concrete example that applies it to derive fixed-parameter tractability results. In this way, we also illustrate the notion of ILP's and we see that still some nontrivial work has to be done in order to apply Theorem 4.1.1 in our fixed-parameter sense.

The example problem we consider is Closest String (cf. Subsection 1.5.3 and Section 3.3). For convenience, we recall the definition here[2]:

> **Input:** k strings $s_1, s_2, \ldots, s_k$ over alphabet Σ of length L each, and a nonnegative integer d.
> **Question:** Is there a string s such that $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$?

The goal is to give an ILP formulation for Closest String such that the number of variables solely depends on the parameter value k, the number of input strings. The key to this lies in the notion of column types. Given a set of k strings of length L, we can think of these strings as a $k \times L$ character matrix. By *columns* of a Closest String instance we refer to the columns of this matrix. In the following, we state that after reordering the columns of the Closest String instance, we can easily obtain solutions for the original instance from solutions for the reordered instance. For reordering the columns, we introduce a permutation on strings as follows. Given a string $s = c_1 c_2 \ldots c_L$ of length L with $c_1, \ldots, c_L \in \Sigma$ and a permutation $\pi : \{1, \ldots, L\} \to \{1, \ldots, L\}$. Then, $\pi(s) = c_{\pi(1)} c_{\pi(2)} \ldots c_{\pi(L)}$. The following lemma is obvious.

Lemma 4.1.1. *Given a set of strings $S = \{s_1, s_2, \ldots, s_k\}$, each of length L, and a permutation $\pi : \{1, \ldots, L\} \to \{1, \ldots, L\}$. Then s is an optimal closest string for $\{s_1, s_2, \ldots, s_k\}$ iff $\pi(s)$ is an optimal closest string for $\{\pi(s_1), \pi(s_2), \ldots, \pi(s_k)\}$.* □

Several columns can be identified due to *isomorphism*. The reason for this is the fact that the columns are independent from each other in the sense that the distance from the closest string is measured columnwise. For instance, consider the case of the two columns $(a, a, b)^t$ and $(b, b, a)^t$ when $k = 3$. Clearly, these two columns are isomorphic because they express the same structure except that the symbols play different roles. For finding the optimal

[2] We use the term "closest" string here for a string which has Hamming distance *at most* d to all given strings.

closest string, however, only the structure matters. Isomorphic columns form *column types.*

This can be generalized as follows. W.l.o.g., let a always denote the letter that occurs most often in a column, let b always denote the letter that has the secondly most often occurrences and so on. This property of being *normalized*, as we will refer to it in the following, can be easily achieved by a simple linear time preprocessing of the input instance. In addition, solving the normalized problem optimally, one again can compute the optimal solution of the original problem instance by simply reversing the above mapping done by the preprocessing. Hence:

Lemma 4.1.2. *To compute an optimal closest string it is sufficient to solve a normalized and reordered instance. From this, the solution of the original instance can be derived in linear time.* □

In the following, we call two input instances *isomorphic* if there is a one-to-one correspondence between the columns of both instances such that each thus determined pair of columns is isomorphic. The following lemma shows that it is sufficient to solve an instance with alphabet size $|\Sigma'| \leq k$.

Lemma 4.1.3. *A* Closest String *instance with arbitrary alphabet* Σ*,* $|\Sigma| > k$*, is isomorphic to a* Closest String *instance with alphabet* Σ'*,* $|\Sigma'| = k$*.*

Proof. Assume that there is an input instance with $|\Sigma| > k$. Clearly, in each column appear at most k different symbols from Σ. Since columns are independent from each other, to solve the underlying closest string problem it suffices to represent the "logical structure" of a column by an isomorphic input instance. To do this, at most k symbols per column are enough. □

Example 4.1.1. For $k = 3$, the set of all possible column types for a Closest String instance consists of

$$(a, a, a)^t, (a, a, b)^t, (a, b, a)^t, (b, a, a)^t, (a, b, c)^t.$$

□

Generally, the number of column types for k strings depends only on k (namely, it is given by the so-called Bell number $B(k) \leq k!$, cf., e.g., [210]). Using the column types, Closest String can be formulated as an ILP having only $B(k) \cdot k$ variables. Let the underlying alphabet be Σ. The ILP can be formulated as follows. It uses $B(k) \cdot k$ variables $x_{t,\varphi}$, where t denotes a column type and $\varphi \in \Sigma$. The value of $x_{t,\varphi}$ denotes the number of columns of column type t whose corresponding character in the desired solution string of Closest String is set to φ. Thus, the ILP seeks to minimize

$$\max_{1 \leq i \leq k} \sum_{t} \sum_{\varphi \in \Sigma - \{\varphi_{t,i}\}} x_{t,\varphi},$$

where $\varphi_{t,i}$ denotes the alphabet symbol at the ith entry of column type t. The following two constraints have to be fulfilled when minimizing the above function.

1. All variables $x_{t,\varphi}$ have to be nonnegative integers.
2. Let $\#_t$ denote the number of columns of type t in the input instance (taking into account isomorphism as described before). Then,

$$\sum_{\varphi \in \Sigma} x_{t,\varphi} = \#_t$$

for every column type t.

Actually, Theorem 4.1.1 refers to the integer linear programming feasibility problem and, moreover, a Closest String instance also gives the maximum distance d allowed. Thus, we may obtain the following "feasibility formulation" where the above two constraints remain unchanged but the goal function that had to be minimized now translates into a third set of constraints, namely:

$$\sum_{t} \sum_{\varphi \in \Sigma - \{\varphi_{t,i}\}} x_{t,\varphi} \leq d$$

for every string i, $1 \leq i \leq k$. Altogether, this yields fixed-parameter tractability for Closest String with respect to parameter k. Note, however, that the combinatorial explosion in k is huge and this approach appears to be impractical for $k > 4$ as some experimental investigations indicated. In this case, however, ILP heuristics such as branch-and-bound strategies may extend the range of practical applicability, still using (although not the algorithm behind Lenstra's theorem) the above ILP formulation.

The above ILP approach, however, at least serves as a tool to help deciding whether a problem is fixed-parameter tractable and maybe after that it is possible to come up with a more efficient, direct approach to solve the given problem. As to Closest String, the ILP approach is the only one known to us that yields fixed-parameter tractability with respect to parameter k. In [137], a direct combinatorial approach (avoiding ILP's) was given for $k = 3$ but already $k = 4$ remained open due to the enormous combinatorial complexity. Finally, note that there is an alternative ILP formulation for Closest String, given by Ben-Dor *et al.* [34], where the variables have only binary values but the number of variables is $|\Sigma| \cdot L$ (for alphabet Σ and string length L). Hence, this ILP formulation does *not* imply the fixed-parameter tractability of Closest String with respect to parameter k. In conclusion, it remains open to give further examples besides Closest String where the described ILP approach turns out to be applicable. More generally, it would be interesting to see more connections between fixed-parameter algorithms and integer linear programming.

4.2 Shrinking Search Trees by Dynamic Programming

The fundamental algorithmic paradigm of dynamic programming also plays a prominent role in fixed-parameter applications. In this chapter, we describe two fairly different settings where dynamic programming occurs. In this section, we show how it can be used to shrink the sizes of search trees, a technique that goes back to Robson [231] and which was introduced into the fixed-parameter context by the conference version of the paper by Chen *et al.* [60]. The application there, however, was flawed. Still, it can be employed in certain settings at the cost of modestly exponential space, thus trading space for time (i.e., search tree size). We give the basic ideas and some technical details once more referring to Vertex Cover. We need two ingredients, namely that Vertex Cover has a "regular" search tree algorithm [203, 204] (the one by [60] is not suitable) and that Vertex Cover has a linear size problem kernel consisting of $2k$ vertices (cf. Section 2.4, Theorem 2.4.1).

Note that search tree algorithms as described in Chapter 3 and, in particular, as available for Vertex Cover [60, 203, 204] do yield exponential running times, but only use a polynomial amount of space. This is true because working through a search tree in a depth-first manner only requires to store the data related to a path of bounded length. Robson introduced the idea to improve the running time by dynamic programming [231]: Choose a size s (i.e., number of vertices) and store all induced subgraphs of the input graph G of size s in a database D. Solve all instances in D and store an optimal solution for each of them. Then apply a "regular" search tree algorithm for Vertex Cover, given graph G. Such a regular algorithm finds an optimal solution for G by recursively computing optimal solutions for induced subgraphs of G. Regular search tree algorithm here means that the only operation to reduce the size of a graph is the deletion of vertices. Normally, the size of the graph is reduced further in the branches of the search tree until the sizes of the graphs—note that each node of the search tree corresponds to an induced subgraph of G—in the leaves reach 0. Having the database D at disposal, the search tree algorithms can stop earlier: As soon as the size of the graphs in the nodes of the search tree are as small as s, a dictionary lookup replaces the remaining part of the search tree. In this way, one may save time to (re-)compute optimal vertex covers for the same (small) induced subgraphs again and again—this being the fundamental idea of dynamic programming in general. Clearly, this cuts down the size of the search tree at the cost of storing optimal solutions for all induced subgraphs of G consisting of up to s vertices.

Before we continue our description of how to transfer Robson's technique into the fixed-parameter context we point out a subtle issue concerning the applicability of the whole scenario. The fastest fixed-parameter algorithm for (unweighted) Vertex Cover that uses only polynomial space takes time $O(1.2852^k + kn)$ [60]. Unfortunately, it does not seem possible to apply dynamic programming to this algorithm as attempted in the conference version

of [60] since this algorithm uses a technique, called "folding" by the authors, that contracts edges. This leads to graphs that are not induced subgraphs of the original instance. In a recent improvement of his algorithm, Robson had to use very complicated case distinctions to be able to avoid having to use foldings—otherwise he would not have been able to apply dynamic programming [233] (see [232] for the fastest exact algorithm for INDEPENDENT SET). The second fastest polynomial-size fixed-parameter algorithm for VERTEX COVER [204] (see [203] for a complete version of the paper) does not use foldings and applying dynamic programming to it results in the fastest fixed-parameter algorithm for VERTEX COVER as of today. The resulting running time is $O(1.2832^k k + kn)$. Hence, the latter algorithm is a valid candidate to try to speed it up by Robson's technique as is the trivial 2^k search tree size algorithm for VERTEX COVER as described in Section 1.4 and Chapter 3.

In the remainder of this section, we explain the dynamic programming algorithm in detail because Robson's technique cannot be directly applied to parameterized problems. Instead of pruning the search tree and to look up the optimal vertex cover of the remaining graph in a database when the *size* of the graph drops below some predetermined size, in the parameterized setting we prune the search tree when the *parameter* reaches or drops below some predetermined value. Here, we make use of the fact that VERTEX COVER possesses a problem kernel of size $2k$; that is, w.l.o.g. we may assume that the given input graph for the search tree algorithm contains at most $2k$ vertices. Thus, we can store all induced subgraphs of size up to $s = \alpha k$ in the database and, as long as $\alpha < 1/2$, there are at most $2 \cdot \binom{2k}{\alpha k}$ many of them. The ratio between s and k is called α and we will choose α rather than s directly. Now fix that the threshold for the parameter value that implies pruning of the search tree shall be $\alpha k/2$, where k is the initial parameter value.

Next, we can estimate the running time of the "pruned search tree algorithm with table look up" (which uses dynamic programming) as follows. Let $\hat{T}(k)$ be the running time of the search tree algorithm and let $T(k)$ be the time for the new algorithm that combines search trees with dynamic programming. The new algorithm works as follows:

1. Build a dictionary of all induced subgraphs of up to αk vertices. Compute (using the same algorithm recursively) optimal solutions for all of them and store them in a database if the size of the optimal solution is at most $\alpha k/2$. This takes time $O\big(T(\alpha k/2)\binom{2k}{\alpha k}k\big)$ because it takes time $O(k)$ to store such an induced subgraph, there are at most $2 \cdot \binom{2k}{\alpha k}$ induced subgraphs of size up to αk as long as $\alpha < 1/2$, and it takes only $O(T(\alpha k/2))$ time to find a solution of size up to $\alpha k/2$—if the optimal solution is greater than $\alpha k/2$ then we can stop after $O(T(\alpha k/2))$ steps.
2. Apply the search tree algorithm to G. If the parameter in a branch reaches $\alpha k/2$, look up an optimal solution in the database. This works because the size of the graph in question can be at most αk (implied by the problem kernel size) and is therefore stored in the database if it has a

vertex cover of up to $\alpha k/2$ vertices. The resulting search tree has size $\hat{T}(k - \alpha k/2)$.

To optimize the running time of this algorithm it is crucial to choose αk wisely. Increasing α makes the database bigger and increases the time to build it. Decreasing α increases the time for the search tree algorithm as the size of the search tree grows.

The running time of this algorithm is

$$\begin{aligned} T(k) &\leq \hat{T}(k - \alpha k/2) \cdot O(k) + T(\alpha k/2) \cdot \binom{2k}{\alpha k} \cdot 2 \\ &= \hat{T}(k - \alpha k/2) \cdot O(k) + T(\alpha k/2) \cdot O\left(k^{-1/2}\left(\frac{4}{\alpha^{\alpha}(2-\alpha)^{2-\alpha}}\right)^{k}\right), \end{aligned}$$

where the last O-term is obtained using Stirling's formula and the extra $O(k)$ factor is the time needed to look up a graph in the database (using, e.g., a trie data structure or perfect hashing) as well as for the work done by the search tree algorithm in each node of the search tree. This factor cannot be avoided by the interleaving technique presented in Section 3.7 because the technique is roughly based on the fact that the work done near the leaves in a search tree is asymptotically dominating as nearly all nodes are near the leaves *and* that the size of the graphs processed in those nodes is very small. Now their size is no longer small but up to αk vertices big.

We have to choose α such that both terms in the above recurrence, namely $\hat{T}(k - \alpha k/2)$ and $T(\alpha k/2) \cdot \left(4/(\alpha^{\alpha}(2-\alpha)^{(2-\alpha)})\right)^{k}$, have the same size up to a constant factor, since this yields the smallest running time possible. For Vertex Cover we have $\hat{T}(k) = O(1.29175^{k} + kn)$ [204]. Equating both terms and using a computer algebra system to solve it numerically, one obtains $\alpha \approx 0.052455$ and $T(k) = O(1.2832^{k}k + kn)$, a slight improvement over the previous exponential bound 1.29175^{k}. Note that the space bound for this choice of α is $O(1.2748^{k}k + n^{2})$, that is, clearly (!), the exponential demand in space grows slower than the running time demand.

One might argue that these improvements are of purely asymptotical (and, thus, purely theoretical) interest. Note, however, that the presented dynamic programming method shrinks the search tree by a larger amount, the bigger the original search tree and the smaller the (linear) problem kernel is. For instance, if the best known search tree for Vertex Cover only were of size 2^{k} then we could reduce the search tree size to approximately 1.89^{k}. By way of contrast, the problem kernel for Dominating Set on planar graphs (size $335k$, see Section 2.5) is too big in order to significantly improve the corresponding search tree (size 8^{k}, see Section 3.6). If we had a size $2k$ problem kernel for Dominating Set on planar graphs (which might be possible but is open, of course) the search tree size could be improved from 8^{k} to approximately 4.67^{k}. Hence, then practical relevance would be within reach.

4.3 Color-Coding and Hashing

Most graph problems studied in this work are special versions of the Subgraph Isomorphism problem:

> **Input:** Two graphs $G = (V, E)$ and $G' = (V', E')$.
> **Question:** Is G' isomorphic to a subgraph in G?

For instance, Clique asks for a subset U of vertices such that in the induced subgraph $G[U]$ each pair of vertices is connected by an edge; Independent Set asks for a subset U of vertices such that in the induced subgraph $G[U]$ there is no edge at all. Both problems are special instances of Subgraph Isomorphism.

Alon *et al.* [13] introduced a randomized method called *color.coding* that can be used to derive (randomized) fixed-parameter algorithms for several subgraph isomorphism problems. We study this technique through an example application to the *NP*-complete Longest Path problem:

> **Input:** A graph $G = (V, E)$ and a nonnegative integer k.
> **Question:** Is there a *simple* path in G containing $k-1$ edges and k vertices?

Note that the restriction to simple paths where no vertex appears more than once is crucial here—otherwise, computing the kth power of the adjacency matrix of G, one can easily find in polynomial time all pairs of vertices that are connected by a path of $k-1$ edges.

The central idea behind color-coding is that to find the desired vertex set U with $|U| = k$ one randomly colors the whole graph with k colors and "hopes" that all vertices in U will obtain different colors. If so, the task to find U is greatly simplified. Color-coding makes strong use of *dynamic programming*. Moreover, the derived randomized fixed-parameter algorithms can be transformed into deterministic fixed-parameter algorithms through the use of hashing techniques at the cost of increased running time. Color-coding obviously is not powerful enough to derive fixed-parameter algorithms for the $W[1]$-complete problems Clique and Independent Set.

In our subsequent presentation, we basically follow [13]. The key to solve Longest Path lies in the concept of *colorful* paths which simply means that each vertex of the path has another color. On the one hand, each colorful path clearly is simple. On the other hand, coloring the graph vertices uniformly at random with k colors, a simple path will consist of k different colors with probability $(k!)/k^k$. Using Stirling approximation, this probability is lower-bounded by e^{-k}. For the time being, let us assume that there is a colorful simple path of k vertices in G. The following lemma shows that it can be found quickly by dynamic programming.

Lemma 4.3.1. *Let $G = (V, E)$ and let $C :\to \{1, \ldots, k\}$ be a coloring. Then a colorful path of k vertices can be found (if it exists) in time $2^{O(k)} \cdot |E|$.*

Proof. In what follows, we describe an algorithm that finds all colorful paths of $k-1$ vertices starting at some vertex s. This is not really a restriction because to solve the general problem, we may just add some extra vertex s' to V, color it with the new color 0, and connect it with each of the remaining vertices of V by an edge.

To find the described paths, we use dynamic programming. Assume that for all $v \in V$ already all possible color sets of colorful paths between s and v consisting of i vertices have been found. For each v, there are at most $\binom{k}{i}$ of these sets. Let now F be such a color set belonging to v. We consider every F belonging to v and every edge $\{u, v\} \in E$: Add $C(u)$ to F if $C(u) \notin F$. In this way, we obtain all color sets belonging to paths of length $i+1$ and so on. Thus, G obtains a colorful path with respect to coloring C iff there exists a vertex $v \in V$ that has at least one color set that corresponds to a path of $k-1$ vertices.

The described algorithm performs $O(\sum_{i=1}^{k} i\binom{k}{i} \cdot |E|)$ steps. Herein, factor i refers to the test whether or not $C(u)$ already is part of F. Factor $\binom{k}{i}$ refers to the number of possible sets F and factor $|E|$ refers to the time needed to test whether or not $\{u, v\} \in E$. The whole expression is upper-bounded by $O(k2^k \cdot |E|)$. □

Observe that in the proof of Lemma 4.3.1 it was crucial that not the paths (i.e., all vertices on them) were stored but only their corresponding color sets were recorded. Thus, for a path of $i \leq k$ vertices at most $\binom{k}{i} = O(k^i)$ candidate colorings are possible. By way of contrast, there are $\binom{n}{i} = O(n^i)$ different vertex sets of size i. This difference exactly reflects the gap between fixed-parameter tractability ("combinatorial explosion $f(k)$") and fixed-parameter intractability ("combinatorial explosion n^k"). It is not hard to effectively construct a colorful path as described in Lemma 4.3.1.

Now, using standard techniques for randomized algorithms (see [197] for a survey), a randomized fixed-parameter algorithm for Longest Path follows.

Theorem 4.3.1. Longest Path *can be solved in expected running time* $2^{O(k)} \cdot |E|$.

Proof. According to the above remarks a simple path of $k-1$ vertices is colorful with probability at least e^{-k}. According to Lemma 4.3.1, such a colorful path can be found in time $2^{O(k)} \cdot |E|$; more precisely, this implies that all colorful path of $k-1$ vertices can be found.

We repeat the following $e^k = 2^{O(k)}$ times:

1. Randomly choose a coloring $C : V \to \{1, \ldots, k\}$.
2. Check using Lemma 4.3.1 whether or not there is a colorful path; if so then this is a simple path of $k-1$ vertices.

□

Theorem 4.3.1 is based on a randomized algorithm. Using *hashing,* it can be de-randomized at the cost of some loss of efficiency. To this end, we need a list of colorings of the vertices in V such that for *each* subset $V' \subseteq V$ with $|V'| = k$ there is at least one coloring in the list that gives to each vertex in V' a unique color. This is formalized by the concept of a k-perfect family of hash functions from $\{1, 2, \ldots, |V|\}$ onto $\{1, 2, \ldots, k\}$.

Definition 4.3.1. *A* k-perfect family of hash functions *is a family* $\mathcal{H}$ *of functions from* $\{1, \ldots, n\}$ *onto* $\{1, \ldots, k\}$ *such that for each* $S \subset \{1, \ldots, n\}$ *with* $|S| = k$ *there exists an* $h \in \mathcal{H}$ *such that* h *is bijective when restricted to* S*.*

According to [13] and the literature cited there, the following holds.

Theorem 4.3.2. *Families of* k*-perfect hash functions from* $\{1, \ldots, n\}$ *onto* $\{1, \ldots, k\}$ *can be constructed which consist of* $2^{O(k)} \log n$ *hash functions. For such a hash function* h *the value* $h(i)$*,* $1 \leq i \leq n$*, can be computed in linear time.* □

In this way, we obtain deterministic fixed-parameter tractability for Longest Path.

Theorem 4.3.3. Longest Path *can be solved deterministically in time* $2^{O(k)} \cdot |E| \cdot \log |V|$.

Proof. Color the graph using all possible hash functions from the family given in Theorem 4.3.2. According to Definition 4.3.1 at least one of these colorings must lead to colorful, simple path. Such a colorful path then can again be found using Lemma 4.3.1.

Because the family from Theorem 4.3.2 consists of $2^{O(k)} \log n$ hash functions, the time complexity of the algorithm from Lemma 4.3.1 has to be multiplied with this factor. In total, we obtain the overall running time

$$2^{O(k)} \log |V| \cdot 2^{O(k)} |E| = 2^{O(k)} |E| \log |V|.$$

□

Although (randomized) color-coding appears as an elegant and prospective tool for designing fixed-parameter algorithms, we are not aware of any implementations and experimental results. Nevertheless, there are several other applications of color-coding to subgraph isomorphism problems, see Alon *et al.* [13] for details. Still, let us briefly discuss why $W[1]$-hard problems such as Clique or Independent Set seem inaccessible to color-coding. Consider Clique. In Lemma 4.3.1, it was decisive that a path could be represented by its start vertex s, its end vertex v, and the color set corresponding to the path. To extend a path by one further vertex, it was sufficient to consider edges with endpoint v and to know the already used colors. By way of contrast,

this would not be sufficient when constructing a clique in such a step-wise fashion. Here, we would need to check the existence of edges of the new vertex to all already selected vertices—the mere information about the colors of these vertices would not suffice. Then, however, we get the "$\binom{n}{i}$-behavior" instead of the "$\binom{k}{i}$-behavior" as discussed before.

We only mention in passing that families of k-perfect hash functions also can be directly used to obtain fixed-parameter algorithms by—similar to the above described de-randomization— systematically going through a search space testing all hash functions. Refer to Downey and Fellows [88] for some examples. To our knowledge, these approaches suffer from bad running times and are far from practical applications.

We conclude with a few words about a potential application of subgraph isomorphism and color-coding in the context of computational molecular biology. The point here is that searching for, e.g., "motifs" in RNA structures can be formulated as subgraph isomorphism problems where the subgraph is the motif and the supergraph models the structure of an RNA sequence. We omit any details here. It is completely open, however, to investigate whether this approach can attain practical usefulness (perhaps making use of special graph structures or classes) and, so far, no closer considerations of this issue have been undertaken.

4.4 Tree Decompositions of Graphs

How tree-like is a given graph? This question stands at the cradle of the concept of tree decompositions of graphs, introduced by Robertson and Seymour about twenty years ago [230]. The motivation behind is that many problems that turn out to be hard on general graphs are comparatively easy to solve on trees, a very simple class of graphs. For instance, it is easy to solve Vertex Cover or Dominating Set in linear time when restricted to trees—start at the leaves and work towards the root vertex. Hence, it is tempting to find out how far one can go by, on the one hand, dealing with a more general class of graphs than trees are, and, on the other hand, to preserve as much as possible of the properties that make trees so nice for several, generally hard algorithmic problems. Hence, in this sense, the notions of *tree decomposition* and the related *treewidth* measure were introduced in order to find a kind of compromise between the generality of graphs and the algorithmic feasibility of trees. In summary, tree decompositions of small width demonstrate algorithmic feasibility (and, in our sense, often fixed-parameter tractability) for many problems on graphs that are "almost" trees. Thorup [257] gave an impressive example for the meaningfulness of the treewidth concept in an application to compiler optimization. He showed that structured (which means goto-free) imperative programs have treewidth at most 6. More precisely, this means that the so-called interference graph of the control-flow graph of

the problem has treewidth at most 6. The corresponding tree decomposition then is used to (approximately) solve the register allocation problem (i.e., a coloring problem) for the given program. Refer to [257] for any details.

