Advance Access publication on July 18, 2007 doi:10.1093/comjnl/bxm040

Techniques for Practical Fixed-Parameter Algorithms

Falk Hüffner, Rolf Niedermeier* and Sebastian Wernicke

Institut für Informatik, Friedrich-Schiller-Universität Jena, Ernst-Abbe-Platz 2, D-07743 Jena, Germany
**Corresponding author: niedermr@minet.uni-jena.de*

The fixed-parameter approach is an algorithm design technique for solving combinatorially hard (mostly NP-hard) problems. For some of these problems, it can lead to algorithms that are both efficient and yet at the same time guaranteed to find optimal solutions. Focusing on their application to solving NP-hard problems in practice, we survey three main techniques to develop fixed-parameter algorithms, namely: kernelization (data reduction with provable performance guarantee), depth-bounded search trees and a new technique called iterative compression. Our discussion is circumstantiated by several concrete case studies and provides pointers to various current challenges in the field.

Keywords: parameterized algorithms; fixed-parameter tractability

Received 25 February 2006; revised 11 December 2006

1. INTRODUCTION

The NP-hard problems are difficult to solve efficiently and optimally at the same time because one has to deal with a combinatorial explosion of the search space. Thus, whenever a problem is proven to be NP-hard and large instances of it need to be solved, it is common practice to employ heuristic algorithms (that are either not guaranteed to yield optimal solutions or give no useful guarantees concerning their running time), approximation algorithms, or even attempt to sidestep the problem altogether.

Not all instances of an NP-hard problem are equally hard to solve, however. Rather, this hardness depends on the particular *structure* of a given instance. Opposed to 'classical' computational complexity theory—which sees problem instances only in terms of their size—the concept of *fixed-parameter tractability* (FPT) [1–3] reflects such differences in structural hardness by expressing them through a so-called parameter, which is a nonnegative integer variable usually denoted k.

The FPT thus generalizes the concept of 'easy special cases' that is known for virtually all NP-hard problems: whenever the parameter k turns out to be *small* for an instance of an NP-hard problem, a fixed-parameter algorithm can solve this instance quite fast (sometimes even in linear time)—with provable bounds on the running time and guaranteeing the optimality of the solution. More precisely, a size-n instance of a fixed-parameter tractable problem can be solved in $f(k) \cdot p(n)$ time, where f is a function that solely depends on the parameter k and $p(n)$ is a polynomial in n. Establishing the FPT of an NP-hard problem thus implies that the combinatorial explosion that is inherent to solving it can be fully confined to the parameter.

As parameterized complexity theory points out, there are problems that are likely not fixed-parameter tractable with respect to a specific parameter [1–3]. For those NP-hard problems that *are* fixed-parameter tractable, however, their FPT constitutes far more than a mere 'theoretical curiosity.' Rather, there are numerous examples where fixed-parameter techniques have proved themselves to be quite useful and relevant to solving some NP-hard problems both efficiently and optimally in practice. The presentation of some of these is the central concern of this paper.

Together with concrete application scenarios, we review three important techniques for designing such *practical* fixed-parameter algorithms.[1] This is accompanied by a discussion of several experimental results, which underpin that the concept of FPT belongs into the toolkit of every algorithm designer—no matter whether they have a more theoretical or a more practical orientation.

Most of the problems we deal with in this work are from graph theory [5]; by default, we consider only simple undirected graphs $G = (V, E)$, and use n to denote the number of its vertices and m to denote the number of edges.

[1] An older survey in this direction is given by Fellows [4]. A broader perspective on the techniques and algorithms we present in this work can be found in the monograph [3].

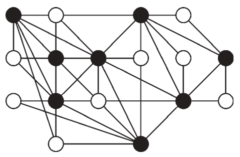

FIGURE 1. A graph with a size-8 vertex cover (cover vertices are marked black).

The fixed-parameter techniques that we exhibit are

- kernelizations, that is, data reduction schemes with provable performance guarantees (Section 2),
- depth-bounded search trees (Section 3) and
- iterative compression (Section 4).

We conclude in Section 5 with a brief discussion of further techniques relevant in the context of FPT.

To better illustrate the techniques we survey, they are all introduced by means of a single natural and easy to grasp problem from graph theory, namely the NP-hard Vertex Cover problem.

Vertex Cover
Input: an undirected graph $G = (V, E)$ and a nonnegative integer k.
Task: find a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.

An illustration for this problem is provided in Fig. 1. Solving Vertex Cover is of relevance to many applications ranging from network monitoring and prevention of denial of service attacks [6] to bioinformatics-related scenarios such as microarray data analysis [7] and the computation of multiple sequence alignments [8]. In FPT research, Vertex Cover plays a special role in that many important discoveries that influenced the whole field originated from the study of this problem.

2. KERNELIZATION: DATA REDUCTION WITH GUARANTEED PERFORMANCE

Before firing up a computationally expensive algorithm to solve a combinatorially hard problem (such as efficiently monitoring network transmission links or cost-effectively locating wireless transmitters), it suggests itself to try a *reduction* of the input data. The idea is to quickly presolve those parts of the input data that are relatively easy to cope with, shrinking it to those parts that form the 'really hard' core of the problem. Costly algorithms need then only be applied to this reduced instance. In some practical scenarios, data reduction may even reduce a seemingly hard problem to triviality [9, 10].

Early examples for data reduction techniques were already given by Quine [11] in 1952. Today, there are many examples of problem instances that would not be solvable without using data reduction and preprocessing algorithms.[2] For example, Bixby [13] gives a striking account with respect to the linear program solver CPLEX, discussing a large linear program that can be solved in half an hour when data reduction is employed but is 'far from even being feasible' [13] for the unreduced instance even after hours of computation. Another impressive account of the power of data reduction is given by Weihe [10] where—in the context of the European railroad network—two simple data reduction rules allow an NP-hard problem to be solved in mere minutes for a graph consisting of more than 1.6×10^5 vertices and 1.6×10^6 edges.

Some data reductions rules are very simple and readily discovered by everybody tackling a problem. Others are hidden gems that require deeper digging and insight into a problem's structure. Once an effective (and efficient) reduction rule has been found; however, it is useful in virtually any problem solving context, whether it be heuristic, approximative or exact.

Clearly, practitioners are likely to already be aware of data reduction rules. Why should they also consider FPT in this context? The reason is that fixed-parameter theory provides a way to use data reduction rules not only in a heuristic way, but to *prove their power* by so-called *kernelizations*. These give an upper bound on the size of a reduced instance that solely depends on the parameter value. This opens the door to a potentially fruitful dialogue between practitioners and theoreticians: kernelizations can explain, and prove, why rules work so well in practice; and the quest for kernelizations can lead to new and powerful data reduction rules based on deep structural insights.

2.1. An introductory example

Consider our running example Vertex Cover. To reduce the input size for a given instance of this problem, it is clearly permissible to remove isolated vertices, that is, vertices with no adjacent edges. This leads to a first simple data reduction rule.

Reduction Rule VC1
Remove all isolated vertices.

In order to cover an edge in the graph, one of its two endpoints *must* be in the vertex cover. If one of these is a degree-1 vertex, then the other endpoint has the potential to cover more edges than the degree-1 vertex, leading to a second reduction rule.

Reduction Rule VC2
For degree-1 vertices, put their neighbouring vertex into the cover.

[2]The use of data reduction techniques is not restricted to a preprocessing phase only. On the contrary, there is empirical as well as theoretical evidence that interleaving data reduction techniques with the 'main' problem solving algorithm can yield significant speedups (see Section 3.4 and [12]).

Here, 'put into the cover' means adding the vertex to the solution set and removing it and its incident edges from the instance. Note that this reduction rule assumes that we are only looking for *one* optimal solution to the VERTEX COVER instance we are trying to solve; there may exist other minimum-cardinality vertex covers that do include the reduced degree-1 vertex. Hence, this reduction rule is not suitable where it is of interest to enumerate *all* vertex covers of a given graph in order to choose the one most suitable for the application at hand.