To find out whether or not a given graph has treewidth k turned out to be one of the cornerstones of fixed-parameter tractability. After a series of preceding work, Bodlaender [38] showed that this in general *NP*-complete problem [17] is fixed-parameter tractable—for constant k he gave a linear time algorithm whose constant factor exponentially depends on k.

In what follows, we will formally introduce the relevant notions and we will show how tree decompositions can be constructed and used to design—in a very general way—fixed-parameter algorithms for several *NP*-complete problems on planar graphs.

4.4.1 Definitions and Preliminaries

Definition 4.4.1. *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 $\{u, v\} \subseteq X_i$;*
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.*

Importantly, an $\ell \times \ell$-grid graph has treewidth ℓ. This can be easily shown by going—in a row-by-row manner—through the grid, always taking one more vertex from the next row and deleting one from the previous row when constructing the bags of the tree decomposition. We omit the details here. The important consequence of this, however, is that even planar graphs in general cannot be expected to have small treewidth. We will come back to that later on when discussing treewidth in the context of parameterized problems on planar graphs.

A tree decomposition with a particularly simple structure is given by the following. Its usefulness will be exhibited when solving problems by dynamic programming on tree decompositions, as will be shown in Section 4.5.

Definition 4.4.2. *A tree decomposition $\langle \{X_i \mid i \in I\}, T\rangle$ is called a* nice tree decomposition *if the following conditions are satisfied:*

1. *Every node of the tree T has at most two children.*
2. *If a node i has two children j and k, then $X_i = X_j = X_k$ (in this case, i is called a* JOIN NODE*).*
3. *If a node i has one child j, then one of the following situations must hold*

a) $|X_i| = |X_j| + 1$ *and* $X_j \subset X_i$ *(in this case,* i *is called an* INTRODUCE NODE*), or*
b) $|X_i| = |X_j| - 1$ *and* $X_i \subset X_j$ *(in this case,* i *is called a* FORGET NODE*).*

It is not hard to transform a given tree decomposition into a nice tree decomposition. More precisely, the following result holds (see [167, Lemma 13.1.3]).

Lemma 4.4.1. *Given a width* k *and* n *nodes tree decomposition of a graph* G*, one can find a width* k *and* $O(n)$ *nodes nice tree decomposition of* G *in linear time.* □

There are several equivalent definitions for graphs of treewidth k—the best known of these is that of *partial* k*-trees*. (See, e.g., [40] for more on that.) In addition, besides the central notion of treewidth there are other concepts such as pathwidth, branchwidth, and cliquewidth that also have been introduced to get algorithmic feasibility for otherwise hard graph problems. Here, we concentrate on treewidth and refer to [40, 44, 167] for more information on other notions.

4.4.2 Tree Decomposition and Graph Separation

Tree decompositions also have close connections to *graph separators*, that is, vertex sets whose removal from the given graph does separate the graph into two or more connected components. Actually, each bag of a tree decomposition forms a separator of the corresponding graph. Here, however, we are more interested in the reverse direction, i.e., constructing tree decompositions from graph separators. The main idea is to find small separators of the graph and to merge the tree decompositions of the resulting subgraphs.

Definition 4.4.3. *Let* $G = (V, E)$ *be a graph. A subset* $S \subseteq V$ *is called a* separator *of* G *if the subgraph* $G[V - S]$ *is disconnected.*

For any given separator splitting a graph into different components, we obtain a simple upper bound for the treewidth of this graph which depends on the size of the separator and the treewidth of the resulting components.

Proposition 4.4.1. *If a connected graph can be decomposed into components of treewidth of at most* t *by means of a separator of size* s *then the whole graph has treewidth of at most* $t + s$*.*

Proof. The separator splits the graph into different components. Suppose that we are given the tree decompositions of these components of width at most t. The goal is to construct a tree decomposition for the original graph. This can be achieved by firstly merging the separator to every bag in each of these given tree decompositions. In a second step, add some arbitrary connections preserving acyclicity between the trees corresponding to the components. It is straightforward to check that this forms a tree decomposition of the whole graph of width at most $t + s$. □

4.4.3 Graph Separators and Parameterized Problems on Planar Graphs

In case of planar graphs, there is a constructive way towards small separators. This is partially based on the "layer view" of planar graphs, expressed by the notion of r-outerplanarity.

Definition 4.4.4. *A plane embedding of a graph G is called* outerplanar *if each vertex lies on the boundary of the outer face. A graph G is called* outerplanar *if it admits an outerplanar embedding in the plane.*

The following generalization of the notion of outerplanarity was introduced by Baker [24].

Definition 4.4.5. *1. A plane embedding of a graph G is called* r-outerplanar *if, for $r = 1$, the embedding is outerplanar, and, for $r > 1$, inductively, when removing all vertices on the boundary of the outer face and their incident edges the embedding of the remaining subgraph is $(r-1)$-outerplanar.*

2. A graph G is called r-outerplanar *if it admits an r-outerplanar embedding.*

3. The smallest number r such that G is r-outerplanar is called the outerplanarity *number.*

In this way, we may speak of the layers $L_1, \ldots, L_r$ of an embedding of an r-outerplanar graph.

Definition 4.4.6. *For a given r-outerplanar embedding of a graph $G = (V, E)$, we define the* ith layer L_i *inductively as follows. Layer L_1 consists of the vertices on the boundary of the outer face, and, for $i > 1$, the layer L_i is the set of vertices that lie on the boundary of the outer face in the embedding of the subgraph $G - (L_1 \cup \ldots \cup L_{i-1})$.*

For plane graphs, i.e., planar graphs with a fixed embedding in the plane, there is an iterated version of Proposition 4.4.1.

Proposition 4.4.2. *Let G be a plane graph with layers L_i, $(i = 1, \ldots, r)$. For $i = 1, \ldots, \ell$, let $\mathcal{L}_i$ be a set of consecutive layers, i.e.,*

$$\mathcal{L}_i = \{L_{j_i}, L_{j_i+1}, \ldots, L_{j_i+n_i}\},$$

such that $\mathcal{L}_i \cap \mathcal{L}_{i'} = \emptyset$ for all $i \neq i'$. Moreover, suppose that G can be decomposed into components, each of treewidth of at most t, by means of separators $S_1, \ldots, S_\ell$, where $S_i \subseteq \bigcup_{L \in \mathcal{L}_i} L$ for all $i = 1, \ldots, \ell$.

Then, G has treewidth of at most $t + 2s$, where $s = \max_{i=1,\ldots,\ell} |S_i|$.

Proof. The proof again uses the merging-technique illustrated in Proposition 4.4.1: Suppose that, w.l.o.g., the sets $\mathcal{L}_i$ appear in successive order, i.e., $j_i < j_{i+1}$. For each $i = 0, \ldots, \ell$, consider the component G_i of treewidth at most t which is cut out by the separators S_i and S_{i+1} (by default, we set $S_0 = S_{\ell+1} = \emptyset$). We add S_i and S_{i+1} to every node in a given tree decomposition of G_i. In order to obtain a tree decomposition of G, we successively add an arbitrary connection between the trees T_i and T_{i+1} of the so-modified tree decompositions that correspond to the subgraphs G_i and G_{i+1}. □

Hence, for plane graphs the goal then can be set as follows: Decompose the given graph into various "small" components by using "small" separators, construct a tree decomposition for each component, and get an overall tree decomposition by applying Proposition 4.4.2. It remains to be discussed how to find these small separators and how to get the tree decompositions for the graph components to be generated. This will be done next.

One easily observes the following central relation between the domination number and the outerplanarity number of a planar graph.

Proposition 4.4.3. *If a planar graph $G = (V, E)$ has a k-dominating set then all plane embeddings of G can be at most $3k$-outerplanar.*

Proof. For a given crossing-free embedding of G in the plane, each vertex in the dominating set can dominate vertices from the previous, the next, or its own layer only. Hence, each vertex in the dominating set can contribute to at most three new layers. □

To understand the techniques used in the following, it is helpful to consider the concept of a *layer decomposition* of an r-outerplanar embedding of graph G. A layer decomposition of an r-outerplanar embedding of graph G is a forest of height $r - 1$. The nodes of the forest correspond to different connected components of the subgraphs of G induced by a layer. One easily observes that the planarity of G implies that the layer decomposition must indeed be a forest.

One more result needed is the following relation between r-outerplanarity and tree treewidth stated in [175, Table 2, page 550] and in [40, Theorem 83]. A constructive proof can be found in [2, 3].

Theorem 4.4.1. *An r-outerplanar graph has treewidth of at most $3r-1$.* □

Proposition 4.4.3 and Theorem 4.4.1 immediately imply the following relationship between the domination number and the treewidth of a planar graph.

Corollary 4.4.1. *If a planar graph has a k-dominating set then its treewidth is bounded by $9k - 1$.* □

Subsequently, we will give an asymptotically stronger bound.

The basic idea of constructing a tree decomposition of small width is the following. If the given graph has few layers, then directly use Theorem 4.4.1. If not, find small graph separators that decompose the graph into chunks of small outerplanarity, apply Theorem 4.4.1 to these graph chunks, and finally combine the tree decompositions of the various chunks into a big one for the overall graph using Proposition 4.4.2. Next, we describe how to find these small graph separators if needed.

It is clear that considering the layer decomposition $(L_1, \ldots, L_r)$ of a given planar graph of outerplanarity r, each layer L_i, $1 \leq i \leq r$, forms a separator. What makes the problem mathematically demanding is that the sizes of the layers L_i might be too large. Thus, it remains to be shown that, nevertheless, small separators can be found in a layerwise fashion. To this end, one makes use of the special properties of the underlying parameterized graph problem. Here, we focus on Vertex Cover and Dominating Set to see how this works. Both problems possess linear size problem kernels, that is, Vertex Cover has a kernel of size $2k$ (see Section 2.4) and Dominating Set on planar graphs has a kernel of size $355k$ (see Section 2.5). Hence, in both cases we trivially have that $|\bigcup_{i=1}^{r} L_i| = O(k)$.

Theorem 4.4.2. *A planar graph with a vertex cover or a dominating set of size k has treewidth $O(\sqrt{k})$.*

Proof. Using the graph layers L_i as separators, go through the sequence of layers $L_1, L_2, L_3, \ldots$ and look for separators of size $s(k) := O(\sqrt{k})$. Due to $|\bigcup_{i=1}^{r} L_i| = O(k)$ such separators of size at most $s(k)$ must appear within each $n(k) := O(\sqrt{k})$ sets in the sequence. In this manner, we obtain a set of disjoint separators of size at most $s(k)$ each, such that any two consecutive separators from this set are at most $O(\sqrt{k})$ layers apart. Clearly, the separators chosen in this way fulfill the requirements in Proposition 4.4.2.

Observe that the components cut out in this way each have $O(\sqrt{k})$ layers and, hence, their treewidth is bounded by $O(\sqrt{k})$ due to Theorem 4.4.1.

Using Proposition 4.4.2, we can upperbound the treewidth of the originally given graph by $O(\sqrt{k})$. □

Observe that the tree structure of the tree decomposition obtained in the preceding proof corresponds to the structure of the layer decomposition forest.

Remark 4.4.1. Up to constant factors, the relation exhibited in Theorem 4.4.2 is optimal. This can be seen, for example, by considering a grid graph G_ℓ of size $\ell \times \ell$, i.e., with ℓ^2 vertices and $2(l^2 - l)$ edges. It is known that the treewidth of G_ℓ is exactly ℓ (see [40, Corollary 89]) and that a minimum vertex cover as well as a minimum dominating set for G_ℓ both consist of $\Theta(l^2)$ vertices, see [149, Theorem 2.39].

Finally, we remark that a mathematically more refined analysis (not making use of linear size problem kernels but employing a more direct way of constructing the separators layerwisely) delivers that a planar graph with a vertex cover of size k has treewidth at most $4\sqrt{3k} + 5$ [7] and that a planar graph with a dominating set of size k has treewidth at most $6\sqrt{34k} + 8$ [3] (refer to [2] for a comprehensive, unified exposition). The constant factors in these upper bounds, however, seem to be rather worst-case and could possibly be improved.

4.4.4 Construction of a Tree Decomposition

We join together the preceding considerations into an algorithm that constructs tree decompositions of width $O(\sqrt{k})$ in case that we are given a planar graph that possesses a vertex cover or a dominating set of size at most k. It is important to note here that the tree decompositions are only constructed for the reduced graphs that are obtained by the reduction to problem kernel for the underlying parameterized problem (Vertex Cover or Dominating Set in this context). The algorithm proceeds in the following steps.

1. Perform a reduction to problem kernel that yields a reduced planar graph whose number of vertices is $O(k)$.
2. Embed the reduced planar graph $G = (V, E)$ crossing-free into the plane. Determine the outerplanarity number r of this embedding and all layers $L_1, \ldots, L_r$. By default, we set $L_i = \emptyset$ for all $i < 0$ and $i > r$.
3. Vertex Cover: If $r > k$, then exit (there exists no size k vertex cover). This is justified by the fact that each layer contains at least one edge which must be covered by a vertex from this layer.
   Dominating Set: If $r > 3k$ then exit (there exists no size k dominating set). This is justified by Proposition 4.4.3.
4. Find separators of size $O(\sqrt{k})$ according to the proof of Theorem 4.4.2.
5. Decompose the graph into subgraphs by removing all the graph separators found in the preceding step. Note that each of these subgraphs has outerplanarity $O(\sqrt{k})$.
6. Construct tree decompositions for all the subgraphs using Theorem 4.4.1. In this way, all subgraphs obtain tree decompositions of width $O(\sqrt{k})$.
7. Merge the tree decompositions of all subgraphs into a tree decomposition of the overall graph. To do so, use the tree decompositions of the subgraphs and the separators that generated these subgraphs (see fifth step above) and apply the "separator merging technique" described in the proof of Proposition 4.4.2.

The above algorithm outline constructively shows how to obtain tree decompositions of width $O(\sqrt{k})$ for parameterized problems such as Vertex Cover or Dominating Set on planar graphs. Clearly, to demonstrate the ultimate use of this approach it remains to be shown how the underlying graph problem can be efficiently solved using dynamic programming on tree

decompositions. This will be the topic of Section 4.2. Another point is the question how the sketched results can be generalized to more problems as well as to more general graph classes than planar graphs are. We discuss these issues now.

4.4.5 Refinements and Generalizations

In our presentation above on the construction of tree decompositions of small width we ignored at least the following three points.

1. What can we do if the given graphs are not planar?
2. What about the constant factors hidden in the O-notation used throughout the previous considerations?
3. To what kind of parameterized graph problems does the whole scenario apply to?

Concerning the first question referring to non-planar graphs, the following is of interest. Firstly, very recently, Demaine *et al.* [81] showed how to apply the methodology as sketched for planar graphs to the more general class of graphs of bounded genus. Whether the approach then remains practical still needs to be clarified. Of course, however, this is still far away from the case of general graphs. Bodlaender's [38] algorithm for tree decomposition construction is considered as impractical due to large constant factors involved. Still, there might be hope to improve the running time in future work. For the time being, however, it is adequate to take approximative and heuristic solutions into consideration. There is quite some ongoing work in this direction (cf., e.g., [168]). Also, there is a relatively easy and efficient factor-3 approximation for constructing tree decompositions—the obtained treewidth is at most by a factor 3 from the optimum value (see [225]).[3] It has to be studied empirically how far these approaches can reach and what their limitations in practice are.

As to the second question concerning the constant factors involved, firstly one should note that the given presentation traded comprehensibility for exact determination of the considered running time and treewidth values. Such a mathematical analysis can be found in [2, 3, 7]. As already indicated, the proven upper bounds involve quite high constant factors. For instance, based on the above approach and several more technical details the running times $O(2^{4\sqrt{3k}} \cdot n)$ and $O(2^{12\sqrt{34k}} \cdot n)$ can be proven for Vertex Cover and Dominating Set on planar graphs, respectively. The bounds are worst-case, however. On the one hand, there seems some room for lowering the bounds (cf. [115, 160] for recent new results in this direction) by refined mathematical analysis and, on the other hand, more importantly, there may be be further algorithmic improvements. And, perhaps, even most importantly, a

[3] Observe, however, that the approximation algorithm is a fixed-parameter algorithm in the sense that its running time is exponential in the treewidth.

theoretical worst-case bound may say little about the practical behavior of an algorithm. Implementation and extensive experimentation with "real-world" graph instances will bring the "final truth." Initial results in this direction, however, appear to be promising [2, 4].

As to the third question, it is only natural to ask in how far the problems Vertex Cover and Dominating Set are special or in how far the described approach can even be lifted to whole classes of problems on (planar) graphs. An at least partial answer to this question is given by means of the so-called *Layerwise Separation Property* as introduced in [7] (also see [2]). We only briefly discuss the fundamental concept here and refer to the given references for any further details. The basic idea is to exploit the layer-structure of a plane graph in order to gain a "nice" separation of the graph. It is important then that a "yes"-instance (G, k) (where G is a plane graph) of the considered graph problem $\mathcal{G}$ admits a so-called "layerwise separation" of small size. By this, one means, roughly speaking, a separation of the plane graph G (i.e., a collection of separators for G) such that each separator is contained in the union of constantly many subsequent layers. For (fixed-parameter) algorithmic purposes, it will be important that the corresponding separators are "small." More precisely, in order to generalize the methodology as presented in Subsection 4.4.3, the goal is to choose a set of separators such that the size of each of these is bounded by $O(\sqrt{k})$ and—at the same time—the subgraphs into which these separators cut the original graph have outerplanarity bounded by $O(\sqrt{k})$. In this way, in the same style as described in the proof of Theorem 4.4.2, tree decompositions of small width in a manner analogous to Subsection 4.4.4 can be constructed. Refer to [2, 7] for any details. Let us only mention in passing here that the Layerwise Separation Property in particular (and more or less trivially) holds for all problems on planar graphs (such as Vertex Cover and Dominating Set) that possess a linear size problem kernel—one more point that underpins the importance of good kernelization algorithms.

4.4.6 Final Remarks

Summarizing, the notions of tree decompositions and treewidth lead to mathematically elegant and clean ways to cope with the computational intractability of graph problems in some cases. Clearly, graphs of bounded treewidth may be considered as one form of a special graph class and there are numerous others [44]. Opposite to many other special graph classes, the task to determine for a given graph (maybe together with a given graph problem to solve) its treewidth has turned into an already "classical" fixed-parameter complexity question. Of course, there are other graph parameters such as pathwidth or branchwidth deserving similar attention as treewidth received (see [115] for a recent example concerning branchwidth). It continues to remain an important research topic to identify as many graphs (and graph problems) with practical relevance where small treewidths occur.

Concerning the practical feasibility of tree decompositions, besides bounding their widths as such it is of same importance to efficiently make use of them. The standard approach to this is dynamic programming as will be explained in the subsequent section. Notably, to get this dynamic programming as efficient and practical as possible is not only a question of running time but also of memory consumption. Finally, observe that these dynamic programming approaches provide optimal solutions to the considered problems. This gives rise to the importance of also approximative (c.f., e.g., [225]) or heuristic (c.f., e.g., [168]) algorithms for constructing tree decompositions as long as the treewidths remain small enough.

4.5 Dynamic Programming on Tree Decompositions

In Subsection 4.4, we became familiar with the concept of treewidth and, in particular, we saw how to construct a tree decomposition of "small" width for parameterized problems on planar graphs. Now, it is time to see how to make use of the "treelikeness" of graphs, that is, to see how the underlying graph problems can be solved by dynamic programming on tree decompositions. Typically, tree decomposition based algorithms proceed according to the following scheme in two stages (cf. [39]):

1. Find a tree decomposition of bounded width for the input graph, and
2. solve the problem by dynamic programming on the tree decomposition.

4.5.1 Solution for Vertex Cover

Dynamic programming is comparatively easy to explain in case of Vertex Cover—the algorithmic details get more involved in case of Dominating Set. This is due to the fact that the "combinatorics" behind Dominating Set is more elusive than the one behind Vertex Cover. Nevertheless, the basic ideas are already exhibited when dealing with Vertex Cover.

Theorem 4.5.1. *For a graph G with given tree decomposition $\langle\{X_i \mid i \in I\}, T\rangle$ an optimal vertex cover can be computed in time $O(2^\omega \cdot |I|)$. Here, ω denotes the width of the tree decomposition.*

Proof. The basic idea is to check for all of the $|I|$ many bags all of at most $2^{|X_i|}$ possibilities to obtain a vertex cover for the subgraph $G[X_i]$ of G induced by the vertices from X_i. This information is stored in tables A_i ($i \in V_T$). In a second step, these tables are compared against each other. Each bag of the tree decomposition thus has a table associated with it. The comparison process works in a bottom-up fashion from the leaves to the root of the tree decomposition, comparing "neighboring" tables (whose corresponding tree nodes are connected by an edge) against each other and updating the current

information. 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.

The algorithmic details are as follows.

Step 0: For each $X_i = \{x_{i_1}, \ldots, x_{i_{n_i}}\}$, $|X_i| = n_i$, compute a table

$$A_i = \begin{array}{ccccc||c} x_{i_1} & x_{i_2} & \cdots & x_{i_{n_i-1}} & x_{i_{n_i}} & m \\ \hline 0 & 0 & \cdots & 0 & 0 & \\ 0 & 0 & \cdots & 0 & 1 & \\ & & \vdots & & & \\ 1 & 1 & \ldots & 1 & 0 & \\ 1 & 1 & \ldots & 1 & 1 & \end{array} \Bigg\} 2^{n_i}$$

The table consists of 2^{n_i} rows and $|n_i| + 1$ columns. Each row represents a so-called "coloring" of subgraph $G[X_i]$. By this we mean a 0-1 sequence of length n_i that determines which of the respective bag vertices from X_i should be put into the current vertex cover ($\widehat{=}$ "1") and which should be not ($\widehat{=}$ "0"). Formally, a coloring is a mapping

$$C_i : X_i = \{x_{i_1}, \ldots, x_{i_{n_i}}\} \to \{0, 1\}.$$

For each of the 2^{n_i} different possibilities for a coloring, the table has one further entry. This last column stores (for each coloring C_i) the number $m_i(C_i)$ of vertices that a minimal vertex cover that contains those vertices from X_i selected by the coloring C_i would need. More precisely, that means that it contains the value

$$\begin{aligned} m_i(C_i) = \min\{|V'| : {} & V' \subseteq V \text{ is a vertex cover} \\ & \text{for } G, \text{ such that } v \in V' \text{ for all } v \in (C_i)^{-1}(1) \\ & \text{and } v \notin V' \text{ for all } v \in (C_i)^{-1}(0)\}. \end{aligned}$$

This value is determined by dynamic programming as described in Step 2 to follow.

Of course, not every possible coloring may lead to a vertex cover. Such a coloring is called *invalid.* To check whether or not a coloring C_i is *valid,* for each $\{u, v\}$ of the subgraph $G[X_i]$ induced by X_i, consider $C_i(u)$ and $C_i(v)$. If there is at least one edge where $C_i(u) = C_i(v) = 0$ then the coloring is invalid; otherwise, it is valid.

Step 1: Table initialization.

For all tables X_i and each coloring $C_i : X_i \to \{0, 1\}$ set

$$m_i(C_i) := \begin{cases} \left|(C_i)^{-1}(1)\right|, & \text{if } C_i \text{ is valid} \\ +\infty, & \text{otherwise} \end{cases}$$

Step 2: Dynamic Programming.

As mentioned before, we now go through the decomposition tree T from the leaves to the root and compare the corresponding tables against each other.

Let $j \in I$ be the parent node of $i \in I$. We show how the table X_j can be updated by the one for X_i. To this end, assume that

$$X_i = \{z_1, \ldots, z_s, u_1, \ldots, u_{t_i}\} \text{and}$$
$$X_j = \{z_1, \ldots, z_s, v_1, \ldots, v_{t_j}\},$$

that is, $X_i \cap X_j = \{z_1, \ldots, z_s\}$.

For each possible coloring

$$C : \{z_1, \ldots, z_s\} \to \{0, 1\}$$

and each extension[4] $C_j : X_j \to \{0, 1\}$ we set

$$\begin{aligned} m_j(C_j) \leftarrow\ & m_j(C_j) \\ & + \min\{\, m_i(C_i) \mid C_i : X_i \to \{0,1\} \text{ is an extension of } C\} \qquad (*) \\ & - \left|C^{-1}(1)\right| \end{aligned}$$

Additionally, we record at this point a coloring $C_i^* : X_i \to \{0, 1\}$ which led to a minimum value in (*). That is, we introduce a pointer from the row of coloring C_j in table X_j to the row of coloring C_i^* in table X_i. In this way, all entries of the last column of A_j are updated by those from A_i.

If a node $j \in V_T$ has several children $i_1, \ldots, i_l \in V_T$ then table A_j is successively updated against all tables $A_{i_1}, \ldots, A_{i_l}$ in the basically same way.

All this is repeated until the root node will finally be updated.

Step 3: Construction of an optimal vertex cover.

The size of an optimal vertex cover is derived from the minimum entry of the last column of the root node table A_r. The coloring of the corresponding row shows which of the vertices of the "root bag" X_r are contained in an optimal vertex cover. By taking down during Step 2 how the respective minimum of each bag was determined by its "children values," one can easily compute all vertices of an optimal vertex cover.

1. *Correctness of the algorithm.*

a) The first condition in the definition of a tree decomposition (see Definition 4.4.1), i.e., $V = \bigcup_{i \in I} X_i$, makes sure that each graph vertex is taken into account during the computation.

[4] By an extension of a coloring $C : W \to \{0, 1\}$ (where $W \subseteq V$) we mean a coloring $\tilde{C} : \tilde{W} \to \{0, 1\}$ with $\tilde{W} \supseteq W$ and $\tilde{C}|_W = C$.

b) The second condition in Definition 4.4.1, i.e., $\forall e \in E\ \exists i_0 \in I:\ e \in X_{i_0}$) makes sure that after the treatment of invalid colorings right after the initialization in Step 0, during the dynamic programming process only actual vertex covers are dealt with.

c) The third condition in Definition 4.4.1 guarantees the consistency of the dynamic programming. If a vertex $v \in V$ occurs in two different bags X_{i_1} and X_{i_2} then is made sure that for the computed optimal vertex cover this vertex does not receive different colors in the two respective rows in the tables A_{i_1} and A_{i_2}. Such a conflict would have been resolved in the bag of the least common ancestor i_0 of i_1 and i_2 in T. This is because of the third condition which guarantees that v also has to occur in X_{i_0}.

2. *Running time of the algorithm.*
Keeping the tables sorted in an adequate way, the comparison of a table A_j against a table A_i can be done in time

$$O(\#\text{rows of } A_i\ + \#\text{rows of } A_j) = O(2^{|X_i|} + 2^{|X_j|}) = O(2^\omega).$$

For each edge $e \in E_T$ in tree T such a comparison has to be done, that is, the overall running time of the algorithm is given by

$$O(2^\omega \cdot |E_T|) = O(2^\omega \cdot |I|).$$

□

Combining Theorem 4.5.1 with Theorem 4.4.2 and the corresponding algorithm that constructs a tree decomposition (see Subsection 4.4.4) results in a fixed-parameter algorithm for Vertex Cover on planar graphs that provides an exponential speedup compared to the best known fixed-parameter algorithms for Vertex Cover on general graphs (where we have running time $O(1.29^k + kn)$ [60, 204]).

Corollary 4.5.1. Vertex Cover *on planar graphs can be solved in time* $2^{O(\sqrt{k})}n$*, where* k *denotes the size of the vertex cover and* n *is the number of graph vertices.* □

Doing a more refined analysis, the exponential factor in Corollary 4.5.1 can be bounded by $2^{4\sqrt{3k}}$ (cf. [2, 7]).

4.5.2 A Glimpse on Dominating Set

The basic technique for solving Dominating Set on a "tree-decomposed" graph is the same as for Vertex Cover. We have already seen, however, that from a combinatorial point of view Dominating Set is a problem that is more elusive than Vertex Cover. This now also reflects in the larger overhead needed to solve Dominating Set in the dynamic programming way. The very first observation is that we need three colors for the dynamic

programming tables instead of only two as we had for VERTEX COVER: Suppose that $G = (V, E)$ and $V = \{x_1, \ldots, x_n\}$. Assume that the vertices in the bags are given in increasing order when used as indices of the dynamic programming tables, i.e., $X_i = (x_{i_1}, \ldots, x_{i_{n_i}})$ with $i_1 \leq \ldots \leq i_{n_i}$. We use three different "colors" that will be assigned to the vertices in the bag:

- "black" (represented by 1, meaning that the vertex belongs to the dominating set),
- "white" (represented by 0, meaning that the vertex is already dominated at the current stage of the algorithm), and
- "grey" (represented by $\hat{0}$, meaning that, at the current stage of the algorithm, one still asks for a domination of this vertex).

Again, mapping $C_i : \{x_{i_1}, \ldots, x_{i_{n_i}}\} \to \{0, \hat{0}, 1\}^{n_i}$ will be called a *coloring* for the bag $X_i = (x_{i_1}, \ldots, x_{i_{n_i}})$, and the *color* assigned to vertex x_{i_t} by C_i given by $C_i(x_{i_t})$.

For each bag X_i (with $|X_i| = n_i$), use a mapping

$$m_i : \{0, \hat{0}, 1\}^{n_i} \longrightarrow \mathbb{N} \cup \{+\infty\}.$$

For a coloring C_i, the value $m_i(C_i)$ stores how many vertices are needed for a minimum dominating set (of the graph visited up to the current stage of the algorithm) under the restriction that the color assigned to vertex x_{i_t} is $C_i(x_{i_t})$ $(t = 1, \ldots, n_i)$. Now, performing a table updating process analogous to VERTEX COVER case described before, it is not too hard to finally come up with a time $O(9^{O(\sqrt{k})}n)$ algorithm (also cf. [255, 256] concerning the dynamic programming) for DOMINATING SET on planar graphs, which parallels Corollary 4.5.1. We wrote $9^{O(\sqrt{k})}$ instead of the equivalent $2^{O(\sqrt{k})}$ in order to emphasize that the exponential factor is basically 9^{ω} for DOMINATING SET and 2^{ω} for VERTEX COVER, where ω denotes the width of the given tree decomposition. The significant increase from base value 2 to 9 is due to to the more complicated "dependence structure" in the combinatorics of DOMINATING SET when implemented in a basically straightforward way. The base 9 can be lowered to 4 by making use of a kind of monotonicity property that holds for the colors $\hat{0}$ and 0: On the color set $\{0, \hat{0}, 1\}$, let $\prec$ be the partial ordering given by $\hat{0} \prec 0$ and $d \prec d$ for all $d \in \{0, \hat{0}, 1\}$. This ordering naturally extends to colorings in a "component-wise" fashion, then using the notion $C \prec C'$.

It is essential for the improved dynamic programming that the mappings m_i are *monotonous* from $(\{0, \hat{0}, 1\}, \prec)$ to $(\mathbb{N} \cup \{+\infty\}, \leq)$, i.e., that for colorings $C : \{x_{i_1}, \ldots, x_{i_{n_i}}\} \to \{0, \hat{0}, 1\}$ and $C' : \{x_{i_1}, \ldots, x_{i_{n_i}}\} \to \{0, \hat{0}, 1\}$, $C \prec C'$ implies $m(c) \leq m(c')$. Roughly speaking, the use of this kind of monotonicity allows us to omit several (in fact, very many) comparisons between corresponding entries in two different tables during the bottom-up updating process of dynamic programming. We omit the technical details and refer to [2, 3, 12] for these. As a last point, we mention in passing that the

Algorithm	$\omega = 5$	$\omega = 10$	$\omega = 15$	$\omega = 20$
$9^\omega n$	0.05 sec	1 hour	6.5 years	$3.9 \cdot 10^5$ years
$4^\omega n$	0.001 sec	1 sec	18 minutes	13 days

Table 4.1. Comparing the $O(4^\omega n)$ algorithm for Dominating Set with the $O(9^\omega n)$ algorithm of Telle and Proskurowski in the case $n = 1000$ (number of nodes of the tree decomposition), we assume a machine executing 10^9 instructions per second and we neglect the constants hidden in the O-terms (which are comparable in both cases).

above sketched improved dynamic programming for Dominating Set also is essentially based on the use of *nice* tree decompositions as introduced in Subsection 4.4.1 (Definition 4.4.2). Nice tree decompositions significantly simplify the reasoning about the updating process in more complicated dynamic programming contexts as used for Dominating Set. We mention in passing that these kinds of improvements apply to a whole class of domination-like problems such as Perfect Code, Total Dominating Set, Independent Dominating Set, etc. (see [2, 3, 12]) for details).

4.5.3 Final Remarks

The most important point in dynamic programming on tree decompositions are the sizes of the tables involved. The table sizes usually are bounded by c^ω, where ω denotes the width of the underlying tree decomposition and c usually depends on the underlying combinatorial problem. Hence, two optimization goals are immediate:

1. Keep the width of the tree decomposition as small as possible (see Section 4.4), and
2. closely investigate the combinatorics of the underlying graph problem in order to keep the base c as small as possible.

Dominating Set provides a striking example for the second goal, as the constant could be improved from 9 to 4. To illustrate the significance of such a result, Table 4.1 compares (hypothetical) running times of the $O(9^\omega n)$ algorithm of Telle and Proskurowski [255, 256] to the $O(4^\omega n)$ monotonicity based algorithm for some realistic values of ω and $n = 1000$. Improving exponential terms always is a "big issue" for fixed-parameter algorithms.

It must be emphasized that besides (exponential) running time also (exponential) memory use is an important issue in making tree decomposition based algorithms useful in practice. Aspvall *et al.* [21] present some ways how to reduce the memory requirement of the table computations in dynamic programming as discussed above. In order to avoid the "memory boundedness" of dynamic programming on tree decompositions, all tricks and techniques should be tried—another promising future research challenge connected with the development of efficient fixed-parameter algorithms.

4.6 Concluding Discussion

Several promising techniques for the design of efficient fixed-parameter algorithms have been presented in this chapter. It is to be hoped that, on the one hand, these further develop into practically useful tools and, on the other hand, that new ways for showing fixed-parameter tractability results will emerge. Perhaps there is also more practical potential in methodologies such as monadic second order logic, graph minor theory, or tree automata (see [88] for some survey) which have been neglected here. Future research has to decide on this.

We conclude with summarizing some main problems, tasks, and challenges stirred up by the considerations in this chapter.

- Linking integer linear programming and fixed-parameter complexity in a stronger way is a highly desirable goal.
- We saw two completely different ways of dynamic programming, one corresponding to search trees and the other corresponding to tree decompositions. Extending described results and opening up new "fixed-parameter dynamic programming" methods is of particular interest when, on the one hand, thinking of the enormous importance of the dynamic programming paradigm in the algorithms applied in computational biology and, on the other hand, noticing the fruitful grounds for parameterization to be found there.
- Width parameters of graphs such as treewidth or branchwidth will continue to play an important role in the fixed-parameter context and many things remain to be investigated and improved here.
- The real practical usefulness of color-coding and hashing techniques still remains to be investigated.
- Most of the considered techniques still lack (sufficient) experimental validation of their merits in practice—algorithm engineering and making available benchmark test instances are major challenges here.
- As an ultimate goal one might consider to gain more insight about which techniques are doomed to remain pure "classification tools" and which techniques at least carry the potential to survive in practical applications.

From today's standpoint, however, it should be emphasized that the two most important techniques in achieving "practical fixed-parameter tractability" continue to be reduction to problem kernel (Chapter 2) and bounded search trees (Chapter 3).

5. Further Case Studies

The purpose of this chapter is to present some further concrete problems that have been amenable to fixed-parameter approaches. The selection of these problems is based on personal preference and is more or less restricted to own work. It is tried to highlight some useful, maybe generalizable observations in the work with fixed-parameter algorithms and the like.

A generally important thing when browsing through the meanwhile extensive literature is to always keep in mind that many publications present fixed-parameter tractability results "in disguise." That is, many published exact algorithms have not been explicitly termed "fixed-parameter algorithms" (although they clearly are) and they do not use any other terms of parameterized complexity. We come back to that point in Section 5.3.

This chapter focuses on two main fields where the fixed-parameter approach has been successfully applied—computational biology and graph resp. network problems. Whereas the second field is "classic" for parameterized complexity, computational biology is a newer, seemingly almost inexhaustible and very prospective area of fixed-parameter investigation. Here, we will further substantiate our observation that computationally hard problems arising in (molecular) biology often carry natural parameters that make them prime candidates for fixed-parameter complexity studies. We begin our exposition with this fast growing field. After that, we return to the already more familiar setting of graph and network problems—still a rich and by far not exhausted source of fixed-parameter algorithmic challenges. Finally, in Section 5.3, we briefly survey several other interesting application fields and also name some concrete challenges for future research on efficient fixed-parameter algorithms.

5.1 Computational Biology

Dealing with biologically motivated, algorithmic problems has become a vast area of research. Even topics such as the reconstruction of phylogenetic trees, the analysis of gene expression data (also closely related to clustering problems), the comparison and structure prediction of protein, RNA, and DNA molecules, or the search for motifs and signals in sequences (closely related to string searching problems) form subfields that are hard

to overview. Some surveys on computational biology are provided by the books [144, 155, 220, 238, 261]. Clearly, the considerations in this section form a very small selection of biologically motivated problems in the fixed-parameter context. Additionally, observe that some of the problems discussed, (e.g., problems on strings), may have applications also in other fields such as, for instance, information retrieval or coding theory.

Before we come to some more concrete examples of fixed-parameter analysis in the computational biology context, however, the following assessment shall be put forward. In coping with *NP*-hard problems, the main theoretical, also much better investigated "competitors" of fixed-parameter algorithms are approximation algorithms. In this respect, we quote one of the anonymous referees of the paper [135], whose statement is to support:

> "...fixed-parameter algorithms do seem laudable approaches to *NP*-hard problems in biology, better than approximation methods in most cases."

A second point that makes computational biology a particularly fruitful area for fixed-parameter studies is the fact that often there are several (and frequently all of them reasonable at the same time) parameters in a given problem and "usually" at least one of them can be considered as small. For instance, recall Closest String (with applications in primer design and motif search) from Sections 3.3 and 4.1. Here, two very natural parameters are the number of input strings k as well as the maximally allowed Hamming distance d to the closest string that is to be found. Both k as well as d in practice are small numbers (e.g., $k \approx 10$ and $d \approx 5$), hence parameterizations in both directions make sense and both actually lead to fixed-parameter algorithms (see Sections 3.3 and 4.1). Last but not least it goes without saying that computational biology offers a vast amount of *NP*-hard problems, thus triggering research for approximative or heuristic and now increasingly also fixed-parameter studies.

In the following concrete examples for fixed-parameter results in computational biology we mainly concentrate on the parameterization issues and the basic ideas of the corresponding fixed-parameter algorithms. For more involved technical details, we always refer to the cited literature. In one case, we will also encounter parameterized intractability ($W[1]$-hardness), see Subsection 5.1.3, and we discuss the arising algorithmic consequences.

5.1.1 Phylogenetic Trees: Minimum Quartet Inconsistency

We follow parts of [133]. To determine the evolutionary relationship of a set of taxa, e.g., based on DNA or protein sequence data, is an important question in computational biology. A common model for this relationship is an *evolutionary tree*, a binary tree T in which the leaves are bijectively labeled by the taxa. In recent years, quartet methods for reconstructing evolutionary

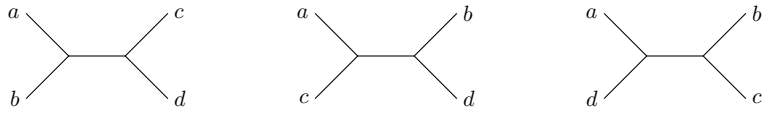

Fig. 5.1. Possible quartet topologies for quartet $\{a, b, c, d\}$, which are (from left to right) $[ab|cd]$, $[ac|bd]$, and $[ad|bc]$.

trees have received considerable attention [63, 156]. Here, a *quartet* is a size four subset $\{a, b, c, d\}$ of the set of taxa and the *quartet topology* for $\{a, b, c, d\}$ induced by T simply is the four leaves subtree of T for $\{a, b, c, d\}$. The three possible quartet topologies for $\{a, b, c, d\}$ are $[ab|cd]$, $[ac|bd]$, and $[ad|bc]$. They are shown in Fig. 5.1.[1] The fundamental goal of quartet methods is, given a set of quartet topologies, to reconstruct the corresponding evolutionary tree. Herein, the given set of quartet topologies can be incomplete, may contain errors or more than one topology for one quartet. Hence, to reconstruct (a good estimation of) the original evolutionary tree becomes an optimization problem.

We focus on the Minimum Quartet Inconsistency (MQI) problem.

> **Input:** A set S of n taxa and a set Q_S of $\binom{n}{4}$ quartet topologies such that there is exactly one topology for *every* quartet corresponding to S and a nonnegative integer k.
> **Question:** Is there an evolutionary tree T where the leaves are bijectively labeled by the elements from S such that the set of quartet topologies induced by T differs from Q_S in at most k quartet topologies?

MQI is *NP*-complete [35, 157]. It is worth noting, as was pointed out by Steel [246], that the quartet cleaning algorithm by Berry *et al.* [35] finds the optimal solution for instances with $k < (n-3)/2$. Therefore, MQI is *NP*-hard only for $k \geq (n-3)/2$. It is known that MQI is polynomial time approximable with a factor n^2 [156], and it is an open question of [156] whether MQI can be approximated with a factor at most n or even with a constant factor. Refer to [80] for some recent partial progress on approximating MQI. Heuristics for the problem include semidefinite programming [33] and the widely used *quartet puzzling* [253]. The parameterized complexity of MQI, however, so far, has apparently been neglected. For the case that the number k of "wrong" quartet topologies is small in comparison with the total number of given quartet topologies, MQI is *fixed-parameter tractable.* It can be solved exactly in worst-case time $O(4^k n + n^4)$. Observe that the input size is $O(n^4)$. The more general variant of MQI where the set Q_S is not required to contain a topology for every quartet is *NP*-complete even if $k = 0$ [245]. Hence, this excludes fixed-parameter complexity studies and also implies inapproximability.

[1] A fourth possible topology is the star topology which is not considered here because it is not binary.

There are several reasons why quartet methods are widely used in practice. They are founded on the fact that an evolutionary tree is uniquely characterized by the quartet topologies for its size four sets of taxa [47]. From this set of topologies, we can efficiently compute the tree in polynomial time $O(n^4)$ [28]. Quartet methods clearly divide the tree construction process in two stages—one can use an arbitrary, even computationally expensive tree construction method for inferring the quartet topologies, while the recombination of topologies can be handled independently of the method chosen for inference. Another reason to use quartet methods is data disparity as discussed by Chor [63]: In practice, one often does not have the same amount of data for all considered taxa, e.g., not the same set of sequenced proteins. In general, tree construction methods cannot take advantage of information available only for a subset of taxa. Quartet methods, however, allow to use the maximum amount of information available for the four taxa of a quartet to compute its topology. The limitation of quartet methods in practice is caused by the process of quartet inference which can be erroneous. Therefore, one cannot be sure that there exists a tree inducing the inferred set of quartet topologies. Assuming that the number of errors is small compared to the number of correct topologies, we overcome this problem by searching for a tree that matches the inferred topologies as "closely" as possible. Refer to [129, 133] for some survey on results with respect to MQI.

A Fixed-Parameter Algorithm for MQI

The key to develop a fixed-parameter solution for MQI is that it is sufficient to examine the size three sets of quartet topologies and to recursively branch on local conflicts. Roughly speaking, this means that the fixed-parameter algorithm solving MQI can process as follows: If the give set of quartet topologies is non-conflicting and if there is exactly one topology for each possible quartet then one can construct the corresponding evolutionary tree in time $O(n^4)$ [28]. Otherwise, as long as the given set of quartet topologies is conflicting then by results of Colonius and Schultze [67] and Bandelt and Dress [25] one can deduce that there must exist a subset of three quartet topologies whose set of taxa altogether contains five elements. Then, these three topologies contradict each other in the sense that there is no binary tree with five leaves labeled by the given taxa such that it induces these three topologies, see [129, 133] for details. In addition, one can efficiently maintain a conflict list containing all current "local" conflicts of the above kind. Thus, the idea for a bounded search tree algorithm becomes immediately clear. The only thing one still has to observe is that there are four ways to get rid of such a local conflict by changing one of the three given quartet topologies. This leads to a branching into four cases, each of which decreases the parameter k of maximally allowed quartet topology changes by one. In summary, one ends up with the following fixed-parameter result.

Theorem 5.1.1. *MQI can be solved in time* $O(4^k \cdot n + n^4)$. □

The algorithm behind Theorem 5.1.1 can be tremendously sped up in practice by adding several heuristic improvements that do not violate the optimality of the obtained solution. A simple addition is to guarantee that no quartet topology is changed more than once. Another one is the fact that if there is a topology t that is involved in more than $3k$ local conflicts (each consisting of three quartet topologies) then t *has* to be changed (so-called "forced change"). Finally, there are two more involved observations that may allow recognizing "hopeless situations" where, as a consequence, subtrees of the search tree can be discarded from further consideration [129, 133]. All this indicates that the performance of fixed-parameter algorithms often can be boosted significantly by adding heuristics (such as branch-and-bound) to further shrink the size of the search tree by possibly omitting some subcases from further consideration. For instance, experiments with synthetic data for MQI for $n = 50$ and $k = 100$ gave a reduction of the search tree size from 4^{100} to 46000 nodes [133, 134]. Moreover, results with real fungi data led to encouraging results with a positive biological interpretation.

A Glimpse on Future Work

From a parameterized point of view, it remains open to find an efficient reduction to problem kernel for MQI. As suggested by Chor [64], one might consider other parameters, e.g., asking whether we can satisfy $(m/3) + k$ quartets for m given quartet topologies. It might also help to identify parameters inherent to the problem. Since MQI can be solved in polynomial time for $k < (n-3)/2$ [35], we can ask—in the spirit of parameterizing above guaranteed values (cf. Subsection 1.5.2)—whether it is fixed-parameter tractable to find a tree that violates at most $(n-3)/2 + k$, $k \geq 0$, quartet topologies. The answer remains to be given.

5.1.2 Breakpoint Medians and Breakpoint Phylogenies

Gene orderings have grown into a popular measure to investigate the evolutionary relationship between species that share a common set of genes in their genome. More precisely, the relative order of the genes on the respective genome, pairwisely compared to each other, is used as a distance measure (so-called breakpoint distance) to construct phylogenetic trees whose leaves are labeled by the given species. Still, however, approaches in this direction hold some heuristic element due to the enormous number of combinatorial possibilities involved. Here, we describe a new approach to the central Breakpoint Median problem together with a new heuristic for constructing "breakpoint phylogenetic trees" based on the developed fixed-parameter algorithm for Breakpoint Median. We mainly follow [135].

Problem Definition

BREAKPOINT MEDIAN is defined as follows.

> **Input:** Signed orderings $\pi_1, \pi_2, \ldots, \pi_k$ on n elements and a non-negative integer d.
> **Question:** Is there a signed ordering π such that $\sum_{i=1}^{k} d_{bp}(\pi_i, \pi) \leq d$?

Herein, $d_{bp}(\pi_i, \pi)$ denotes the *breakpoint distance* (defined below) between orderings π_i and π. Given a set $G = \{1, \ldots, n\}$, an *ordering* π on G is a one-to-one function $\pi : G \to G$. We require that every ordering is extended by two special elements s, marking the start, and t, marking the end, and we write ordering π as $\langle\, s\ \pi(1)\ \pi(2)\ \ldots\ \pi(n)\ t\,\rangle$. We write G_s for $G \cup \{s\}$ (G_t and $G_{s,t}$, analogously). An ordering π is *signed* iff every $\pi(x)$, $x \in G$, is equipped with a sign $\{+, -\}$, denoting the "orientation" of the element, such that $\pi(x)$ can be, for $y \in G$, a "positive" element $+y$ (or, for sake of brevity, only y), having left-to-right orientation, or a "negative" element $-y$, having right-to-left orientation. Note that a signed ordering contains either y or $-y$ but not both at the same time. The special elements s and t are always unsigned. For $x \in G^{\pm}$ we define $succ_\pi(x) := y$ if we can find $l \in G$ such that one of the following two conditions applies:

1. $\pi(l) = x$ and $\pi(l+1) = y$, or
2. $\pi(l) = -x$ and $\pi(l-1) = -y$.

Note that this definition also includes $x < 0$. For the special cases that $x = s$ or that $y \in \{s, t\}$, we define $succ_\pi(s) := y$ if $\pi(1) = y$; for $x \in G^{\pm}$, $succ_\pi(x) := t$ if $\pi(n) = x$, and $succ_\pi(x) := s$ if $\pi(1) = -x$.

Given two signed orderings π_1 and π_2, both over G, we call a pair (x, y), $x \in G_s^{\pm}$ and $y \in G_t^{\pm}$, a *breakpoint* of π_1 with respect to π_2, if

1. $x = s$ or $\pi_1(l) = x$ for some $l \in G$, and
2. $succ_{\pi_1}(x) = y$ and $succ_{\pi_2}(x) \neq y$.

Using the notion of breakpoints, the *breakpoint distance* d_{bp} between two signed orderings is defined as follows:

$$d_{bp}(\pi_1, \pi_2) = \left|\{\, (x, y) \mid x, y \in G_{s,t}^{\pm}, x, y \text{ is breakpoint of } \pi_1 \text{ w.r.t. } \pi_2 \,\}\right|$$

Due to symmetry, $d_{bp}(\pi_1, \pi_2) = d_{bp}(\pi_2, \pi_1)$.

BREAKPOINT MEDIAN fits into the larger field of consensus analysis problems, which occur in many computational biology settings (also cf. [129]). Subject to different types of input strings (here we have orderings) and different kinds of distance measures (here we have breakpoint distance), various complexity results and algorithms were published in this context during the last years, e.g., [55, 106, 151, 179, 240]. Not surprisingly, in general BREAKPOINT MEDIAN is *NP*-complete, and remains so even in the case of only three input orderings [46, 218]. In the case of three input orderings, Pe'er

and Shamir [219] developed a polynomial-time algorithm with approximation factor 7/6. Keeping an eye on its application in the phylogenetic context, however, note that Moret *et al.* [195] emphasize that "because suboptimal solutions can yield very different evolutionary reconstructions, exact solutions are strongly preferred over approximate solutions." Hence, fixed-parameter algorithms are of concern.

Fixed-Parameter Algorithm for Breakpoint Median

The following intuitive lemma from [46] gives a way to simplify a given input instance by preprocessing:

Lemma 5.1.1. *Given signed orderings $\pi_1, \pi_2, \ldots, \pi_k$, all on a set G of n elements, and elements $x, y \in G^{\pm}_{s,t}$ which are adjacent in $\pi_1, \pi_2, \ldots, \pi_k$, i.e., $succ_{\pi_r}(x) = y$ for all $r = 1, \ldots, k$. Then x and y are also adjacent in an optimal breakpoint median π, i.e., $succ_{\pi}(x) = y$.* □

Using Lemma 5.1.1, we can preprocess the instance by "contracting" elements adjacent in all input sequences. This can be interpreted as a "reduction to problem kernel," where the original instance consisting of k orderings of n elements each is reduced to a new instance consisting of k orderings of at most d elements each (still, of course, all orderings have exactly the same number of elements). Therefore, we can assume that in the given set $\Pi = \{\pi_1, \pi_2, \ldots, \pi_k\}$, for every element x, there are at least two orderings in which x has different successors.

Surprisingly, an optimal breakpoint median can have adjacencies that are not present in any of the input orderings [46].

Lemma 5.1.2. *Given signed orderings $\pi_1, \pi_2, \ldots, \pi_k$, all on a set G of n elements, and an optimal breakpoint median π. Then there can be elements $x, y \in G^{\pm}_{s,t}$ with $succ_{\pi}(x) = y$ and $succ_{\pi_r}(x) \neq y$ for all $r = 1, \ldots, k$.* □

Here, we only sketch the basic idea behind the search tree algorithm and refer to [129, 135] for any details.

The algorithm starts its search for a median ordering π with the set of unconnected elements G, i.e., no element is assigned a successor or a predecessor. Then, the algorithm searches a median by introducing link by link into π until all elements in π are linked to a successor or predecessor (except for s and t, which only have a successor or a predecessor, respectively). Importantly, in its search for the median, the algorithm prefers links that are also present in the input orderings. However, due to Lemma 5.1.2, we also have to consider links that are not present in the input orderings. In this case, the algorithm defers the determination of a successor (or predecessor) and a special "nil" successor or predecessor is chosen. Thus, the recursive algorithm builds a search tree to construct π from initially unconnected elements; in

one node of the search tree, it selects an element $x \in G^{\pm}_{s,t}$ without successor or predecessor in π. It decides on a set of possible successor (or predecessor) values and recursively considers these values by branching into one subcase for each successor (or predecessor) value in the set. In this search, it keeps track of the number of induced breakpoints and stops the recursion if more than d breakpoints are introduced.

Altogether, the following result is obtained [129, 135].

Theorem 5.1.2. Breakpoint Median *for $k \geq 3$ can be solved in time $O((2.15)^d \cdot kn)$.* □

Notably, the worst-case branching vector of the recursion is of the form $(k-1, 1, k/2)$. Hence, the constant base $c = 2.15$ only occurs for $k = 3$ given orderings. For $k = 5$, e.g., we obtain $c = 1.68$ and for $k = 20$ we obtain $c = 1.21$. Observe that this decrease is "necessary" from an applied point of view because with increasing number k of given orderings also the parameter value d should grow—d "sums up" the breakpoint distances of the median ordering to all given orderings. It is open whether the problem remains fixed-parameter tractable in case of setting the maximum distance as the measure instead of the sum of distances. Also see [129, 135] for further discussion of the phenomenon of decreasing exponential bases and required "normalized" distance parameters as a challenge for future research.

Application to Breakpoint Phylogeny Reconstruction

In [129, 135], the following heuristic based on the fixed-parameter algorithm for Breakpoint Median was proposed and successfully applied to the "benchmark" data set of the plant family Campanulaceae [70, 71].

Given gene orderings $\Pi = \{\pi_1, \ldots, \pi_k\}$ for a set of k taxa, the algorithm starts by computing a root node, called *virtual root* of the tree (only necessary for the construction) and, then, the algorithm recursively divides the set of taxa into two subsets, associating new nodes with these subsets; the new nodes become child nodes of the virtual root and roots for the subtrees corresponding to the subsets. The recursion ends when the subsets have size exactly one.

To label the virtual root node, the heuristic computes the breakpoint median π_r for the given set of gene orderings. To obtain a bipartition of the set of taxa it considers all 2^{k-1} distinct bipartitions of Π into non-empty sets Π_1 and Π_2. It computes the optimal breakpoint medians π_1 for $\Pi_1 \cup \{\pi_r\}$, inducing a score of d_1 breakpoints, and π_2 for $\Pi_2 \cup \{\pi_r\}$, inducing a score of d_2 breakpoints. Among all these bipartitions, it chooses the ones with a minimum total number of induced breakpoints, i.e., the ones for which $d_1 + d_2$ is minimum. The breakpoint medians π_1 and π_2 corresponding to such an optimal bipartition are chosen to label the two child nodes of the node labeled π_r. Now, if Π_1 (Π_2 is completely analogous) consists of two elements

only then create two child nodes of the π_1 node, each child labeled with one element from Π_1. If Π_1 contains more than two elements then process this set recursively, taking the π_1 node as the virtual root and Π_1 as the set of gene orderings, again considering all bipartitions of Π_1.

In comparison to previous approaches that search through the whole space of all $\prod_{j=3}^{k}(2j-5)$ possible trees over k taxa [193, 194, 236], the search space of the new heuristic is determined by $k2^{k-1}$ considered bipartitions.

The whole scenario was applied to the Campanulaeceae data set and phylogenetic trees were found as good as those known in the literature in less than two minutes—significantly faster than previous approaches (see [129, 135]).

A Glimpse on Future Work

It remains open whether the closely related Breakpoint Center, where, by way of contrast to Breakpoint Median, not the sum of distances but the *maximum* distance shall be minimized, is also fixed-parameter tractable with respect to d.

In the multiple sequence alignment context, there also arises a median problem, but then with edit distance instead of breakpoint distance. It is interesting to ask whether this median problem is fixed-parameter tractable with respect to the corresponding distance parameter—an approach similar to the one employed here seems possible. Note, however, that trying to confine the combinatorial explosion to the number k of input species again seems fruitless, since it can be deduced from results of [41] and [151] that the problem is $W[t]$-hard for all t (see [129, 135] for more on that).

As to Breakpoint Median, future theoretical research might deal with the mentioned normalization effects for d. Also, it would be desirable to extend the algorithm to the case in which not all orderings are over the same set of elements or when elements occur more than once in one ordering; these cases apply when genomes have a different set of genes or contain duplicated genes. Further experiments could address the application of the presented breakpoint phylogeny heuristic to new biological datasets or to synthetic datasets. Also, with respect to the desired application to phylogeny reconstruction, it might be useful to extend the considerations and experiments to weighted variants of Breakpoint Median. Finally, it would be desirable to identify further applications of Breakpoint Median besides the breakpoint phylogeny application presented here.

5.1.3 Motif Search

Motif search problems are of central importance for sequence analysis in computational molecular biology. These problems have applications in fields such as genetic drug target identification or signal finding (see [45, 174, 178, 179,

221] and the references cited therein for more details and further applications). In Sections 3.3 and 4.1, we already introduced the CLOSEST STRING problem. It is directly applicable in motif search if the input consists of already aligned strings such that one can easily shift a "comparison window" throughout the given strings (cf. [129, 137, 251, 252] for more details) when searching for a common motif. In the more general case of unaligned input strings, however, we arrive at the problems CLOSEST SUBSTRING [178] and CONSENSUS PATTERNS [179]:

> **Input:** k strings $s_1, s_2, \ldots, s_k$ over alphabet Σ and nonnegative integers d and L.
> **Question in case of** CLOSEST SUBSTRING**:** Is there a string s of length L, and for every $i = 1, \ldots, k$, a substring s'_i of length L such that, for all $i = 1, \ldots, k$, $d_H(s, s'_i) \leq d$?
> **Question in case of** CONSENSUS PATTERNS**:** Is there a string s of length L, and for every $i = 1, \ldots, k$, a substring s'_i of length L such that $\sum_{i=1}^{k} d_H(s, s'_i) \leq d$?

Here $d_H(s, s'_i)$ denotes the Hamming distance between s and s'_i. What is currently known about these two problems is summarized as follows. CLOSEST SUBSTRING is *NP*-complete, and remains so for the special case CLOSEST STRING, where the string s that we search for is of same length as the input strings. CLOSEST STRING is *NP*-complete even for the further restriction to a binary alphabet [116, 174]. On the positive side, both CLOSEST SUBSTRING and CLOSEST STRING admit polynomial time approximation schemes (PTAS's) where the objective function to minimize is the distance of the "closest string" s [178, 179]. In the PTAS's for both CLOSEST STRING and CLOSEST SUBSTRING, the exponent of the polynomial bounding the running time depends on the goodness of the approximation. These are not efficient PTAS's (EPTAS's) in the sense of [57] and, therefore, probably are of limited interest for bioinformatics practice.

CONSENSUS PATTERNS is *NP*-complete and remains so for the restriction to a binary alphabet [178]; it admits a PTAS [179] where the objective function to minimize is the distance of the "consensus string" s. The known PTAS's for CONSENSUS PATTERNS are not EPTAS's.

The key distinguishing point between CLOSEST SUBSTRING and CONSENSUS PATTERNS lies in the definition of the distance measure d between the "solution" string s and the substrings of the k input strings. CLOSEST SUBSTRING uses the maximum distance metric and CONSENSUS PATTERNS uses the sum of distances metric. This is of particular importance when discussing values of parameter d occurring in practice. Whereas it makes good sense for many applications to assume that d is a fairly small number in case of CLOSEST SUBSTRING, this is a less reasonable assumption in the case of CONSENSUS PATTERNS. This will be of some importance when discussing the result for CONSENSUS PATTERNS.

Parameter	*Constant size alphabet*	*Unbounded alphabet*
d	?	$W[1]$-hard
k	$W[1]$-hard	$W[1]$-hard
d, k	?	$W[1]$-hard
L	FPT	$W[1]$-hard
d, k, L	FPT	$W[1]$-hard

Table 5.1. Overview on the parameterized complexity of both Closest Substring and Consensus Patterns with respect to different parameterizations, where k is the number of given strings, L is the length of the substrings we search for, and d is the Hamming distance allowed. The FPT results for constant size alphabet can be achieved by enumerating all length L strings over Σ. Cases where the parameterized complexity is not known are indicated by a question mark.

Many algorithms applied in practice try to solve motif search problems exactly, often using enumerative approaches in combination with heuristics [36, 45, 221]. Until recently, the parameterized complexity of Closest Substring and Consensus Patterns was completely open. Of course, similar fixed-parameter tractability results for parameters both d and k as those described for Closest String in Sections 3.3 and 4.1 would be highly desirable. Unfortunately, the news in this respect are bad [106]: Closest Substring and Consensus Patterns are $W[1]$-hard with respect to the parameter k of the number of input strings even in case of a binary alphabet. For unbounded alphabet size, the problems are $W[1]$-hard for the combined parameters L, d, and k. In the case of constant alphabet size, the complexity of the problems remain open when parameterized by d and k together, or by d alone. Note that in the case of Consensus Patterns these result gains particular importance because here the distance parameter d usually is not small, whereas assuming small k is reasonable. Until now, it was only known that if one additionally considers the substring length L as a parameter then running times exponential in L can be achieved [36, 100, 235]. An overview on known parameterized complexity results for Closest Substring and Consensus Patterns is given in Table 5.1 (taken from [106, 129]). Thus, the above results give strong theory-based support for the common intuition that Closest Substring ($W[1]$-hard) seems to be a much harder problem than Closest String (in FPT [137]). Notably, this could *not* be expressed by "classical complexity measures" since both problems are NP-complete as well as both do have a PTAS. The parameterized complexity of Closest Substring and Consensus Patterns, parameterized by "distance parameter" d, remains open for alphabets of constant size. If these problems are also $W[1]$-hard, then an efficient and practically useful PTAS would appear to be impossible [57, 88] unless further structure of natural input distributions is taken into account in a more complex aggregate parameterization of these basic computational problems of bioinformatics.

In order to give some flavor of the in general technically involved $W[1]$-hardness proofs, following [106] we sketch the construction for the case of unbounded alphabet size. Then, Closest Substring is $W[1]$-hard with respect to every combination of the parameters k, L, and d. Note that the transfer to binary alphabet for parameter k still needs significantly more technical effort. Refer to [106, 129] for all the details.

We describe a reduction of the $W[1]$-complete Clique problem to Closest Substring which is a parameterized m-reduction with respect to the aggregate parameter (L, d, k) in case of unbounded alphabet size.

Reduction of Clique to Closest Substring

A Clique instance is given by a graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_n\}$, a set E of m edges, and a nonnegative integer k denoting the desired clique size. We describe how to generate a set S of $\binom{k}{2}$ strings such that G has a clique of size k (k-clique for short) iff there is a string s of length $L := k + 1$ such that every $s_i \in S$ has a substring s'_i of length L with $d_H(s, s'_i) \leq d := k - 2$. If a string $s_i \in S$ has a substring s'_i of length L with $d_H(s, s'_i) \leq d$, we call s'_i a *match*. We assume $k > 2$, because $k = 1, 2$ are trivial cases.

Alphabet. The alphabet of the produced instance is given by the disjoint union of the following sets:

- $\{\, \sigma_i \mid v_i \in V \,\}$, i.e., an alphabet symbol for every vertex of the input graph; we call them *encoding symbols*;
- $\{\, \varphi_j \mid j = 1, \ldots, \binom{k}{2} \,\}$, i.e., a unique symbol for every of the $\binom{k}{2}$ produced strings; we call them *string identification symbols*;
- $\{\#\}$ which we call the *synchronizing symbol*.

This makes a total of $n + \binom{k}{2} + 1$ alphabet symbols.

Choice strings. We generate a set of $\binom{k}{2}$ *choice strings* $S_c = \{c_{1,2}, \ldots, c_{1,k}, c_{2,3}, c_{2,4}, \ldots, c_{k-1,k}\}$ and we assume that the strings in S_c are ordered as shown. *Every* choice string will encode the whole graph; it consists of m concatenated strings, each of length $k+1$, called *blocks*; by this, we have one block for every edge of the graph. The blocks will be separated by *barriers* which are length k strings consisting of k identification symbols corresponding to the respective string. A choice string $c_{i,j}$ which, according to the given order, is the i'th choice string in S_c, is given by

$$c_{i,j} := \langle \text{block}(i, j, e_1) \rangle \, (\varphi_{i'})^k \, \langle \text{block}(i, j, e_2) \rangle \, (\varphi_{i'})^k \ldots (\varphi_{i'})^k \, \langle \text{block}(i, j, e_m) \rangle,$$

where $e_1, e_2, \ldots, e_m$ are the edges of G and $\langle \text{block}() \rangle$ will be defined below. The solution string s will have length $k+1$ which is exactly the length of one block.

Block in a choice string. Every block is a string of length $k + 1$ and it encodes an edge of the input graph. Every choice string contains a block for every edge of the input graph; different choice strings, however, encode the

edges in different positions of their blocks: For a block in choice string $c_{i,j}$, positions i and j are called *active* and these positions encode the edge. Let e be the edge to be encoded and let e connect vertices v_r and v_s, $1 \leq r < s \leq n$. Then, the ith position of the block is σ_r in order to encode v_r and the jth position is σ_s in order to encode v_s. The last position of a block is set to the synchronizing symbol #. Let $c_{i,j}$ be the i'th choice string in S_c; then, all remaining positions in the block are set to $c_{i,j}$'s identification symbol $\varphi_{i'}$. Thus, the block is given by

$$\langle \text{block}(i, j, (v_r, v_s)) \rangle \;:=\; (\varphi_{i'})^{i-1}\, \sigma_r\, (\varphi_{i'})^{j-i-1}\, \sigma_s\, (\varphi_{i'})^{k-j}\, \#.$$

Values for L and d. We set $L := k + 1$ and $d := k - 2$.

Correctness of the Reduction

To prove the correctness of the proposed reduction, we have to show an equivalence consisting of two directions. The easier direction is to see that a k-clique implies a closest substring fulfilling the given requirements.

Proposition 5.1.1. *For a graph with a k-clique, the above construction produces an instance of* Closest Substring *which has a solution, i.e., there is a string s of length L such that every $c_{i,j} \in S_c$ has a substring $s_{i,j}$ with $d_H(s, s_{i,j}) \leq d$.*

Proof. Let the input graph have a clique of size k. Let $h_1, h_2, \ldots, h_k$ denote the indices of the clique's vertices, $1 \leq h_1 < h_2 < \cdots < h_k \leq n$. Then, we claim that a solution for the produced Closest Substring instance is

$$s := \sigma_{h_1} \sigma_{h_2} \ldots \sigma_{h_k} \#.$$

Consider choice string $c_{i,j}$, $1 \leq i < j \leq k$. As the vertices $v_{h_1}, v_{h_2}, \ldots, v_{h_k}$ form a clique, we have an edge connecting v_{h_i} and v_{h_j}. Choice string $c_{i,j}$ contains a block $s_{i,j} := \langle \text{block}(i, j, (v_{h_i}, v_{h_j})) \rangle$ encoding this edge:

$$s_{i,j} \;:= (\varphi_{i'})^{i-1}\, \sigma_{h_i}\, (\varphi_{i'})^{j-i-1}\, \sigma_{h_j}\, (\varphi_{i'})^{k-j} \#,$$

where i' is the number (according to the given order) of the choice string in S_c. We have $d_H(s, s_{i,j}) = k-2$, and we can find such a block for every $c_{i,j}$, $1 \leq i < j \leq k$. □

For the reverse direction, we show in Proposition 5.1.2 that a solution in the produced Closest Substring instance implies a k-clique in the input graph. To this end, we need two technical lemmas which show that a solution to the instance constructed in Subsection 5.1.3 has encoding symbols at its first k positions and the synchronizing symbol # at its last position. The proofs are omitted and can be found in [106, 129].

Lemma 5.1.3. *A closest substring s contains at least two encoding symbols and at least one synchronization symbol.* □

Based on Lemma 5.1.3, one can now exactly specify the numbers and positions of the encoding and synchronizing symbols in the closest substring.

Lemma 5.1.4. *A closest substring s contains encoding symbols at its first k positions and a symbol # at its last position.* □

Proposition 5.1.2. *The first k characters of a closest substring correspond to k vertices of a clique in the input graph.*

Proof. By Lemma 5.1.4, a closest substring s has encoding symbols at its first k positions and a synchronizing symbol at its last position. Consequently, the blocks are the only possible matches of s in the choice string. Now, assume that $s = \sigma_{h_1}\sigma_{h_2}\ldots\sigma_{h_k}\#$ for $h_1, h_2, \ldots, h_k \in \{1, \ldots, n\}$. Consider any two h_i, h_j, $1 \leq i < j \leq k$, and choice string $c_{i,j}$. Recall that in this choice string, the blocks encode edges at their ith and jth position, they have # at their last position, and all their other positions are set to a string identification symbol unique for this choice string. Thus, we can only find a block that is a match if there is a block with σ_{h_i} at its ith position and σ_{h_j} at its jth position. We have such a block only if there is an edge connecting v_{h_i} and v_{h_j}. Summarizing, the closest substring s implies that there is an edge between every pair of $\{v_{h_1}, v_{h_2}, \ldots, v_{h_k}\}$; these vertices form a k-clique in the input graph. □

Propositions 5.1.1 and 5.1.2 establish the following hardness result. Note that hardness for the combination of all three parameters also implies hardness for each subset of the three.

Theorem 5.1.3. Closest Substring *with unbounded alphabet is $W[1]$-hard for every combination of the parameters L, d, and k.* □

Some Positive News

We briefly sketch a possible way how to "circumvent" the above $W[1]$-hardness misery. We present an enumerative type algorithm that uses Closest String as a subproblem. Similar to [221], in a first phase enumerate candidates, where, in our case, the "filtering process" is more elaborate since, in a second phase, we check the outcome of the first phase:

1. Identify the "k-cliques" of substrings, taking one substring from each given string, with pairwise distance at most $2d$ each time.
2. For each such candidate set, use a Closest String algorithm to test whether it gives rise to a motif.

Step 1 is a recursive algorithm that enumerates all possible sets of substrings: For each length L substring s_1 of the first input string, it considers each length L substring s_2 of the second input string with distance at most d to s_1. For each such pair s_1, s_2 it considers each length L substring s_3 with distance at most d to s_1 and s_2, and continues recursively. A refinement of the algorithm exploits information gained from the cases where two substrings in the set have distance exactly $2d$: then we can restrict the set of possible closest strings since each of its characters must match the corresponding character in one of the two substrings.

The poster abstract [136] reports on successful experiments with this approach.

5.1.4 Structure Comparison for RNA

Structure comparison for RNA and for protein sequences has become a central computational problem bearing many challenging computer science questions. In this context, the Longest Arc Preserving Common Subsequence problem (LAPCS) recently has received considerable attention [98, 99, 158, 183]. It is a sound and meaningful mathematical formalization of comparing the secondary structures of molecular sequences:[2] For a sequence s, an *arc annotation* A of s is a set of unordered pairs of positions in s. Focusing on the case of two given arc-annotated input sequences, LAPCS in its general version is defined as follows.

> **Input:** Two arc-annotated sequences s_1 and s_2 and nonnegative integers k_1 and k_2.
> **Question:** Can one delete at most k_1 letters (also called *bases*) from s_1—when deleting a letter at position i, then *all* arcs with endpoint i are also deleted—and at most k_2 letters from s_2 such that in both cases the same arc-annotated sequence t emerges?

Thus, t forms an arc-annotated subsequence of s_1 as well as s_2. For related studies concerning algorithmic aspects of (protein) structure comparison using "contact maps," refer to [123, 173].

Whereas the Longest Common Subsequence problem for two sequences without arc annotations is solvable in quadratic time (it only becomes *NP*-complete when allowing for an arbitrary number of input sequences[3]), LAPCS for two sequences is *NP*-complete [98, 99]. According to Lin *et al.* [183] LAPCS for *nested arc annotations* is "generally thought

[2] As usual in computational biology, we identify the terms "sequence" and "string" here. Note, however, that the terms "subsequence" and "substring" have to be clearly distinguished from each other, the first concept being the much more general one.

[3] Fixed-parameter studies for this problem with mostly "negative" results (i.e., intractability results) have been undertaken in [41, 42, 222] (also cf. [88]).

of as the most important variant of the LAPCS problem." Here, one requires that no two arcs share an endpoint and no two arcs cross each other, referred to by LAPCS(NESTED,NESTED). Answering an open question of Evans [98], Lin *et al.* [183] showed that LAPCS(NESTED,NESTED) is *NP*-complete. In addition, they gave polynomial time approximation schemes for (also *NP*-complete) special cases of LAPCS(NESTED,NESTED). Here, matches between two given input sequences are allowed only in a "local area" (of constant size) with respect to matching position numbers. As to the general LAPCS(NESTED,NESTED) problem, Jiang *et al.* [158] gave a quadratic time approximation algorithm with approximation ratio 1/2.

By way of contrast, here we briefly sketch a fixed-parameter algorithm that solves the general LAPCS(NESTED,NESTED) problem in running time $O(3.31^{k_1+k_2} \cdot n)$ where n is the maximum input sequence length. We refer to [10, 143] for any details.

The fixed-parameter algorithm for LAPCS(NESTED,NESTED) employs a bounded search tree. The case distinction in the search algorithm works as follows. For sake of clarity, we choose the presentation in a recursive style: Based on the current instance, we make a case distinction, branch into one or more subcases of somehow simplified instances and invoke the algorithm recursively on each of these subcases. The algorithm works through both given sequences from left to right and it considers the following main cases. Either

1. both sequences differ in the symbols at the current respective positions or
2. they carry the same symbols at this position. Then, either
   a) there are no arcs attached to these positions or
   b) there is only one arc, i.e., one of the sequences has an arc at its position and the other does not have an arc at its position or
   c) both positions have arcs attached to them but
      i. either the symbols at the right points of these arcs differ
      ii. or an arc match is really possible.

Actually, Case 2.c)ii. is the most complicated one and has to be split in several subcases. Refer to [10, 143] for the details. The worst-case branching vector is $(1, 1, 2, 1)$. It implies the worst-case upper bound $3.31^{k_1+k_2}$ on search tree size. Finally, the following result can be proven.

Theorem 5.1.4. LAPCS(NESTED,NESTED) *for two sequences s_1 and s_2 with $|s_1|, |s_2| \leq n$ can be solved in time $O(3.31^{k_1+k_2} \cdot n)$ where k_1 and k_2 are the number of deletions needed in s_1 and s_2.* □

The parameterized complexity of LAPCS(NESTED,NESTED) when parameterized by subsequence length instead of number of deletions is open. Depending on what (relative) length of the longest common subsequence is to be expected one or the other parameterization might be more appropriate. The complexity analysis is worst-case, however, and it is a topic of future

investigations to study the practical usefulness of the above search tree algorithm. In this context, it is also meaningful to take a closer look at the (special case) problem "APS(NESTED,NESTED)" where one asks whether a given sequence forms an arc-preserving subsequence of another. Very recent work [130] shows that APS(NESTED,NESTED) can be solved in quadratic time. This further supports the hope for practical implementations of the whole scenario. In addition, it was observed [130] that APS(NESTED,NESTED) generalizes a problem occurring in information retrieval [165, 166], thus showing that these string problems also occur in completely different contexts.

5.1.5 Final Remarks

Computational biology is a fruitful research area concerning fixed-parameter questions. We have seen that there often are several natural ways to parameterize given *NP*-hard problems. In what follows, we briefly list few more fixed-parameter results in various fields of computational biology.

Perfect Phylogeny

Agarwala and Fernández-Baca [1] (also cf. [88]) approach the question of building a phylogenetic tree for a given set of species in the following model. For a given set C of m characters they allow a character $c \in C$ to take one state of a fixed set of character states A_c. These characters may, e.g., represent properties of single organisms or the positions in its DNA sequence with the nucleotide bases being the character states. In this setting, we are given a set S of n species, for which we intend to construct a tree. Each species $s \in S$ is represented by a vector of character states $s \in A_1 \times \cdots \times A_m$. The PERFECT PHYLOGENY problem is then to determine whether there is a tree T with nodes $V(T) \subseteq A_1 \times \cdots \times A_m$ where each leaf of the tree is a species. In addition, we require for every $c \in C$ and every $j \in A_c$ the set of all nodes u of the tree with u's character c being in state j to induce a subtree. Downey and Fellows [88] refer to this problem as BOUNDED CHARACTER STATE PERFECT PHYLOGENY. This indicates that the parameter here is the maximum number of character states $r = \max_{c \in C} |A_c|$. Using a dynamic programming approach and building perfect phylogenies from bottom up, Agarwala and Fernández-Baca present an $O(2^{3r}(nm^3 + m^4))$ time algorithm which was improved by Kannan and Warnow to $O(2^{2r}nm^2)$ [162]. A generalization of PERFECT PHYLOGENY is the l-PHYLOGENY problem in which the question is to construct a tree T such that, given the fixed integer l, each character state does not induce more than l connected components in T. A parameterized analysis of the problem was initiated by Goldberg *et al.* [121].

Gene Duplication

For a given set of species, we may obtain several phylogenetic trees, e.g., by building trees based on different gene families. These so-called *gene trees* are a good hypothesis for a *species tree*, i.e., the evolutionary relationship of the species, if they are all the same. However, the gene trees can differ from the species tree. A way to explain the contradictions in the trees is the possibility that genes are duplicated in the evolutionary history and evolve independently. This observation motivates the following model to infer a species tree from several, possibly contradictory gene trees, the Gene Duplication problem: Given a set of species and a set of trees (the gene trees) with their leaves labeled from the species set, the question is, intuitively speaking, to find a tree (the species tree) that requires a minimum number of gene duplications in order to explain the given gene trees (refer to [107, 247] for further details). Note that in this model we count duplication events copying only one gene at a time. Stege [247] gives a fixed-parameter algorithm for Gene Duplication, with the allowed number of duplications as the parameter. As a duplication event in evolutionary history copies a piece of DNA with possibly many genes on it, Fellows *et al.* [107] study the Multiple Gene Duplication problem. In contrast to Gene Duplication, here, one duplication event copies a set of genes. With the upper bound on the number of duplications as parameter, they show even the easier version to be $W[1]$-hard where we are also given the species tree and only ask for the minimum number of required duplications.

Genome Rearrangements

Knowing the succession of genes on a chromosome, a way to measure the similarity of two corresponding chromosomes from different organisms with the same set of genes, is to count the number of mutation events required to obtain one succession of genes from the other. Examples of such mutation events are, e.g., inverting a subsequence, called *reversal*, or their deletion and insertion at another position, called *transposition*. Reversals are the most common kind of these mutations. Restricting to them, the comparison of two sequences of the same set of genes is modeled in the Sorting by Reversals problem: Given a permutation π of $\{1, 2, \ldots, n\}$, the question is to find the minimum number of reversals we need to transform π into the identity function. Sorting by Reversals is NP-complete [54]. Hannenhalli and Pevzner's results [146], however, imply a fixed-parameter algorithm for the problem when parameterized by the number of reversals. Another genome-level distance measure that was shown to be fixed-parameter tractable is the Syntenic Distance [109]. Herein, a genome is represented as a set of chromosomes and a chromosome is represented as a set of genes (which itself can be represented as positive integers). The mutation events in this model are the union of two chromosome sets, the splitting of a chromosome set into

two sets, and the exchange of genes between two sets. Given two genomes G_1 and G_2, the SYNTENIC DISTANCE problem is to compute the minimum number of mutation events needed to transform G_1 into G_2. DasGupta *et al.* [77] showed that computing the SYNTENIC DISTANCE is *NP*-hard and fixed-parameter tractable when parameterized by the distance. The currently best exact solution with respect to parameter d is due to Liben-Nowell [180], who gave an algorithm with running time $O(n^2 + 2^{O(d \log d)})$, where n is the number of genes in the given genome.

Several more problems and solutions with "fixed-parameter" flavor can be found—just to name a few let us mention various applications of suffix trees in the biological context [226, 235], the analysis of repeats such as in the SINGLE GENE DUPLICATION PROBLEM of [254], or the emerging field of studying "single nucleotide polymorphisms haplotyping" with applications including medical diagnostic and drug design [227].

5.2 Graph and Network Problems

Here, we come to our second main problem field in the context of fixed-parameter solving algorithms. We only pick a small, personally biased selection to have a somewhat closer look at. It goes without saying that this huge field alone bears many opportunities for designers of fixed-parameter algorithms. We start with our favorite ground problem.

5.2.1 Weighted VERTEX COVER Problems

VERTEX COVER without weights on the graph vertices is the most popular fixed-parameter tractable problem. Now, let us see what happens in the case that the input consists of a graph with various weights on its vertices:

> **Input:** A graph $G = (V, E)$, a weight function $\omega : V \to \mathcal{R}^+$, and $k \in \mathcal{R}^+$.
> **Question:** Does there exist a vertex cover set such that the sum of the weights of its vertices can be bounded by k?

Clearly, in the special case that ω assigns the value 1 to all vertices we have the standard unweighted VERTEX COVER problem. We consider three natural variants of WEIGHTED VERTEX COVER, following [207].

1. INTEGER-WVC, where the weights are arbitrary positive integers.
2. REAL-WVC, where the weights are real numbers ≥ 1.
3. GENERAL-WVC, where the weights are positive real numbers.

Whereas all three versions are clearly *NP*-complete, it turns out that their parameterized complexity differs significantly: While INTEGER-WVC and

Real-WVC are fixed-parameter tractable, General-WVC is *not* fixed-parameter tractable unless $P = NP$.

Before we come to the three subproblems named above, we briefly note that for Integer-WVC and Real-WVC clearly Buss' reduction to problem kernel (cf. beginning of Chapter 2) applies. Moreover, there are also weighted versions of the Nemhauser-Trotter theorem (cf. Theorem 2.4.1) that apply and that yield linear size problem kernels. As we want to apply the bounded search tree technique in the following, due to the interleaving technique presented in Section 3.7 the particular problem kernel size is of minor importance in the following and it will be neglected, therefore.

General- and Integer-WVC

Integer- and General-WVC are easily dealt with: General-WVC is NP-complete for any fixed $k > 0$, and there is a straightforward reduction from the unweighted Vertex Cover to General-WVC with $k = 1$. This implies, however, that there cannot be a time $f(k) \cdot n^{O(1)}$ or even $n^{O(k)}$ algorithm for General-WVC unless $P = NP$. This is true because otherwise we would obtain a polynomial time algorithm for an NP-complete problem. Hence, we have:

Proposition 5.2.1. General-WVC *is not fixed-parameter tractable unless* $P = NP$. □

In the remainder, we exhibit that we can easily reduce Integer-WVC to unweighted Vertex Cover via a simple parameterized many-one reduction that does not change the value of the parameter. To prove the following theorem, we may safely assume that the maximum vertex weight is bounded by k (the according preprocessing needs only polynomial time).

Theorem 5.2.1. Integer–WVC *can be solved as fast as unweighted* Vertex Cover *up to an additive term polynomial in* k.

Proof. An instance of Integer–WVC is transformed into an instance of Vertex Cover as follows: Replace each vertex i of weight u with a cluster i' consisting of u vertices. We do not add intra-cluster edges to the graph. Furthermore, if $\{i, j\}$ is an edge in the original graph, then we connect every vertex of cluster i' to every vertex of cluster j'. Now, it is easy to see that both graphs (the instance for Integer-WVC and the new instance for Vertex Cover) have minimum vertex covers of same weight/size. Herein, it is important to observe that the following is true for the constructed instance for Vertex Cover: Either all vertices of a cluster are in a minimum vertex cover or none of them is. Assume that one vertex of cluster i' is not in the cover but the remaining are. Then all vertices in all neighboring clusters have to be included and, hence, it makes no sense to include any vertex of cluster i' in the vertex cover.

Let $t(k, n)$ be the time needed to solve Vertex Cover. The running time of the algorithm on the "cluster instance" is clearly bounded by $t(k, wn) \leq t(k, kn)$, where $w \leq k$ is the maximum vertex weight in the given graph. Using the interleaving technique (see Section 3.7, the running time is not increased by a polynomial factor, but only by a polynomial amount of *additional* processing is needed. □

Real-WVC

Real-WVC allows a bounded search tree algorithm that is similar in flavor, but nevertheless significantly different from the search tree algorithm for unweighted Vertex Cover. We only sketch some integral parts of the algorithm and refer to [207] for further details. Recall that the weights are real numbers ≥ 1. We indicate that Real-WVC can be solved in time $O(1.40^k + kn)$. Observe, however, that the search tree algorithm, as usual, can only guarantee to find at least *one* optimal vertex cover, but *not* necessarily all of them. The basic outline is as follows. First, one observes that if a graph has maximum vertex degree two, then there is an easy dynamic programming solution. After that, one distinguishes three main cases (in the given order): when there is a vertex of degree one in the graph, when there is a triangle (i.e., a cycle of size 3) in the graph, and when there is no triangle in the graph. The overall structure of the algorithm is as follows. The subsequent instructions are executed in a loop until all edges of the graph are covered or $k \leq 0$ which means that no cover could be found.

1. If there is no vertex with degree greater than two then solve Real-WVC in linear time by dynamic programming.
2. Execute the lowest numbered, applicable step of the following.
   a) If there is a vertex x of degree at least four then branch into the two cases of either bringing itself or all its neighbors into the vertex cover. The corresponding branching vector is at least $(1, 4)$, implying branching number 1.39 or better.
   b) If there is a degree-one vertex then one can obtain the branching vector $(1, 4)$ or better, implying branching number 1.39 or better.
   c) If there is triangle in the graph then one can obtain the branching vector $(3, 4, 3)$ or better, implying branching number 1.40 or better.
   d) If there is no triangle in the graph then one can obtain the branching vector $(3, 4, 3)$ or better, implying branching number 1.40 or better.

Since in solving unweighted Vertex Cover degree-one vertices form a trivial case (that is, just put the neighbor of the degree-one vertex into the vertex cover), it is of interest to see why this case gets more complicated for Real-WVC. Thus, we provide the details of Case 2.b) above in the following.

Assume that there is at least one vertex that has degree one. Let x be such a vertex and let a be its only neighbor. In addition, let w be the weight

of x and let w' be the weight of a. If $w \geq w'$ then it is optimal to include a into the vertex cover. In the following, we handle the more complicated situation that $w < w'$.

Case 1: a has degree 2. Then, a path starts at x that proceeds over vertices with degree 2 and ends in a vertex y that has degree 1 or 3. If y has degree 1 then we can find an optimal cover for this graph component by dynamic programming. Otherwise, we branch on y, bringing either y or its three neighbors into the vertex cover. This gives branching vector $(1, 3)$. If we put y into the vertex cover then we create a new graph component that includes x and a and has only vertices with degree at most 2. We can again apply dynamic programming and we get a branching vector at least $(2, 3)$ for the whole subgraph. We only mention in passing here that such a kind of "bonus point system" where we obtain "easy graph components" that can be handled without branching of the recursion will be reconsidered (and discussed) in Subsection 5.2.2.

Case 2: a has degree 3 and it has at least one neighbor with degree 3. Let y be a's degree-3 neighbor. We branch on y. If y is in the cover, then a will have degree 2 and Case 1 applies. The $(1, 3)$ branching vector thus can be improved to $(1 + 2, 1 + 3, 3) = (3, 4, 3)$.

Case 3: a has degree 3 and it has two neighbors with degree 2. Let y and b be a's degree-2 neighbors. We branch on x. If x is in the cover, then a is not and a's other neighbors y and b are in the cover. This gives branching vector $(1, 3)$, which is not yet good enough. Hence, by considering several more subcases, we do a more complicated branching.

Let z be y's other neighbor and assume that y has weight u, z has weight v, and $u \geq v$. Then we can branch on a and get branching vector $(2, 3)$; note that if a is in the cover, then it is optimal to also include z (instead of y).

Assume next that the weight w' of a is at least 2: Then, branching on a, we have branching vector $(2, 3)$. Let w be the weight of x. We can assume in the following that $w' < w + v$ and $u < v$.

Let us return to the branch on x: If x is in the cover, so are y and b. We can now assume that z is *not* in the cover. Otherwise, we could replace x and y with a, which is better and is already covered by the branch that does not include x in the vertex cover. Then all neighbors of z are in the cover, too, and among them must be some vertex other than x, y, or b. If not, interchange the roles of y and b. In this way, we get a branching vector of at least $(1, 4)$ (unless the component has only six vertices and, thus, can be handled in constant time).

Case 4: Remaining cases. What remains to be considered are the case when a has degree 3 and all its neighbors have degree 1, and when a has degree 3 and two of its neighbors have degree 1 and one has degree 2. The first case is easily handled in constant time, because we then have a graph component of constant size. For to the second subcase, basically the same strategy as in

Case 1 applies, because the second degree 1 neighbor of a (besides x) only makes necessary a slight, obvious modification to what is done in Case 1.

The other two main cases 2.c) and 2.d) above need (in size) similar case distinctions. In this way, the following can be proven (see [207] for details):

Theorem 5.2.2. Real-WVC *can be solved in time* $O(1.40^k + kn)$. □

5.2.2 Constraint Bipartite Vertex Cover

Constraint Bipartite Vertex Cover (CBVC for short) is defined as follows.

> **Input:** A bipartite graph $G = (V_1, V_2, E)$ and two nonnegative integers k_1 and k_2.
> **Question:** Are there two subsets $C_1 \subseteq V_1$ and $C_2 \subseteq V_2$ of sizes $|C_1| \leq k_1$ and $|C_2| \leq k_2$ such that each edge in E has at least one endpoint in $C_1 \cup C_2$?

The existence of *two* parameters and two vertex sets makes Constraint Bipartite Vertex Cover (CBVC) quite different from the original Vertex Cover problem. Thus, whereas classical Vertex Cover (with only one parameter) restricted to bipartite graphs is solvable in polynomial time (because it is equivalent to a polynomial time solvable maximum matching problem for bipartite graphs [82, 171]), by a reduction from Clique it has been shown that CBVC is *NP*-complete [171]. For the application in reconfigurable VLSI design, see [171, 111].

A Fixed-Parameter Algorithm for CBVC

The algorithm works in basically the same way as the fixed-parameter algorithms for Vertex Cover do. The main part is to build a bounded search tree: To cover an edge, we have to put at least one of its two endpoints into the (optimal) vertex cover sets. Thus, starting with an arbitrary edge, we can make a binary decision between its two endpoints. In each subcase, we delete the corresponding vertex chosen and its incident edges and repeat this until we have built a search tree of size $2^{k_1+k_2}$. As a consequence, it is easy to obtain an algorithm running in time $O(2^{k_1+k_2}(n+m))$, where n denotes the number of vertices and m denotes the number of edges in the graph. The exponential base can be significantly improved, however.

The algorithm as a whole consists of three pieces:

1. A reduction to problem kernel;
2. a search tree processing;
3. a special treatment of graphs consisting of vertices with maximum degree two and some slightly more general graphs.

Only the second part of the algorithm has exponential time complexity. As is usually the case, we achieve a reduction of the search tree size by distinguishing between the degree of graph vertices. Since for CBVC we have to minimize with respect to two parameters, this gets significantly harder than in the classical Vertex Cover case. For instance, in the classical instance, taking the neighbor of a degree-one vertex will always lead to an optimal vertex cover. Thus, a branching in the search tree is avoided. This is no longer possible in the CBVC case because the neighbor belongs to the second vertex set in the bipartite graph and we have to minimize with respect to two vertex cover set sizes. In particular, the size of a minimal solution for CBVC is no longer uniquely determined because the *signature* $s := (|C_1|, |C_2|)$ of a vertex cover C_1 and C_2 is a tuple of numbers instead of simply a number. The algorithm provides, however, for each minimal s a corresponding minimal solution.

Before getting a bit more detailed about the cases under consideration, we firstly observe that the idea of getting rid of high-degree vertices (see Buss' reduction to problem kernel for Vertex Cover) also works in this setting. This simple observation was already used by Kuo and Fuchs [171] in 1987, which led to the so-called "must-repair-analysis" preprocessing in their algorithms. Deleting all these "high-degree-vertices" together with their incident edges, we can infer that after reduction to problem kernel the graph consists of at most $2k_1k_2 + k_1 + k_2$ vertices and at most $2k_1k_2$ edges. By assuming appropriate input data structures, this reduction to problem kernel can be implemented to run in time $O((k_1 + k_2)n)$, where $n := |V_1 \cup V_2|$ is the number of vertices in G.

Let us briefly begin with the easy special case that all graph vertices have maximum degree two. Clearly, we can deal with each connected component separately. So, let us assume that the graph is connected. If the maximum vertex degree of a graph is at most two, then CBVC is easy to solve: We know that, in this case, the graph is either a cycle or a path. In both cases, however, it is fairly easy to compute the linear number of possible minimal vertex covers in linear time. We omit the lesser details. In addition, as previously mentioned, here we have to take into consideration that our given graph may be split into several connected components. Since the various components are "independent" from each other, we simply can combine them using component-wise addition and then again looking for the minimal values. Consequently, by using simple data structures, this can be done in $O((k_1 + k_2)^2)$ time, because $1 + \min(k_1, k_2)$ is an upper bound for the number of minimal vertex covers belonging to a component. Furthermore, we can assume that there are at most $k_1 + k_2$ components since otherwise the graph is not coverable and we know that each output of merging two minimal vertex covers always is bounded by $O(k_1 + k_2)$. As a result of this, we have $k_1 + k_2$ merge steps each of time complexity $O((k_1 + k_2)^2)$. Summarized, this gives:

Proposition 5.2.2. *For bipartite graphs with maximum vertex degree two CBVC can be solved in time* $O((k_1 + k_3)^3)$. □

Because of Proposition 5.2.2 in the following description of the basic structure of our search tree algorithm we now may concentrate on graphs with maximum degree at least three.

The following algorithm skeleton leads to the so far best fixed-parameter algorithm for CBVC. The technical details of the branchings and the corresponding numerous case distinctions are quite awkward and are omitted. We refer to [111] for the complete algorithm. Here, we only sketch the basic structure of the algorithm. It is of central importance for the correctness of our algorithm to execute the various steps in the given order—that is, we always choose an applicable step with minimal number:

1. If there is a vertex v with degree at least four, then branch according to v and its neighbors.
   Branching vector and branching number: $(1, 4)$ and 1.39.
2. If the graph is three-regular then pick any vertex v and branch according to v and its neighbors. (This step has to be applied at most once and thus does not influence the algorithm's asymptotic complexity. Similarly, if G contains only a small constant number of vertices of degree three, such a branching does not affect the overall time analysis.)
3. Deal with tails (i.e., a degree-three vertex followed by a (possibly empty) sequence of degree-two vertices, followed by one degree-one vertex) of size at least two.
   Branching vector and branching number: $(2, 3)$ and 1.33.
4. Deal with cycles of length four.
   Branching vector and branching number: $(2, 2)$ and 1.42. This can be improved to $(6, 7, 7, 7, 7, 9, 9, 9, 9, 8, 8, 10, 10, 10, 10, 12)$ and 1.40 by a lengthy case analysis [111].
5. Deal with chains (i.e., paths consisting of degree-two vertices) of length at least three.
   Branching vector and branching number: $(3, 4, 3)$ and 1.40.
6. Deal with degree-three vertices with three neighbors of degree two.
   Branching vector and branching number: $(4, 6, 6, 7, 3)$ and 1.40.
7. Deal with degree-three vertices with two neighbors of degree two.
   Branching vector and branching number: $(6, 7, 4, 2)$ and 1.40.
8. Deal with degree-three vertices with one neighbor of degree two.
   Branching vector and branching number:
   $(8, 9, 8, 10, 11, 10, 10, 11, 10, 10, 12, 11, 9, 10, 9, 10, 12, 11, 7, 9, 8, 9, 10, 9)$ and 1.40.

The above steps can be easily shown to provide a complete case distinction handling all cases that may occur. More specifically, from this point of view, steps 3–5 even would be superfluous—they are, however, necessary in order to get a small search tree size by handling "nice special cases" in advance.

The harder cases shown above are 4, 6, 7, and 8. The worst case branching vector occurs in case 8 and it implies a search tree size $1.3999^{k_1+k_2}$, rounded to $1.40^{k_1+k_2}$.

Combining the above sketched search tree algorithm with the reduction to problem kernel from the beginning, and applying the interleaving technique from Subsection 3.7, the following can be proven [111].

Theorem 5.2.3. Constraint Bipartite Vertex Cover *can be solved in running time* $O(1.40^{k_1+k_2} + (k_1 + k_2)n)$. □

The Deferred Analysis Trick

Concerning the analysis of the search tree size above, we want to point to a small trick that might be useful elsewhere. The idea is to use a kind of bonus points to reduce the search tree size, i.e., to measure the expected parameter reduction for certain graph components which are analyzed in detail later with a polynomial-time algorithm. More precisely, the trick of deferred analysis works as follows. We have noted that we can cope with a graph having vertices of degree at most two in polynomial time (Proposition 5.2.2). Therefore, if we create a non-empty line component starting at a degree-two vertex v we can make use of the fact that in order to cover that component at least one vertex from that component is needed in the cover. Although we do not know which vertex to take into the cover, we can safely decrease the parameter value by one more unit. In other words: the exact analysis of the path component is deferred to the third polynomial-time phase of the algorithm which deals with all remaining degree-two components as a whole. Nevertheless, already at that point of time (i.e., during the search tree processing) we have the "bonus" to decrease the search tree depth bound by one more unit.

A Simplified Version of CBVC

Chen and Kanj [58] considered a simplified version of CBVC which allows for a more efficient fixed-parameter algorithm. The Constrained Minimum Vertex Cover problem is defined as follows.

> **Input:** A bipartite graph $G = (V_1, V_2, E)$ and two nonnegative integers k_1 and k_2.
> **Question:** Is there a minimum vertex cover of G with at most k_1 vertices in V_1 and at most k_2 vertices in V_2?

Note that the decisive difference to CBVC as considered before is that here one asks for a *minimum* vertex cover (i.e., adding up the number of vertices from both vertex sets of the bipartite graph) under the given "constraints" k_1 and k_2 whereas CBVC minimizes with respect to the constraints. In particular, the previously defined term signature does not make sense for Constrained Minimum Vertex Cover.

The nice thing about Constrained Minimum Vertex Cover is that due to its somewhat simpler combinatorial structure it allows for simpler and more efficient algorithms than CBVC does. In particular, classical results from matching theory become applicable and allow for a much simpler search tree structure. The key tool is the so-called Gallai-Edmonds structure theorem from matching theory (cf. [185]) which implies a reduction to problem kernel. More precisely, based on this theorem it can be shown that there is a linear problem kernel consisting of only $2(k_1+k_2)$ vertices, and, moreover, the corresponding kernel graph has a perfect matching (see [58] for details.) Then, the so-called Dulmage-Mendelsohn decomposition [185] for graphs with a perfect matching is applied. This leads to a much simpler search tree procedure than the one known for CBVC. In summary, Constrained Minimum Vertex Cover thus can be solved in time $O(1.26^{k_1+k_2} + kn)$, where n is the number of graph vertices [58].

5.2.3 Maximum Cut

The *NP*-complete Maximum Cut (MaxCut) problem is another example for a graph and network problem that plays an important role in algorithm theory and practice (refer to Poljak and Tuza [223] for a survey).

> **Input:** A graph $G = (V, E)$ where edges are assigned integer weights and a nonnegative integer k.
> **Question:** Is there a cut of maximum weight, i.e., is there a partition of V into V_1 and V_2 such that the sum of weights over those edges $(s, t) \in E$ for which $s \in V_1$ and $t \in V_2$ is at least k?

The special thing about MaxCut is that we can easily treat it as Maximum 2-Satisfiability (Max2Sat) problem, that is, we can easily reduce MaxCut to Max2Sat. The resulting formulas expose a very special structure. After presenting the reduction, we formulate, in the following, a condition that tries to capture this structure. Then, to derive an exact algorithm for MaxCut one can develop an exact algorithm for a special kind of Max2Sat. We follows parts of [131] in our presentation.

For the reduction of MaxCut to Max2Sat [223], translate a graph $G = (V, E)$ into a formula in conjunctive normal form with clauses containing two literals (2-CNF) that has the vertices as variables and that has clause set

$$C = \{\, (w, \{i, j\}) \mid \text{edge } \{i, j\} \in E \text{ having weight } w \,\} \\ \cup \{\, (w, \{\bar{i}, \bar{j}\}) \mid \text{edge } \{i, j\} \in E \text{ having weight } w \,\}.$$

In this way, a graph having n vertices and m edges of total weight M results in a formula having n variables and $2m$ clauses of total weight $2M$. All these clauses are 2-clauses. The graph G has a cut of weight k iff the formula has simultaneously satisfiable clauses of weight $M + k$; every optimal assignment to the formula translates into a maximum cut, namely with all

vertices corresponding to satisfied variables on one side and all vertices corresponding to falsified variables on the other side. An assignment satisfying a maximum number of clauses in the resulting formula will satisfy at least one of the clauses $(w, \{i, j\})$ and $(w, \{\bar{i}, \bar{j}\})$, which are created for an edge $\{i, j\}$ of weight w, but will satisfy both clauses only if the edge is in the cut.

As we can see, the formulas created by this reduction exhibit a characteristic structure which we call MaxCut *Condition (MCC)*:

> For each 2-clause of weight w containing literals x and y, there is also a 2-clause of weight w containing literals $\bar{x}$ and $\bar{y}$.

Now, it can be shown that there is an exact algorithm for this special kind of Max2Sat problem that makes use of (MCC) and keeps (MCC) as an invariant of the algorithm when applying transformation and splitting rules in a way analogous to the general MaxSat problem (cf. Section 3.5). Thus, one may obtain a time $\text{poly}(|F|) \cdot 2^{K_2/6}$ algorithm, where K_2 is the total weight of 2-clauses in F and $|F|$ is the length of representation of the input. This translates back into an exact algorithm for MaxCut, see [131] for details. Then, given a graph G having n vertices and edges of total weight M, we can solve (weighted) MaxCut in time $\text{poly}(|G|) \cdot 2^{M/3}$, where $|G|$ is the length of representation of the input. Very recently, Fedin and Kulikov [101] reported on an improvement of this time bound to $\text{poly}(|G|) \cdot 2^{M/4}$ by employing a direct search tree approach for MaxCut.

Similar to the case of Max2Sat (cf. Subsection 3.5.4), a parameterization of MaxCut with the parameter "total weights of the edges in the cut" so far obviously does not lead to potentially faster algorithms than the above discussed, non-parameterized algorithm. MaxCut (besides MaxSat) has been considered in the parameterizing above guaranteed values setting (cf. Subsection 1.5.2) by Mahajan and Raman [186]. Here, analogous remarks as for MaxSat and Max2Sat apply (cf. concluding discussion of Section 3.5).

5.2.4 Planar Graphs Revisited

With Sections 4.4 and 4.5 we already exhibited special fixed-parameter algorithms for *NP*-complete planar graph problems. These algorithms were based on dynamic programming on tree decompositions of graphs where the point was that many parameterized planar graph problems allow for small treewidth. Here, we briefly sketch a similar approach but now based on the divide-and-conquer strategy using well-known graph separation theorems. The underlying material can be found in much greater depth in [8].

Definition 5.2.1. *Let $G = (V, E)$ be a graph. A* separator *$S \subseteq V$ of G divides V into two* parts *$A_1 \subseteq V$ and $A_2 \subseteq V$ such that*[4]

[4] In general, of course, A_1, A_2 and S will be non-empty. In order to cover boundary cases we did not put this into the separator definition.

- $A_1 + S + A_2 = V$, *and*
- *no edge joins vertices in* A_1 *and* A_2.

We write δA_1 *(or* δA_2*) as shorthand for* $A_1 + S$ *(or* $A_2 + S$*, respectively). The triple* (A_1, S, A_2) *is also called a* separation *of* G.

Clearly, this definition can be generalized to the case where a separator partitions the vertex set into ℓ subsets instead of only two. The techniques we mention here all are based on the existence of "small" graph separators, which means that $|S|$ is bounded by $o(|V|)$.

Definition 5.2.2. *According to Lipton and Tarjan [184], an* $f(\cdot)$-separator theorem (with constants $\alpha < 1$, $\beta > 0$) *for a class* $\mathbb{G}$ *of graphs which is closed under taking subgraphs is a theorem of the following form: If* G *is any* n*-vertex graph in* $\mathbb{G}$*, then there is a separation* (A_1, S, A_2) *of* G *such that*

- *neither* A_1 *nor* A_2 *contains more than* αn *vertices, and*
- S *contains no more than* $\beta f(n)$ *vertices.*

Again, this definition easily generalizes to ℓ-separators with $\ell > 2$.

Stated in this framework, the planar separator theorem due to Lipton and Tarjan [184] is a $\sqrt{\cdot}$-separator theorem with constants $\alpha = 2/3$ and $\beta = 2\sqrt{2} \approx 2.83$. The currently best value for $\alpha = 2/3$ is $\beta = \sqrt{2/3} + \sqrt{4/3} \approx 1.97$ [84]. Djidjev has shown a lower bound of $\beta \approx 1.55$ for $\alpha = 2/3$ [83]. For $\alpha = 1/2$, the "record" of $\beta = 7 + 1/\sqrt{3} \approx 7.58$ due to Venkatesan [259] was recently outperformed by Bodlaender [40], yielding $\beta = 2\sqrt{6} \approx 4.90$. A lower bound of $\beta \approx 1.65$ is known in this case [244]. For $\alpha = 3/4$, the best known value for β is $\sqrt{2\pi/\sqrt{3}} \cdot (1 + \sqrt{3})/\sqrt{8} \approx 1.84$ with a known lower bound of $\beta \approx 1.42$, see [244].

The basic idea to develop divide-and-conquer algorithms then simply is to divide the graph into parts using graph separators, solving the arising subproblems recursively, and then to "glue" together the solutions of the subproblems to obtain the solution of the whole problem. The paper [8] provides a formal framework to characterize problems that allow for such an approach in the fixed-parameter context, coining the very technical notions of "select& verify" problems and "glueability" of graph problems. The technical expenditure is too big in order to present it within this work.[5] Let us just mention that the key notion of glueable select&verify problems captures intricate graph problems such as Dominating Set or Total Dominating Set. Various glueable select&verify problems allow time $c^{\sqrt{k}} \cdot n^{O(1)}$-algorithms on graph classes that admit a $\sqrt{k}$-separator theorem. Then, the constant c is determined in terms of some problem-specific parameters. By exploiting further ideas on the use of separator theorems, one may lower these constants.

[5] Meanwhile, Alber [2] developed a significantly simplified exposition of this framework.

Observe that time $c^{\sqrt{k}} \cdot n^{O(1)}$ algorithms easily follow from this framework for parameterized planar graph problems with linear size problem kernel. Since constants occurring in separator theorems directly contribute to the constant c above, work on improving these is desirable in order to lower c. So far, however, the constant c and the further overhead involved still seem too big in order to give hope for a practical implementation of this approach. It might at least serve as a classification tool for fixed-parameter tractability and, interestingly, this approach in principle is also applicable to non-planar graphs, as well. Refer to [8] for more details and to [2] for a comprehensive overview on this framework.

5.2.5 Final Remarks

Compared to computational biology problems, as a rule, graph and network problems usually are easier to formalize and to understand. They are not easier to solve, though. In what follows, we briefly list few more fixed-parameter results and problems in this area.

Graph Modification

The *NP*-complete Minimum Fill-In problem asks whether a graph can be triangulated by adding at most k edges. Kaplan *et al.* [163] developed a search tree based $O(2^{4k}m)$ time algorithm (which improves to $O((4^k/(k+1)^{3/2})(m+n))$ due to a refined analysis by Cai [50]) and a more intricate $O(k^2nm + k^6 2^{4k})$ algorithm for the problem. In those fixed-parameter algorithms, n denotes the number of vertices and m denotes the number of edges in the graph. This also illustrates that it can be very important (and difficult!) to make the exponential term "additive" (as in the second case) instead of only "multiplicative" (as in the first case). In addition, Kaplan *et al.* show that Proper Interval Graph Completion (with a motivation from computational biology) and Strongly Chordal Graph Completion, both *NP*-hard, are fixed-parameter tractable (see [163] for details). Closely related graph modification problems (with applications to clustering problems) are considered in [198, 239]. They deserve more attention from a parameterized point of view.

Layout Problems

Two classical problems here and to some extent also in the parameterized complexity field are Cutwidth and Bandwidth. We start with Cutwidth. A *layout* of a graph $G = (V, E)$ is a one-to-one function $f : V \to \{1, \ldots, |V|\}$. If we regard $[1, |V|]$ as an interval on real numbers and consider $\alpha \in [1, |V|]$ then we call the number of edges $\{u, v\} \in E$ with $f(u) < \alpha$ and $f(v) > \alpha$ the value of the cut at α. The cutwidth of a layout f of G then is the maximum

of the value of the cut over all α. The CUTWIDTH problem for G is to find the minimum of the cutwidths of all possible layouts of G. The decision version of this problem is *NP*-complete [119]. In the parameterized version, for a given nonnegative integer k we ask whether a given graph has cutwidth $\leq k$. It is known that CUTWIDTH is fixed-parameter tractable [88] but the bounds on the exponential term seem to be huge. By way of contrast, BANDWIDTH, is $W[t]$-hard for all t [88] and thus appears to be fixed-parameter intractable. The bandwidth of a layout f of G is the maximum of $|f(u) - f(v)|$ over all edges $\{u, v\} \in E$. The bandwidth of G then is the minimum bandwidth of all possible layouts of G. The decision version of this problem is *NP*-complete [119]. Again, the parameterized version asks, given a graph $G = (V, E)$ and a nonnegative integer k, does G have bandwidth $\leq k$? Despite of its great practical importance, bandwidth seems to be a problem where parameterized complexity studies cannot help in general. A recent survey paper [102] sketches various approaches how to cope with the hardness of bandwidth, rising many open questions also concerning the development of exact algorithms.

Graph Parameters and Graph Classes

Many in general hard graph problems can efficiently be solved when restricted to special graph classes. For instance, this holds for problems such as VERTEX COVER or DOMINATING SET when restricted to graphs of bounded treewidth (cf. Sections 4.4 and 4.5). Two other graph parameters (and, thus, consider the graph classes implied in this way) are the *crossing number* and the *genus* of a graph. Both these parameters deal with the "degree of non-planarity" of a graph. The crossing number measures how many edge crossings are needed to draw a graph in the plane. To determine the (minimum) crossing number of a graph is *NP*-complete [120]. Recently, Grohe [139] has shown that this problem is fixed-parameter tractable—the underlying algorithm, however, seems to be impractical. By way of contrast, Mohar [191] gave a fixed-parameter algorithm for determining the genus of a graph. Again, this algorithm seems to be impractical. Besides asking for improvements of these algorithms, it is also important to investigate whether or to what extent fixed-parameter results achieved for planar graphs can be transferred to those more general graph classes. First positive news in this direction, referring to graphs of bounded genus, are reported in [97, 115]. Further special graph classes extending the concept of planarity in a more general way are that of *disk graphs* [66] (cf. [2, 9] for a recent result in this direction) and *map graphs* as introduced in the context of geographic information systems [61, 62]; for map graphs obviously no fixed-parameter studies have been undertaken so far. Refer to [44] for a general survey on graph classes.

Hypergraph Problems

The 3-Hitting Set problem as we have studied in Sections 2.3 and 3.4 is the only hypergraph problem we have considered so far. Simply speaking, hypergraphs mean a generalization of the graph concept such that an edge may consist of more than two vertices. In the 3-Hitting Set problem, for instance, we had hypergraphs where edges had up to three vertices. Hypergraph problems have many applications in fields such as constraint satisfaction, data bases, model checking, or artificial intelligence and non-monotonic reasoning [96, 126, 127]. One key open question in this context refers to the concept of *hypertree decompositions* [125]. Besides the "right" definition of this concept there are many open questions concerning parameterized complexity, e.g., what the fixed-parameter complexity of recognizing hypergraphs of bounded treewidth is [125]. Clearly, most parameterized problems studied in the graph context also make sense in the hypergraph context. So far, little has been explored here.

Again, much more could be reported concerning the fixed-parameter complexity of graph and network problems—may it be with respect to concrete problems such as the Directed Steiner Network problem [103] or the Max Leaf Spanning Tree problem [108] or may it be the still challenging step from abstract graph problems (together with corresponding fixed-parameter results) to real life applications and their special features including implementations and experiments.

5.3 Concluding Discussion

With few exceptions, the parameterized problems studied in this work either were drawn from computational biology or they were related to graphs and networks. But the quest for fixed-parameter solutions adheres to all fields of computation where hard problems have to be attacked. Hence, not surprisingly, there are many more fixed-parameter results in a great variety of application areas; subsequently, we sketch two of these in a little more detail. As a rule of thumb, however, one might say that nearly always when the investigation of approximation algorithms makes sense then also fixed-parameter algorithms are something to ask for. In this way, the list of problems (with approximation results) given in [23, 74] can be regarded as a good source of problems to be seen through parameterized glasses. Two more problem fields briefly discussed in the following are propositional logic and databases.

Propositional Logic

We have already taken a closer look at the Maximum Satisfiability problem (see Sections 2.2 and 3.5). Another, more special example of a problem

drawn from logic with a special kind of parameterization is the Falsifiability Problem for Pure Implicational Formulas. The complexity of this problem was first studied by Heusch [150]. A Boolean formula is in pure implicational form if it contains only positive literals and the only logical connective being used is the implication. Heusch considered the special case when all variables except at most one (denoted z) occur at most twice. This problem still is *NP*-complete [150]. However, he proved that if the number of occurrences of z is restricted to be at most k then there is an $O(|F|^k)$ time algorithm for certifying falsifiability. Franco *et al.* [117] subsequently showed how to solve the Falsifiability Problem for Pure Implicational Formulas in time $O(k^k n^2)$; thus, the problem is fixed-parameter tractable.

Databases

Downey *et al.* [93] and Yannakakis [265] initiated the consideration of database problems in the parameterized context. Papadimitriou and Yannakakis [216] analyzed the complexity of database queries. Here, the basic observation is that the size of the queries is typically orders of magnitude smaller than the size of the database. They analyze the complexity of the queries (e.g., conjunctive queries, first-order, Datalog, fixpoint logic etc.) with respect to two types of parameters: the query size itself and the number of variables that appear in the query. In this setting, they classify the relational calculus and its fragments at various levels of the W-hierarchy, hence showing parameterized intractability. On the positive side, they show that the extension of acyclic queries with inequalities is fixed-parameter tractable (refer to [216] for details). In the last few years, however, progress has been made concerning the fixed-parameter tractability for various restricted classes of database instances—refer to Grohe's survey paper [141] for more on this. Finally, observe that there also are close connections to *model checking* as described in [112, 118, 140].

We conclude this chapter with an enumeration of several more application areas with ongoing fixed-parameter research, pointing to some of the literature that may serve as starting points for further investigations (also cf. [104] for some recent survey):

- graph drawing [94, 95];
- automata theory [260];
- type checking in logic programs [56, 182];
- artificial intelligence [127, 196];
- routing in networks [14, 242];
- scheduling [187].

6. Further Topics and Future Challenges

This work left several topics of parameterized complexity more or less untouched—some of them consciously and some of them probably due to the lack of insight. Here, we want to provide a brief (and undoubtedly incomplete) survey on current and future issues in relation with fixed-parameter complexity that go beyond the contents of this text.

6.1 Implementation and Experiments

The ultimate goal of algorithmic research is to see the developed algorithms implemented and applied. The design and analysis of fixed-parameter algorithms is a relatively new field and today only few empirical evaluations of fixed-parameter algorithms are available. And still, even given these first experimental investigations as in the case of Vertex Cover [4, 79, 234] it is still open whether these implementations will turn out to be useful in "real practice" where problems to be solved normally lack compact mathematical formalizations but carry several constraints and side conditions to be taken into account. Fixed-parameter algorithmics is at the very beginning here.

Well, one might argue that it is a common attitude in theoretical computer science to assume that the real work is done when the algorithm is proven correct and the running time is analyzed mathematically. Just consider the vast number of results concerning approximation algorithms for hard problems. It is hard to give any numbers but it seems more than obvious that the great majority of algorithms never made it into an implementation. Nevertheless this research plays a major role in theoretical computer science and beyond. Hence, to some extent this also applies to fixed-parameter algorithms and so let us allow the purely theoretical game also here. Of course, this is not completely satisfactory by obvious reasons but for fixed-parameter complexity analysis there is still more on that.

Why are implementation and experiments of particular importance with respect to fixed-parameter tractability? There are several aspects that have to be taken into account, some of them enumerated in the following.

1. In the definition of fixed-parameter tractability the growth of the function "f" depending on the parameter may be completely arbitrary, per-

haps already making the considered algorithm impractical already for tiny parameter values.
2. Fixed-parameter complexity means worst-case analysis and many fixed-parameter algorithms can be much faster in average than they are according to the worst-case analysis.
3. Many fixed-parameter algorithms are suitable for combination with heuristic approaches and the practical benefits from this can only be determined by experimental investigations.
4. Very often, more than one parameterization of a problem exists and one may turn out to be better than the other because of the respective parameter values occurring in practice.
5. Ideas for new, alternative parameterizations of a problem often go in hand with practical experiences.
6. Exponential time algorithms such as those based on tree decompositions of graphs additionally may suffer from the need for a large amount of memory. Thus, space may become the bottleneck instead of time and often the mutual tradeoff has to be investigated empirically.

In summary, there are many good reasons to experiment with fixed-parameter algorithms. And, indeed, this is a growing area. In particular, fixed-parameter algorithms derived for problems from computational biology have been tested in practice, cf., e.g., [36, 133, 135, 226, 254]. Generally speaking, however, there are some difficulties that have to be overcome and which do not only apply to testing fixed-parameter algorithms. The most important ones are that it usually means hard work to get realistic sets of data (synthetically generated and, more importantly, from real-world instances) and to experiment with them.[1] Moreover, as a rule of thumb, the theoretical fixed-parameter algorithms additionally need quite some algorithm engineering to show their full performance. All this needs a lot of (wo)manpower and the community of people developing fixed-parameter algorithms still seems rather small when compared to other communities. But things start to get better here and the future prospects appear to be promising.

6.2 Heuristics, Approximation, and Parallelization

In this section, we deal with three theoretically and/or practically important ways in order to get fast solutions of computationally hard problems. All of them are related to fixed-parameter algorithms.

Heuristics

In the previous section it already was indicated that heuristics may help to significantly speed up fixed-parameter algorithms. As a rule, these are special

[1] Starting a publically accessible collection of data sets for core parameterized problems would be a very useful contribution waiting for its realization.

kinds of heuristics which are designed to lower the running time without destroying the optimality of the solution. Since heuristic approaches are of high importance in the practice of computing it is worth pursuing further links between parameterized complexity and heuristics. Specifically, considering fixed-parameter algorithms as particular heuristics (namely of a kind with reliable statements concerning the quality of the solution and guaranteed worst-case upper bounds on the running time) may deliver a new way to better understand in at least some cases why many heuristic approaches work well in practice although they deal with *NP*-hard problems. Moreover, there are so many facets and methodologies all covered by the word "heuristics" such that we are at the starting point of an investigation concerning the mutual links and possible stimulations of both fields. We expect that topics such as local search or evolutionary and memetic algorithms will become new subjects of fixed-parameter studies; [90, 196] contain first investigations in this direction.

Approximation

As the major theory-based tool of attacking intractability so far are approximation algorithms,[2] it is not surprising that some tight connections between approximation and fixed-parameter complexity have already been detected. For instance, it is fairly straightforward to prove that if an optimization problem possesses a fully polynomial time approximation scheme (FPTAS) then its natural parameterized counterpart (where the value to be optimized is turned into the parameter as, e.g., in the case of Vertex Cover) is fixed-parameter tractable [51]. Observe that this result implies that if a parameterized problem is shown to be $W[1]$-hard then its natural optimization counterpart does not have an FPTAS unless $FPT = W[1]$. Also, all parameterized variants of the optimization problems in the maximization class *MaxSNP* (and also another minimization class) are fixed-parameter tractable, see [51] for details.

Currently, the following two topics in the intersection between approximation and fixed-parameter complexity deserve particular attention. Firstly, there is a strong interest on what parameterized complexity can say about the practical feasibility of polynomial time approximation schemes (PTAS). An important contribution to this was made by Cesati and Trevisan [57]. They distinguished between a PTAS (where the degree of the polynomial running time may depend on the approximation ratio) and an *efficient* PTAS (where the degree of the polynomial running time may *not* depend on the approximation ratio but—contrary to the FPTAS case—may only contribute an exponentially (or worse) growing *factor* to the running time. Cesati and Trevisan related the existence of the (practically more relevant) efficient PTAS

[2] Of course, also approximation algorithms may serve as special kinds of heuristics with proven quality bounds.

to the concept of $W[1]$-hardness in the sense that $W[1]$-hardness for the corresponding parameterized version of an optimization problem excludes an efficient PTAS for that optimization problem unless $FPT = W[1]$ (see [57] for details). Cai *et al.* [52] recently continued and extended these studies, examining upper and lower bounds on efficient PTAS's for a variety of problems contained in syntactically defined approximation complexity classes. Specifically, they state concrete lower bounds for the asymptotically best running times achievable for efficient PTAS's for several problems on planar graphs based on the hypothesis that $FPT \neq W[1]$.

The second point deserving more attention is a little more vague but appears to be of big practical importance. In Chapter 2, we emphasized the great significance of data reduction by efficient preprocessing. Small problem kernels may be considered as key achievements of fixed-parameter complexity and decisively contribute to the practicability of the whole methodology. Clearly among the best kernelizations known so far is that due to Nemhauser and Trotter (see Theorem 2.4.1 and [199]) providing a size $2k$ problem kernel for Vertex Cover. Interestingly, this results seems optimal in the sense that a problem kernel of size $(2 - \epsilon)k$ for constant $\epsilon > 0$ would mean a polynomial-time approximation algorithm for Vertex Cover with approximation ratio better than $1/2$. It is a longstanding open problem [23, 152], however, whether such an approximation algorithm for Vertex Cover exists. By way of contrast, it is hoped that a corresponding lower bound can be proven which, of course, would also show the optimality of the Nemhauser-Trotter kernelization. In case of restricting Vertex Cover to planar graphs (where a PTAS for Vertex Cover is known [24]), however, there is still hope for a problem kernel of size less than $2k$. Maybe again results developed in the approximation context can help here. More generally, the future challenge is to provide and exploit stronger links between approximation and fixed-parameter complexity in order to develop new results for problem kernels.

Parallelization

Parallel algorithms are, strictly speaking, not really an answer to the computational intractability of *NP*-hard problems. The point is that, at the current state of the art where usually say hundreds or thousands of processors make a parallel machine, a speedup of at most the same dimensions can be expected—strictly speaking, however, this "only" means a constant factor speedup. As a consequence, dealing with *NP*-hard problems where we encounter exponential growth of the running time it is first of all more important to get the involved combinatorial explosion as small as possible. This is the main theme of this work. There may come, however, the point where further shrinking the size of the combinatorial explosion seems, in spite of serious efforts, hopeless. Then, it could still matter whether one has to wait say two weeks or few hours for a desired solution—and here parallelization

comes into play. Fixed-parameter algorithms based on bounded search trees are easy to parallelize because of the inherent partitioning into subtasks given by a search tree. Moreover, little communication is necessary because of the independence of the tasks. Dehne *et al.* [79] provided stimulating work in this direction, giving a parallelization for a VERTEX COVER search tree algorithm. It is a matter of future research to investigate how good other parameterized problems and fixed-parameter techniques such as reduction to problem kernel or dynamic programming parallelize.

6.3 Zukunftsmusik

The good prospects for fixed-parameter algorithms as predicted in the 1998 survey [202] have become reality. Fixed-parameter tractability will surely continue to prosper in various ways. The emphasis of this work was on algorithms. The main part of the monograph [88] deals with more structural and complexity-theoretical questions. Also in this respect there is ongoing fruitful research. Let us only stress two things here—the recent development of parameterized counting complexity classes [113, 188] or the investigation of complexity classes between FPT and $W[1]$ [105] as an effort to capture parameterized problems whose complexity could not have been classified. Further important topics here are the whole area of lower bounds [53] or alternative characterizations of parameterized complexity classes [114, 138].

We avoid listing concrete algorithmic challenges at this point since numerous of them are spread (many of them implicitly) all over the text. And, clearly, to come up with completely new ones, a simple trick, for instance, is to have a somewhat closer look at the vast literature on approximation algorithms and to study the fixed-parameter complexity of one of the corresponding problems. As fixed-parameter algorithms are still under-represented in the literature, this seems an easy thing to do. Another point that has been neglected so far is that fixed-parameter tractability may, in some cases, also be an alternative to polynomial time solvability. Imagine one only has an algorithm with a high-degree polynomial running time for some problem. A fixed-parameter algorithm for that problem with a perhaps linear time component in the overall input size and an exponential time component exclusively depending on a small parameter (so-called "linear fixed-parameter tractability") still might be beneficial in this case.

Finally, two things that should further accelerate the maturing process of fixed-parameter complexity into a well-established research field (with the corresponding publicity it deserves) and which are worth striving for are

- a collection of fixed-parameter results (perhaps in the style of Crescenzi and Kann's webpage [74]) and
- a publically available source of well-documented implementations and sound data for testing (new) fixed-parameter algorithms.

All this does not come for free and it will need many people to join the fixed-parameter track. Be invited!

References

1. → 119
   R. Agarwala and D. Fernández-Baca. A polynomial-time algorithm for the perfect phylogeny problem when the number of character states is fixed. *SIAM Journal on Computing*, 23(6):1216–1224, 1994.
2. → 34, 39, 68, 69, 70, 91, 93, 94, 95, 99, 100, 101, 131, 132, 133
   J. Alber. *Exact Algorithms for NP-hard Problems on Planar and Related Graphs: Design, Analysis, and Implementation*. PhD thesis in preparation, Universität Tübingen, Germany. 2002.
3. → 14, 22, 91, 93, 94, 100, 101
   J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks and R. Niedermeier. Fixed parameter algorithms for dominating set and related problems on planar graphs. *Algorithmica*, 33:461–493, 2002.
4. → 14, 16, 18, 34, 95, 137
   J. Alber, F. Dorn, and R. Niedermeier. Empirical Evaluation of a tree decomposition based algorithm for vertex cover on planar graphs. Manuscript, July 2002. Submitted to *Discrete Applied Mathematics*.
5. → 39, 67, 68, 69
   J. Alber, H. Fan, M. R. Fellows, H. Fernau, R. Niedermeier, F. Rosamond, and U. Stege. Refined search tree technique for dominating set on planar graphs. In *Proc. 26th MFCS*, Springer-Verlag LNCS 2136, pp. 111–122, 2001. Long version submitted to *Journal of Computer and System Sciences.*
6. → 34, 39
   J. Alber, M. R. Fellows, and R. Niedermeier. Efficient data reduction for dominating set: a linear problem kernel for the planar case. In *Proc. 8th SWAT*, Springer-Verlag LNCS 2368, pp. 150–159, 2002.
7. → 14, 40, 93, 94, 95, 99
   J. Alber, H. Fernau, and R. Niedermeier. Parameterized complexity: exponential speed-up for planar graph problems. In *Proc. 28th ICALP*, Springer-Verlag LNCS 2076, pp. 261–272, 2001. Long version available as Technical Report TR01-023, Electronic Colloquium on Computational Complexity (ECCC), Trier, March 2001.
8. → 14, 40, 130, 131, 132
   J. Alber, H. Fernau, and R. Niedermeier. Graph separators: a parameterized view. In *Proceedings 7th COCOON*, Springer-Verlag LNCS 2108, pp. 318–327, 2001. Very long version available as Technical Report WSI-2001-8, Wilhelm-Schickard-Institut für Informatik, Universität Tübingen. Long version accepted by *Journal of Computer and System Sciences.*
9. → 133
   J. Alber and J. Fiala. Geometric separation and exact solutions for the parameterized independent set problem on disk graphs. In *Proc. 17th IFIP World*

Computer Congress, 2nd IFIP International Conference on Theoretical Computer Science (TCS 2002), pp. 26–37, Kluwer Academic Publishers, 2002.

10. → 74, 118
J. Alber, J. Gramm, J. Guo, and R. Niedermeier. Towards optimally solving the longest common subsequence problem for sequences with nested arc annotations in linear time. In *Proc. 13th CPM* Springer-Verlag LNCS 2373, pp. 99–114, 2002.
11. → 1
J. Alber, J. Gramm, and R. Niedermeier Faster exact solutions for hard problems: a parameterized point of view. *Discrete Mathematics*, 229(1-3):3–27, 2001.
12. → 22, 100, 101
J. Alber and R. Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In *Proc. 5th LATIN*, Springer-Verlag LNCS 2286, pp. 613–627, 2002.
13. → 84, 86
N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42(4):844–856, 1995.
14. → 135
R. S. Anand, T. Erlebach, A. Hall, and S. Stefanakos. Call control with k rejections. In *Proc. 8th SWAT*, Springer-Verlag LNCS 2368, pp. 308–317, 2002.
15. → 13, 27
K. Appel and W. Haken. Every planar map is four colorable. I. Discharging. *Illinois J. Math.*, 21:429–490, 1977.
16. → 13, 27
K. Appel and W. Haken. Every planar map is four colorable. II. Reducibility. *Illinois J. Math.*, 21:491–567, 1977.
17. → 11, 88
S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8:277–284, 1987.
18. → 66
S. Arora and C. Lund. Hardness of approximations. In D. S. Hochbaum (ed.) *Approximation Algorithms for NP-Hard Problems*, pp. 399–446, PWS Publishing Company, 1997.
19. → 15
V. Arvind and V. Raman. Approximation algorithms for some parameterized counting problems. Technical Report TR02-031 (Revision 1), Electronic Colloquium on Computational Complexity (ECCC), Trier, Germany, June 2002.
20. → 66
T. Asano and D. P. Williamson. Improved approximation algorithms for MAX SAT. In *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 96–105, 2000.
21. → 101
B. Aspvall, A. Proskurowski, and J. A.Telle. Memory requirements for table computations in partial k-tree algorithms. *Algorithmica*, 27:382–394, 2000.
22. → 5
M. J. Atallah. *Algorithms and Theory of Computation Handbook.* CRC Press, 1999.
23. → 1, 5, 134, 140
G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi. *Complexity and Approximation.* Springer-Verlag, 1999.
24. → 13, 90, 140
B. S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *Journal of the ACM*, 41:153–180, 1994.

25. → 106
H.-J. Bandelt and A. Dress. Reconstructing the shape of a tree from observed dissimilarity data. *Advances in Applied Mathematics*, 7:309–343, 1986.
26. → 55
R. Battiti and M. Protasi. Reactive Search, a history-base heuristic for MAX-SAT. *ACM Journal of Experimental Algorithmics*, 2:Article 2, 1997.
27. → 54
R. Battiti and M. Protasi. Approximate algorithms and heuristics for MAX-SAT. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 1, pages 77–148. Kluwer Academic Publishers, 1998.
28. → 106
V. Berry and O. Gascuel. Inferring evolutionary trees with strong combinatorial evidence. *Theoretical Computer Science*, 240:271–298, 2000. Software available through `http://www.lirmm.fr/~vberry/PHYLOQUART/phyloquart.html`.
29. → 66, 73, 75
N. Bansal and V. Raman. Upper bounds for MaxSat: Further improved. In *Proc. 10th ISAAC*, Springer-Verlag LNCS 1741, pp. 247–258, 1999.
30. → 12, 75
R. Balasubramanian, M. R. Fellows, and V. Raman. An improved fixed parameter algorithm for vertex cover. *Information Processing Letters*, 65(3):163–168, 1998.
31. → 13
R. Bar-Yehuda and S. Even. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2:198–203, 1981.
32. → 12, 13, 27, 31, 33
R. Bar-Yehuda and S. Even. A local-ratio theorem for approximating the weighted vertex cover problem. *Annals of Discrete Mathematics*, 25:27–45, 1985.
33. → 105
A. Ben-Dor, B. Chor, D. Graur, R. Ophir, and D. Pelleg. Constructing phylogenies from quartets: elucidation of eutherian superordinal relationships. *Journal of Computational Biology*, 5:377–390, 1998.
34. → 80
A. Ben-Dor, G. Lancia, J. Perone, R. Ravi. Banishing bias from consensus sequences. In *Proc. 8th CPM*, Springer-Verlag LNCS 1264, pp. 247–261, 1997.
35. → 19, 105, 107
V. Berry, T. Jiang, P. Kearney, M. Li, and T. Wareham. Quartet cleaning: improved algorithms and simulations. In *Proc. 7th ESA*, Springer-Verlag LNCS 1643, pp. 313–324, 1999.
36. → 113, 138
M. Blanchette, B. Schwikowski, and M. Tompa. Algorithms for phylogenetic footprinting. *Journal of Computational Biology*, 9(2):211–224, 2002.
37. → 9
A. Blummer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Learnability and the Vapnik-Chervonenkis dimension. *Journal of the ACM*, 36:929–965, 1989.
38. → 11, 22, 88, 94
H. L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.
39. → 96
H. L. Bodlaender. Treewidth: Algorithmic techniques and results. In *Proc. 22nd MFCS*, Springer-Verlag LNCS 1295, pp. 19–36, 1997.

40. → 89, 91, 92, 131
H. L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209:1–45, 1998.
41. → 111, 117
H. L. Bodlaender, R. G. Downey, M. R. Fellows, M. T. Hallett, and H. T. Wareham. Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences*, 11:49–57, 1995.
42. → 117
H. L. Bodlaender, R. G. Downey, M. R. Fellows, and H. T. Wareham. The parameterized complexity of sequence alignment and consensus. *Theoretical Computer Science*, 147:31–54, 1995.
43. → 7
R. B. Boppana and M. Sipser. The complexity of finite functions. In J. van Leeuwen, editor, *Algorithms and Complexity*, volume A of *Handbook of Theoretical Computer Science*, chapter 14, pages 757–804. Elsevier, 1990.
44. → 14, 22, 89, 95, 133
A. Brandstädt, V. B. Le, and J. P. Spinrad. *Graph Classes: a Survey*. SIAM Monographs on Discrete Mathematics and Applications. Society for Industrial and Applied Mathematics, 1999.
45. → 111, 113
J. Buhler and M. Tompa. Finding motifs using random projections. *Journal of Computational Biology*, 9(2):225–242, 2002.
46. → 18, 108, 109
D. Bryant. The complexity of the breakpoint median problem. Technical report CRM-2579. Centre de Recherches Mathématiques, Université de Montréal, Canada. 1998.
47. → 106
P. Buneman. The recovery of trees from measures of dissimilarity. In Hodson *et al.*, editors, *Anglo-Romanian Conference on Mathematics in the Archaeological and Historical Sciences*, pages 387–395, 1971. Edinburgh University Press.
48. → 26
J. F. Buss and J. Goldsmith. Nondeterminism within P. *SIAM Journal on Computing*, 22(3):560–572, 1993.
49. → 25
M. Cadoli, F. M. Donini, P. Liberatore, and M. Schaerf. Preprocessing of intractable problems. *Information and Computation*, 176:89–120, 2002.
50. → 132
L. Cai. Fixed-parameter tractability of graph modification problems for hereditary properties. *Information Processing Letters*, 58:171–176, 1996.
51. → 66, 75, 139
L. Cai and J. Chen. On fixed-parameter tractability and approximability of NP optimization problems. *Journal of Computer and System Sciences*, 54:465–474, 1997.
52. → 140
L. Cai, M. Fellows, D. Juedes, and F. Rosamond. Efficient polynomial-time approximation schemes for problems on planar graph structures: upper and lower bounds. Unpublished Manuscript, May 2001.
53. → 16, 141
L. Cai and D. Juedes. On the existence of subexponential parameterized algorithms. Manuscript, submitted for publication. October 2001. Revised version of the paper "Subexponential parameterized algorithms collapse the W-hierarchy" in *Proceedings 28th ICALP*, Springer-Verlag LNCS 2076, pp. 273–284, 2001. This conference version contains some major flaws, cf. [91].

54. → 120
A. Caprara. Sorting permutations by reversals and eulerian cycle decompositions. *SIAM Journal on Discrete Mathematics*, 12:91–110, 1999.
55. → 108
A. Caprara. On the practical solution of the reversal median problem. In *Proc. 1st WABI*, Springer-Verlag LNCS 2149, pp. 238–251, 2001.
56. → 135
W. Charatonik. Directional type checking for logic programs: beyond discriminative types. In *Proc. ESOP'00*, Springer-Verlag LNCS 1782, pages 72–87, 2000.
57. → 112, 113, 139, 140
M. Cesati and L. Trevisan. On the efficiency of polynomial time approximation schemes. *Information Processing Letters*, 64(4):165–171, 1997.
58. → 15, 40, 73, 128, 129
J. Chen and I. A. Kanj. On constrained minimum vertex covers of bipartite graphs: improved algorithms. In *Proc. 27th WG*, Springer-Verlag LNCS 2204, pp. 55-65, 2001.
59. → 66, 67, 73, 75
J. Chen and I. A. Kanj. Improved exact algorithms for MAX-SAT. In *Proc. 5th LATIN*, Springer-Verlag LNCS 2286, pp. 341–355, 2002.
60. → 2, 6, 10, 12, 16, 27, 31, 33, 34, 44, 73, 74, 75, 81, 82, 99
J. Chen, I. A. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.
61. → 133
Z.-Z. Chen. Approximation algorithms for independent sets in map graphs. *Journal of Algorithms,* 41(1):20–40, 2001.
62. → 133
Z.-Z. Chen, M. Grigni, and C. H. Papadimitriou. Planar map graphs. *Journal of the ACM*, 49(2):127–138, 2002.
63. → 105, 106
B. Chor. From quartets to phylogenetic trees. In *Proc. 25th SOFSEM*, Springer-Verlag LNCS 1521, pp. 36–53, 1998.
64. → 107
B. Chor. Personal communication. August 2001.
65. → 19, 22
B. Chor and M. Sudan. A geometric approach to betweeness. *SIAM Journal on Discrete Mathematics*, 11(4): 511–523, 1998.
66. → 133
B. N. Clark, C. J. Colbourn, and D. S. Johnson. Unit disk graphs. *Discrete Mathematics*, 86:165–177, 1990.
67. → 106
H. Colonius, H. H. Schultze. Tree structure for proximity data. *British Journal of Mathematical and Statistical Psychology*, 34:167–180, 1981.
68. → 6
D. Coppersmith and S. Winograd. Matrix multiplication via arithmetical progression. *J. Symbolic Computations*, 9:251–280, 1990.
69. → 2, 33
T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms* (Second Edition). The MIT Press, 2001.
70. → 110
M. E. Cosner, R. K. Jansen, B. M. E. Moret, L. A. Raubeson, L. S. Wang, T. Warnow, and S. Wyman. An empirical comparison of phylogenetic methods

on chloroplast gene or der data in Campanulaceae. In D. Sankoff and J. Nadeau (eds.): *Comparative Genomics*, pp. 99–121. Kluwer, 2000.

71. → 110
M. E. Cosner, R. K. Jansen, B. M. E. Moret, L. A. Raubeson, L. S. Wang, T. Warnow, and S. Wyman. A new fast heuristic for computing the breakpoint phylogeny and experimental phylogenetic analyses of real and synthetic data. In *Proc. 8th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pp. 104–115, AAAI Press, 2000.
72. → 20
D. Cox, M. Burmeister, E. Price, S. Kim, and M. Myers. Radiation hybrid mapping: a somatic cell genetic method for constructing high-resolution maps of mammalian chromosomes. *Science*, 250:245–250, 1990.
73. → 55
P. Crescenzi and V. Kann. How to find the best approximation results—a follow-up to Garey and Johnson. *ACM SIGACT News*, 29(4):90–97, 1998.
74. → 134, 141
P. Crescenzi and V. Kann. A compendium of NP optimization problems. Available at http://www.nada.kth.se/theory/problemlist.html, August 1998.
75. → 66, 74
E. Y. Dantsin, M. R. Gavrilovich, E. A. Hirsch, and B. Y. Konev. Approximation algorithms for MAX SAT: a better performance ratio at the cost of a longer running time. *Annals of Pure and Applied Logic*, 113(1–3), pp. 81–94, 2001.
76. → 16
E. Y. Dantsin, A. Goerdt, E. A. Hirsch, and R. Kannan, J. Kleinberg, C. H. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2 - 2/(k + 1))^n$ *algorithm* for k-SAT based on local search. To appear in *Theoretical Computer Science*, 2002.
77. → 121
B. DasGupta, T. Jiang, S. Kannan, M. Li, and E. Sweedyk. On the complexity and approximation of syntenic distance. *Discrete Applied Mathematics*, 88:59–82, 1998.
78. → 56
M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
79. → 16, 74, 137, 141
F. Dehne, A. Rau-Chaplin, U. Stege, and P. J. Taillon. Solving large FPT problems on coarse grained parallel machines. Manuscript, July 2001.
80. → 105
G. Della Vedova, T. Jiang, J. Li, and J. Wen. Approximating minimum quartet inconsistency (abstract). In *13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 894–895, 2002.
81. → 14, 94
E. D. Demaine, M. Hajiaghayi, and D. T. Thilikos. Exponential speedup of fixed-parameter algorithms on $K_{3,3}$-minor-free or K_5-minor-free graphs. To appear in *Proc. 13th ISAAC*, Springer-Verlag LNCS, Vancouver, Canada, December 2002.
82. → 125
R. Diestel. *Graph Theory*. Springer-Verlag, 1997.
83. → 131
H. N. Djidjev. On the problem of partitioning planar graphs. *SIAM J. Algebraic Discrete Methods*, 3(2):229–240, 1982.

84. → 131
H. N. Djidjev and S. M. Venkatesan. Reduced constants for simple cycle graph separation. *Acta Informatica*, 34:231–243, 1997.
85. → 16, 34
F. Dorn. *Tuning Algorithms for Hard Graph Problems*. Study Work, Universität Tübingen, Germany, in preparation, 2002.
86. → 9
R. G. Downey, P. Evans, and M. R. Fellows. Parameterized learning complexity. In *6th Annual Conference on Learning Theory (COLT)*, pp. 51–57. ACM Press, 1993.
87. → 9, 70
R. G. Downey and M. R. Fellows. Parameterized computational feasibility. In *P. Clote, J. Remmel (eds.): Feasible Mathematics II*, pp. 219–244. Birkhäuser, 1995.
88. → 1, 2, 6, 7, 8, 9, 10, 11, 12, 26, 42, 54, 70, 77, 87, 102, 113, 117, 119, 133, 141
R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.
89. → 1
R. G. Downey and M. R. Fellows. Parameterized complexity after (almost) ten years: Review and open questions. In *Proc. Combinatorics, Computation, and Logic (DMTCS'99 and CATS'99)*, Australian Computer Science Communications, Volume 21 Number 3, pp. 1–33. Springer-Verlag Singapore, 1999.
90. → 139
R. G. Downey, M. R. Fellows, and C. McCartin. Parameterized local search. Manuscript, June 2000.
91. → 146
R. G. Downey, M. R. Fellows, R. Niedermeier, and P. Rossmanith (eds.). Parameterized complexity. Dagstuhl-Seminar-Report No. 316, August 2001.
92. → 1, 12, 54, 71, 75
R. G. Downey, M. R. Fellows, and U. Stege. Parameterized complexity: A framework for systematically confronting computational intractability. In *Contemporary Trends in Discrete Mathematics: From DIMACS and DIMATIA to the Future*, volume 49 of *AMS-DIMACS*, pp. 49–99. AMS, 1999.
93. → 135
R. G. Downey, M. R. Fellows, and U. Taylor. The parameterized complexity of relational database queries and an improved characterization of W[1]. In *Proc. Combinatorics, Complexity, and Logic (DMTCS'96)*, pp. 194–213, 1996.
94. → 135
V. Dujmovic, M. R. Fellows, M. T. Hallett, M. Kitching, G. Liotta, C. McCartin, N. Nishimura, P. Ragde, F. A. Rosamond, M. Suderman, S. Whitesides, D. R. Wood. On the parameterized complexity of layered graph drawing. In *Proc. 9th ESA*, Springer-Verlag LNCS 2161, pp. 488-499, 2001.
95. → 135
V. Dujmovic, M. R. Fellows, M. T. Hallett, M. Kitching, G. Liotta, C. McCartin, N. Nishimura, P. Ragde, F. A. Rosamond, M. Suderman, S. Whitesides, D. R. Wood. A fixed-parameter approach to two-layer planarization. In *Proc. 9th GD*, Springer-Verlag LNCS 2265, pp. 1–15, 2001.
96. → 15, 134
T. Eiter and G. Gottlob. Identifying the minimal transversals of a hypergraph and related problems. *SIAM Journal on Computing*, 24(6):1278–1304, 1995.
97. → 70, 133

J. Ellis, H. Fan, and M.R. Fellows. The dominating set problem is fixed parameter tractable for graphs of bounded genus. In *Proc. 8th SWAT*, Springer-Verlag LNCS 2368, pp. 180–189, 2002.
98. → 117, 118
P. A. Evans. *Algorithms and Complexity for Annotated Sequence Analysis*. PhD thesis, University of Victoria, Canada. 1999.
99. → 117
P. A. Evans. Finding common subsequences with arcs and pseudoknots. In *Proc. 10th CPM*, Springer-Verlag LNCS 1645, pp. 270–280, 1999.
100. → 21, 45, 113
P. A. Evans and H. T. Wareham. Practical non-polynomial time algorithms for designing universal DNA oligonucleotides: a systematic approach. Manuscript, April 2001.
101. → 130
S. S. Fedin and A. S. Kulikov. A $2^{|E|/4}$-time algorithm for MAX-CUT. Manuscript, St. Petersburg State University, May 2002.
102. → 133
U. Feige. Coping with the *NP*-hardness of the graph bandwidth problem. In *Proc. 7th SWAT*, Springer-Verlag LNCS 1851, pp. 10–19, 2000.
103. → 134
J. Feldman and M. Ruhl. The directed Steiner network problem is tractable for a constant number of terminals. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 299–308, 1999.
104. → 1, 135
M. R. Fellows. Parameterized complexity: the main ideas and some research frontiers. In *Proc. 12th ISAAC*, Springer-Verlag LNCS 2223, pp. 441–453, 2001.
105. → 141
M. R. Fellows. Personal communication. June 2002.
106. → 108, 113, 114, 115
M. R. Fellows, J. Gramm, and R. Niedermeier. Parameterized intractability of closest substring. In *Proc. 19th STACS*, Springer-Verlag LNCS 2285, pp. 262–273, 2002.
107. → 120
M. R. Fellows, M. T. Hallett, and U. Stege. On the multiple gene duplication problem. In Proc. 9th ISAAC, Springer-Verlag LNCS 1533, pp. 347–356, 1998.
108. → 40, 73, 134
M. R. Fellows, C. McCartin, F. A. Rosamond, and U. Stege. Coordinatized kernels and catalytic reductions: an improved FPT algorithm for max leaf spanning tree and other problems. In *Proc. 20th FSTTCS*, Springer-Verlag LNCS 1974, pp. 111–122, 2000.
109. → 120
V. Feretti, J. H. Nadeau, and D. Sankoff. Original synteny. In *Proc. 7th CPM*, Springer-Verlag LNCS 1075, pp. 159–167, 1996.
110. → 15
H. Fernau. Parameterized enumeration. In *Proc. 8th COCOON*, Springer-Verlag LNCS 2387, pp. 564–573, 2002.
111. → 15, 21, 40, 73, 74, 125, 127, 128
H. Fernau and R. Niedermeier. An efficient exact algorithm for constraint bipartite vertex cover. *Journal of Algorithms*, 38(2): 374–410, 2001.
112. → 9, 135
J. Flum and M. Grohe. Fixed-parameter tractability, definability, and model checking. *SIAM Journal on Computing* 31: 113-145, 2001.

113. → 15, 141
J. Flum and M. Grohe. The parameterized complexity of counting problems. To appear in *Proc. 43rd IEEE Symposium on Foundations of Comupter Science (FOCS)*, 2002.
114. → 9, 141
J. Flum and M. Grohe. Describing parameterized complexity classes. In *Proc. 19th STACS*, Springer-Verlag LNCS 2285, pp. 359–371, 2002.
115. → 14, 94, 95, 133
F. V. Fomin and D. T. Thilikos. Dominating sets in planar graphs: branch-width and exponential speed-up. Technical report, Departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Barcelona, Spain, 2002.
116. → 112
M. Frances and A. Litman. On covering problems of codes. *Theory of Computing Systems*, 30:113–119, 1997.
117. → 135
J. Franco, J. Goldsmith, J. Schlipf, E. Speckenmeyer, and R. P. Swaminathan. An algorithm for the class of pure implicational formulas. *Discrete Applied Mathematics*, 96-97:89–106, 1999.
118. → 135
M. Frick. *Easy Instances for Model-Checking.* PhD thesis, Universität Freiburg, Germany. 2001.
119. → 5, 13, 133
M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979.
120. → 133
M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM J. Algebraic Discrete Methods*, 4: 312–316, 1983.
121. → 119
L. A. Goldberg, P. W. Goldberg, C. A. Phillips, E. Sweedyk, and T. Warnow. Minimizing phylogenetic number to find good evolutionary trees. *Discrete Applied Mathematics*, 71(1–3):111–136, 1996.
122. → 15
M. K. Goldberg, T. H. Spencer, and D. A. Berque. A low-exponential algorithm for counting vertex covers. *Graph Theory, Combinatorics, Algorithms, and Applications*, 1:431–444, 1995.
123. → 117
D. Goldman, S. Istrail, and C. H. Papadimitriou. Algorithmic aspects of protein structure similarity. In *Proc. 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 512–521, 1999.
124. → 20
S. Goss and H. Harris. New method for mapping genes in human chromosomes. *Nature*, 255:680–684, 1975.
125. → 134
G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions: a survey. In *Proc. 26th MFCS*, Springer-Verlag LNCS 2136, pp. 37–57, 2001.
126. → 134
G. Gottlob and R. Pichler. Hypergraphs in model checking: acyclicity and hypertree-width versus clique-width. In *Proc. 28th ICALP*, Springer-Verlag LNCS 2076, pp. 708–719, 2001.
127. → 134, 135
G. Gottlob, F. Scarcello, and M. Sideri. Fixed-parameter complexity in AI and nonmonotonic reasoning. *Artificial Intelligence*, 138(1–2):55–86, 2002.

128. → 66
J. Gramm. *Exact Algorithms for Max2Sat and Their Applications.* Diploma thesis, Universität Tübingen, Germany, October 1999.
129. → 106, 107, 108, 109, 110, 111, 112, 113, 114, 115
J. Gramm. *Fixed-Parameter Algorithms for Consensus Analysis of Genomic Data*. PhD thesis in preparation, Universität Tübingen, Germany. 2002.
130. → 119
J. Gramm, J. Guo, and R. Niedermeier. Pattern-matching in arc-annotated sequences. To appear in *Proc. 22nd FSTTCS*, Springer-Verlag LNCS, Kanpur, India, December 2002.
131. → 67, 129, 130
J. Gramm, E. A. Hirsch, R. Niedermeier, and P. Rossmanith. New worst-case upper bounds for MAX-2-SAT with application to MAX-CUT. To appear in *Discrete Applied Mathematics*, 2002. Preliminary version appears as ECCC Technical Report TR00-037, Trier, Germany.
132. → 66, 67
J. Gramm and R. Niedermeier. Faster exact solutions for Max-2-Sat. In *Proc. 4th CIAC,* Springer-Verlag LNCS 1767, pp. 174–186, 2000.
133. → 19, 40, 74, 104, 106, 107, 138
J. Gramm and R. Niedermeier. Minimum quartet inconsistency is fixed parameter tractable In *Proc. 12th CPM*, Springer-Verlag LNCS 2089, pp. 241–256, 2001. Long version accepted by *Journal of Computer and System Sciences*.
134. → 107
J. Gramm and R. Niedermeier. Evaluating an algorithm for parameterized minimum quartet inconsistency. In N. El-Mabrouk, T. Lengauer, and D. Sankoff (eds.), *Currents in Computational Molecular Biology 2001*, pp. 195–196. Les Publications CRM, Montréal, 2001.
135. → 18, 40, 104, 107, 109, 110, 111, 138
J. Gramm and R. Niedermeier. Breakpoint medians and breakpoint phylogenies: a fixed-parameter approach. To appear in *Proc. First European Conference on Computational Biology*, October 2002, Saarbrücken. Supplement to *Bioinformatics*, Oxford University Press.
136. → 21, 117
J. Gramm, F. Hüffner, and R. Niedermeier. Closest strings, primer design, and motif search. In L. Florea *et al.* (eds.), *Currents in Computational Molecular Biology 2002*, poster abstracts of *RECOMB 2002*, pp. 74–75, 2002.
137. → 20, 21, 45, 80, 112, 113
J. Gramm, R. Niedermeier, and P. Rossmanith. Exact solutions for closest string and related problems. In *Proc. 12th ISAAC*, Springer-Verlag LNCS 2223, pp. 441–453, 2001.
138. → 9, 141
M. Grohe. Descriptive and parameterized complexity. In *13th CSL*, Springer-Verlag LNCS 1683, pp. 14–31, 1999.
139. → 133
M. Grohe. Computing crossing numbers in quadratic time. In *Proc. 33rd ACM Symposium on Theory of Computing (STOC)*, pp. 231–236, 2001.
140. → 135
M. Grohe. Generalized model-checking problems for first-order logic. In *Proc. 18th STACS*, Springer-Verlag LNCS 2010, pp. 12–26, 2001.
141. → 135
M. Grohe. The parameterized complexity of database queries. In *Proc. 20th ACM Symposium on Principles of Database Systems (PODS)*, pp. 82–92, 2001.

142. → 14
S. Guha, R. Hassin, S. Khuller, and E. Or. Capacitated vertex covering with applications. In *13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 858–865, 2002.
143. → 118
J. Guo. *Exact Algorithms for the Longest Common Subsequence Problem for Arc-Annotated Sequences*. Diploma thesis, Universität Tübingen, Germany. February 2002.
144. → 104
D. Gusfield. *Algorithms on Strings, Trees, and Sequences (Computer Science and Computational Biology)*. Cambridge University Press, 1997.
145. → 70
T. Hagerup. Personal communication. July 2002.
146. → 120
S. Hannenhalli and P. Pevzner. To cut ... or not to cut (applications of comparative physical maps in molecular evolution). In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 304–313, 1996.
147. → 54
P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.
148. → 2
J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation,* 2. ed., Addison-Wesley, 2001.
149. → 92
T. W. Haynes, S. T. Hedetniemi, and P. J. Slater. *Fundamentals of Domination in Graphs*. Monographs and Textbooks in Pure and Applied Mathematics Vol. 208, Marcel Dekker, 1998.
150. → 135
P. Heusch. The complexity of the falsifiability problem for pure implicational formulas. *Discrete Applied Mathematics*, 96-97:127–138, 1999.
151. → 108, 111
C. De La Higuera and F. Casacuberta. Topology of strings: median string is NP-complete. *Theoretical Computer Science*, 230(1/2): 39–48, 2000.
152. → 1, 5, 33, 140
D. S. Hochbaum (ed.). *Approximation algorithms for NP-hard problems*. Boston, MA: PWS Publishing Company, 1997.
153. → 1
M. Hofri. *Analysis of algorithms: computational methods and mathematical tools*. Oxford University Press, 1995.
154. → 2, 5
J. Hromkovič. *Algorithmics for Hard Problems (Introduction to Combinatorial Optimization, Randomization, Approximation, and Heuristics)*. Springer-Verlag, 2001.
155. → 104
T. Jiang (ed.). *Current Topics in Computational Molecular Biology*. The MIT Press, 2002.
156. → 105
T. Jiang, P. Kearney, and M. Li. Some open problems in computational molecular biology. *Journal of Algorithms*, 34:194–201, 2000.
157. → 19, 105
T. Jiang, P. Kearney, and M. Li. A polynomial time approximation scheme for inferring evolutionary trees from quartet topologies and its application. *SIAM Journal on Computing*, 30(6):1942-1961, 2001.

158. → 117, 118
T. Jiang, G.-H. Lin, B. Ma, and K. Zhang. The longest common subsequence problem for arc-annotated sequences. In *Proc. 11th CPM*, Springer-Verlag LNCS 1848, pp. 154–165, 2000. Full paper accepted by *Journal of Discrete Algorithms*.
159. → 54
D. S. Johnson and M. A. Trick, editors. *Cliques, Coloring and Satisfiability, Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Ser. Discr. Math. Theor. Comput. Sci.* AMS, 1996.
160. → 94
I. A. Kanj and L. Perkovic. Improved parameterized algorithms for planar dominating set To appear in *Proc. 27th MFCS*, Springer-Verlag LNCS, Warszawa - Otwock, Poland, August 2002.
161. → 77
R. Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of Operations Research*, 12:415–440, 1987.
162. → 119
S. Kannan and T. Warnow. A Fast algorithm for the computation and enumeration of perfect phylogenies. *SIAM Journal on Computing*, 26(6):1749–1763, 1997.
163. → 132
H. Kaplan, R. Shamir, and R. E. Tarjan. Tractability of parameterized completion problems on chordal, strongly chordal, and proper interval graphs. *SIAM Journal on Computing*, 28(5):1906–1922, 1999.
164. → 31
S. Khuller. Algorithms column: the vertex cover problem. *ACM SIGACT News*, 33(2):31–33, 2002.
165. → 119
P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Data Bases*. PhD thesis, University of Helsinki, Finland. 1992.
166. → 119
P. Kilpeläinen and H. Mannila. Ordered and unordered tree inclusion. *SIAM Journal on Computing* 24(2): 340–356, 1995.
167. → 89
T. Kloks. *Treewidth. Computations and Approximations*. Springer-Verlag LNCS 842, 1994.
168. → 22, 94, 96
A. M. C. A. Koster, H. L. Bodlaender, and S. P. M. Hoesel. Treewidth: computational experiments. *Electronic Notes in Discrete Mathematics 8*, Elsevier Science Publishers, 2001.
169. → 43
O. Kullmann. New methods for 3-SAT decision and worst-case analysis. *Theoretical Computer Science*, 223(1-2):1–72, 1999.
170. → 43
O. Kullmann and H. Luckhardt. Algorithms for SAT/TAUT decision based on various measures. Preprint, 71 pages, available from `http://www.cs.toronto.edu/~kullmann`, February 1999.
171. → 15, 21, 125, 126
S.-Y. Kuo and W.K. Fuchs. Efficient spare allocation for reconfigurable arrays. *IEEE Design and Test*, 4:24–31, 1987.
172. → 77
J. C. Lagarias. Point lattices. In R. L. Graham *et al.* (eds.) *Handbook of Combinatorics*, pp. 919–966. The MIT Press, 1995.

173. → 117
G. Lancia, R. Carr, B. Walenz, and S. Istrail. 101 optimal PDB structure alignments: a branch-and-cut algorithm for the maximum contact map overlap problem. In *Proc. 5th ACM Annual International Conference on Computational Molecular Biology (RECOMB)*, pp. 193–202, 2001.
174. → 21, 111, 112
J. K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang. Distinguishing string selection problems. In *Proc. 10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 633–642, 1999. Long version to appear in *Information and Computation*.
175. → 91
J. van Leeuwen. Graph Algorithms. In *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity*, pp. 525–631, North Holland, 1990.
176. → 43, 70
S. Lehnert. *Experimental Analysis of a Search Tree Algorithm for Dominating Set*. Study Work, Universität Tübingen, Germany, in preparation, 2002.
177. → 77
H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983.
178. → 111, 112
M. Li, B. Ma, and L. Wang. Finding similar regions in many sequences. To appear in *Journal of Computer and System Sciences*, 2002.
179. → 108, 111, 112
M. Li, B. Ma, and L. Wang. On the closest string and substring problems. *Journal of the ACM*, 49(2):157–171, 2002.
180. → 121
D. Liben-Nowell. Gossip is synteny: incomplete gossip and the syntenic distance between genomes. *Journal of Algorithms*, 43(2):264–283, 2002.
181. → 25
P. Liberatore. Monotonic reductions, representative equivalence, and compilation of intractable problems. *Journal of the ACM*, 48(6):1091–1125, 2001.
182. → 135
O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their specification. In *Proc. 12th Annual ACM Symp. on Principles of Prog. Lang. (POPL)*, pp. 97–107, 1985.
183. → 117, 118
G.-H. Lin, Z.-Z. Chen, T. Jiang, and J. Wen. The longest common subsequence problem for sequences with nested arc annotations. In *Proc. 28th ICALP*, Springer-Verlag LNCS 2076, pp. 444–455, 2001.
184. → 131
R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Mathematics*, 36(2):177–189, 1979.
185. → 129
L. Lovasz and M. D. Plummer. *Matching Theory*. Annals of Discrete Mathematics, 29, North Holland, 1986.
186. → 13, 19, 29, 66, 67, 75, 130
M. Mahajan and V. Raman. Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms*, 31:335–354, 1999.
187. → 135
C. McCartin. An improved algorithm for the jump number problem. *Information Processing Letters*, 79:87–92, 2001.

188. → 15, 141
C. McCartin. Parameterized counting problems. To appear in *Proc. 27th MFCS*, Springer-Verlag LNCS, Warszawa - Otwock, Poland, August 2002.
189. → 2, 11
K. Mehlhorn. *Graph Algorithms and NP-Completeness*. Springer-Verlag, 1984.
190. → 1
Z. Michalewicz and D. B. Fogel. *How to Solve It: Modern Heuristics*. Springer-Verlag, 2000.
191. → 133
B. Mohar. A linear time algorithm for embedding graphs in an arbitrary surface. *SIAM J. Discrete Mathematics*, 12(1):6–26, 1999.
192. → 13
B. Monien and E. Speckenmeyer. Ramsey numbers and an approximation algorithm for the vertex cover problem. *Acta Informatica*, 22:115–123, 1985.
193. → 111
B. M. E. Moret, D. A. Bader, and T. Warnow. High-performance algorithm engineering for computational phylogenetics. *Journal of Supercomputing*, 22:99–111, 2002.
194. → 111
B. M. E. Moret, S. K. Wyman, D. A. Bader, T. Warnow, and M. Yan. A new implementation and detailed study of breakpoint analysis. In *Proc. 6th Pacific Symposium on Biocomputing*, pp. 583–594. 2001.
195. → 109
B. M. E. Moret, L. Wang, T. Warnow, and S. K. Wyman. New approaches to phylogeny reconstruction from gene order data. *Bioinformatics* 17:S165–S173, 2001.
196. → 135, 139
P. Moscato. New optimization and decision problems on graphs arising from the theory of memetic algorithms. Manuscript, December 2000.
197. → 1, 85
R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.
198. → 132
A. Natanzon, R. Shamir, and R. Sharan. Complexity classification of some edge modification problems. *Discrete Applied Mathematics*, 113(1):109–128, 2001.
199. → 12, 26, 27, 31, 33, 140
G. L. Nemhauser and L. E. Trotter Jr. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975.
200. → 77
G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, 1988.
201. → 6
J. Nešetřil and S. Poljak. On the complexity of the subgraph problem. *Commentationes Mathematicae Universitatis Carolinae*, 26(2):415–419, 1985.
202. → 1, 141
R. Niedermeier. Some prospects for efficent fixed parameter algorithms. In *Proc. 25th SOFSEM*, Springer-Verlag LNCS 1521, pp. 168–185, 1998.
203. → 44, 81, 82
R. Niedermeier and P. Rossmanith. Upper bounds for vertex cover further improved. Technical Report KAM-DIMATIA Series 98-411, Faculty of Mathematics and Physics, Charles University, Prague, November 1998.

204. → 2, 6, 10, 12, 16, 44, 73, 74, 75, 81, 82, 83, 99
R. Niedermeier and P. Rossmanith. Upper bounds for vertex cover further improved. In *Proc. 16th STACS*, Springer-Verlag LNCS 1563, pp. 561–570, 1999.
205. → 10, 70
R. Niedermeier and P. Rossmanith. A general method to speed up fixed-parameter-tractable algorithms. *Information Processing Letters*, 73:125–129, 2000.
206. → 54, 73, 75
R. Niedermeier and P. Rossmanith. New upper bounds for Maximum Satisfiability. *Journal of Algorithms*, 36: 63–88, 2000.
207. → 2, 6, 10, 12, 14, 20, 73, 74, 121, 123, 125
R. Niedermeier and P. Rossmanith. On efficient fixed-parameter algorithms for weighted vertex cover. In *Proc. 11th ISAAC*, Springer-Verlag LNCS 1969, pp. 180–191, 2000. Revised version submitted to *Journal of Algorithms*.
208. → 14, 30, 48, 73
R. Niedermeier and P. Rossmanith. An efficient fixed parameter algorithm for 3-Hitting Set. *Journal of Discrete Algorithms*, 2(1):93–107, 2002.
209. → 14
N. Nishimura, P. Ragde, and D. T. Thilikos. Fast fixed-parameter tractable algorithms for nontrivial generalizations of vertex cover. In *Proc. 7th WADS*, Springer-Verlag LNCS 2125, pp. 75–86, 2001.
210. → 79
A. M. Odlyzko. Asymptotic enumeration methods. In R. L. Graham *et al.* (eds.) *Handbook of Combinatorics*, pp. 1063–1229. The MIT Press, 1995.
211. → 19, 20
J. Opatrny. Total ordering problem. *SIAM Journal on Computing*, 8(1):111–114, 1979.
212. → 2, 5, 7
C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
213. → 77
C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.
214. → 5, 66
C. H. Papadimitriou and M. Yannakakis. Optimization, approximation, and complexity classes. *Journal of Computer and System Sciences*, 43:425–440, 1991.
215. → 9, 11
C. H. Papadimitriou and M. Yannakakis. On limited nondeterminism and the complexity of the V-C dimension. *Journal of Computer and System Sciences*, 53:161–170, 1996.
216. → 135
Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. In *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pp. 12–19, 1997.
217. → 12, 27
V. T. Paschos. A survey of approximately optimal solutions to some covering and packing problems. *ACM Computing Surveys*, 29(2):171–209, 1997.
218. → 18, 108
I. Pe'er and R. Shamir. The median problems for breakpoints are NP-complete. ECCC Technical Report TR98-071, Trier, Germany, 1998.

219. → 109
I. Pe'er and R. Shamir. Approximation algorithms for the permutations median problem in the breakpoint model. In D. Sankoff and J. Nadeau (eds.): *Comparative Genomics*, pp. 225–241. Kluwer, 2000.
220. → 104
P. A. Pevzner. *Computational Molecular Biology: An Algorithmic Approach*. The MIT Press, 2000.
221. → 112, 113, 116
P. A. Pevzner and S.-H. Sze. Combinatorial approaches to finding subtle signals in DNA sequences. In *Proc. of 8th International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pp. 269–278, 2000. AAAI Press.
222. → 117
K. Pietrzak. On the parameterized complexity of the fixed alphabet shortest common supersequence and longest common subsequence problems. To appear in *Journal of Computer and System Sciences*, 2002.
223. → 129
S. Poljak and Z. Tuza. Maximum cuts and large bipartite subgraphs. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 20:181–244, 1995.
224. → 1
V. Raman. Parameterized complexity. In *Proceedings of the 7th National Seminar on Theoretical Computer Science (Chennai, India)*, pp. I–1–I–18, June 1997.
225. → 94, 96
B. Reed. Finding approximate seperators and computing tree width quickly. In *Proc. 24th ACM Symposium on Theory of Computing (STOC)*, pp. 221–228, 1992.
226. → 121, 138
S. Rash and D. Gusfield. String barcoding: uncovering optimal virus signatures. In *Proc. 6th ACM Annual International Conference on Computational Molecular Biology (RECOMB)*, pp. 254–261, 2002.
227. → 121
R. Rizzi, V. Bafna, S. Istrail, and G. Lancia. Practical algorithms and fixed-parameter tractability for the single individual SNP haplotyping problem. To appear in *Proc. 2nd Workshop on Algorithms in Bioinformatics*, Springer-Verlag LNCS, Rome, September 2002.
228. → 13, 27
N. Robertson, D. P. Sanders, P. Seymour, and R. Thomas. Efficiently four-coloring planar graphs. In *Proc. 28th ACM Symposium on Theory of Computing (STOC)*, pp. 571–575, 1996.
229. → 13, 27
N. Robertson, D. P. Sanders, P. Seymour, and R. Thomas. The four-color theorem. *Journal of Combinatorial Theory, Series B*, 70(1):2–44, 1997.
230. → 22, 87
N. Robertson and P. D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7:309–322, 1986.
231. → 2, 81
J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7:425–440, 1986.
232. → 2, 82
J. M. Robson. Finding a maximum independent set in time $O(2^{n/4})$ Technical Report 1251-01, Université Bordeaux, LaBRI, 2001.

233. → 82
J. M. Robson. Personal communication. August 2001.
234. → 15, 16, 74, 137
P. Rossmanith. Personal communication. May 2002.
235. → 113, 121
M.-F. Sagot. Spelling approximate repeated or common motifs using a suffix tree. In *Proc. 3rd LATIN*, Springer-Verlag LNCS 1380, pp. 111–127, 1998.
236. → 18, 111
D. Sankoff and M. Blanchette. Multiple genome rearrangement and breakpoint phylogeny. *Journal of Computational Biology*, 5:555–570, 1998.
237. → 77
A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, 1999.
238. → 104
J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, 1997.
239. → 132
R. Shamir, R. Sharan, and D. Tsur. Cluster graph modification problems. To appear in *Proc. 28th Workshop on Graph Theoretical Concepts in Computer Science (WG)*, Springer-Verlag LNCS, 2002.
240. → 108
A. C. Siepel and B. M. E. Moret. Finding an optimal inversion median: experimental results. In *Proc. 1st WABI*, Springer-Verlag LNCS 2149, pp. 189–204, 2001.
241. → 2
S. S. Skiena. *The Algorithm Design Manual.* Springer-Verlag, 1998.
242. → 135
A. Slivkins and M. Pál. On fixed-parameter tractability of some routing problems. Manuscript, Cornell University, July 2002.
243. → 22
D. Slonim, L. Stein., L. Kruglyak, and E. Lander. Rhmapper: An interactive computer package for constructing radiation hybrids maps. `http://www.genome.wi.mit.edu/ftp/pub/software/rhmapper/`, 1996.
244. → 131
D. A. Spielman and S.-H. Teng. Disk packings and planar separators. In *12th Annual ACM Symposium on Computational Geometry (SCG)*, pp. 349–358, 1996.
245. → 105
M. Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9:91–116, 1992.
246. → 19, 105
M. Steel. Personal communication. May 2001.
247. → 120
U. Stege. Gene trees and species trees: The gene-duplication problem is fixed-parameter tractable. In *Proc. 6th WADS*, Springer-Verlag LNCS 1663, pp. 288–293, 1999.
248. → 16
U.Stege. *Resolving Conflicts in Problems from Computational Biology.* PhD thesis No. 13364, ETH Zürich, 2000.
249. → 12, 71, 75
U. Stege and M. Fellows. An improved fixed-parameter-tractable algorithm for vertex cover. Technical Report 318, Department of Computer Science, ETH Zürich,

250. → 22
L. Stein, L. Kruglyak, D. Slonim, and E. Lander. Building human genome maps with radiation hybrids. In *Proceedings of the 1st ACM Annual International Conference on Computational Molecular Biology (RECOMB)*, pp. 277–286, 1997.
251. → 112
N. Stojanovic, P. Berman, D. Gumucio, R. Hardison, and W. Miller. A linear-time algorithm for the 1-mismatch problem. In *Proc. 5th WADS*, Springer-Verlag LNCS 1272, pp. 126–135, 1997.
252. → 112
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.
253. → 105
K. Strimmer and A. von Haessler. Quartet puzzling: a quartet maximum-likelihood method for reconstructing tree topologies. *Molecular Biology and Evolution*, 13(7):964–969, 1996.
254. → 121, 138
M. Tang, M. Waterman, and S. Yooseph. Zinc finger gene clusters and tandem gene duplication. In *Proc. 5th ACM Annual International Conference on Computational Molecular Biology (RECOMB)*, pp. 297–304, 2001.
255. → 100, 101
J. A. Telle and A. Proskurowski. Practical algorithms on partial k-trees with an application to domination-like problems. In *Proc. 3rd WADS*, Springer-Verlag LNCS 709, pp. 610–621, 1993.
256. → 100, 101
J. A. Telle and A. Proskurowski, Algorithms for vertex partitioning problems on partial k-trees, *SIAM Journal on Discrete Mathematics*, 10(4):529–550, 1997.
257. → 87, 88
M. Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142:159–181, 1998.
258. → 5
V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.
259. → 131
S. M. Venkatesan. Improved constants for some separator theorems. *Journal of Algorithms*, 8:572–578, 1987.
260. → 135
H. T. Wareham. The parameterized complexity of intersection and composition operations on sets of finite-state automata. In *Proc. 5th CIAA*, Springer-Verlag LNCS 2088, pp. 302–310, 2000.
261. → 104
M. S. Waterman. *Introduction to Computational Biology (Maps, Sequences and Genomes)*. Chapman & Hall, 1995.
262. → 25
K. Weihe. Covering trains by stations or the power of data reduction. In *Proc. 1st ALEX'98*, pp. 1–8, 1998.
`http://rtm.science.unitn.it/alex98/proceedings.html`
263. → 25
K. Weihe. On the differences between "practical" and "applied". In *Proc. WAE 2000*, Springer-Verlag LNCS 1982, pp. 1–10, 2001.

264. → 67
M. Yannakakis. On the approximation of maximum satisfiability. *Journal of Algorithms*, 17(3):457–502, 1994.
265. → 135
M. Yannakakis. Perspectives on database theory. In *Proc. 36th IEEE Symposium on Foundations of Computer Science*, pp. 224–246, 1995.
266. → 66
U. Zwick. Outward rotations: a tool for rounding solutions of semidefinite programming relaxations, with applications to MAX CUT and other problems. In *Proc. 31st ACM Symposium on Theory of Computing (STOC)*, pp. 679–687, 1999.