After having applied the easy rules VC1 and VC2, we can further do the following in the fixed-parameter setting where we ask for a vertex cover of size at most k.

REDUCTION RULE VC3
If there is a vertex of degree at least $k + 1$, put this vertex into the cover.

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

After exhaustively performing the rules VC1–VC3, no vertex in the remaining graph has a degree higher than k, meaning that choosing a vertex into the cover can cause at most k edges to become covered. Since the solution set may be no larger than k, the remaining graph can have at most k^2 edges if it is to have a solution. By rules VC1 and VC2, every vertex has degree at least two, which implies that the remaining graph can contain at most k^2 vertices.

More abstractly speaking, what have we done here? After applying a number of rules *in polynomial time* to an instance of VERTEX COVER, we arrived at a reduced instance whose size can *solely be expressed in terms of the parameter k*. These properties are formalized in the concept of a *problem kernel* [1].

DEFINITION 2.1. *Let $\mathcal{L}$ be a parameterized problem, that is, $\mathcal{L}$ consists of input pairs (I, k), where I is the problem instance and k is the parameter. A reduction to a problem kernel (or kernelization) means to replace an instance (I, k) by a reduced instance (I',k') called problem kernel such that*

(1) $k' \leq k$,
(2) *the size of I' is smaller than $g(k)$ for some function g only depending on k and*
(3) *(I, k) has a solution if and only if (I',k') has one. The reduction from (I, k) to (I',k') must be computable in polynomial time.*

While this definition does not formally require that it is possible to reconstruct a solution for the original instance from a solution for the problem kernel, all kernelizations we are aware of easily allow for this.

The methodological approach of devising kernelizations, including various techniques of data reduction, is best learned by some concrete examples which we discuss in Sections 2.2 and 2.3.

Before moving on to these case studies, let us state some useful general observations and remarks concerning Definition 2.1 and its connections to FPT. Most notably, there is a close connection between fixed-parameter tractable problems and those problems for which there exists a problem kernel—they are exactly the same.

THEOREM 2.1 [14]. *Every problem that is fixed-parameter tractable is kernelizable and vice versa.*

Unfortunately, the practical use of this theorem is limited: the running time of a fixed-parameter algorithm obtained directly from a kernelization is usually not practical; and, in the other direction, the theorem does not constructively provide us with a data reduction scheme for a fixed-parameter tractable problem. Hence, the main use of Theorem 2.1 is to establish the FPT or amenability to kernelization of a problem—or show that we need not search any further (e.g. if a problem is known to be fixed-parameter intractable, we do not need to look for a kernelization).

Rule VC3 explicitly needed the value of the parameter k. We call this a *parameter-dependent* rule as opposed to the parameter-independent rules VC1 and VC2, which are oblivious to k. In applications, one typically does not know the actual value of k in advance and has to get around this by iteratively increasing the values of k. Although asymptotically this is not slower, in practice one would of course prefer to avoid this extra outer loop. However, assuming explicit knowledge of the parameter clearly adds some leverage to finding data reduction rules and is hence frequently encountered in kernelizations.

We continue with two case studies in Sections 2.2 and 2.3. A broader overview can be found in the literature [3, 15, 16].

2.2. VERTEX COVER kernelization further explored

In Section 2.1, we discussed a simple to achieve size-$O(k^2)$ problem kernel for VERTEX COVER. The corresponding parameter-dependent kernelization is based on a simple observation concerning high-degree vertices. There are several more kernelization techniques for VERTEX COVER—two of which we explore further here—which feature parameter independence and much improved bounds on the kernel size. (A more detailed treatment can be found in [17–19].)

2.2.1. Kernelization based on matching

The kernelization for VERTEX COVER in Section 2.1 is based on piecing together very simple data reduction rules that examine *local* substructures. This is a frequently employed and successful approach. In this section, in contrast, we show a kernelization based on global properties of a VERTEX COVER instance. It is based on the following theorem of Nemhauser and Trotter from 1975 [20].

THEOREM 2.2. *For an n-vertex graph $G = (V, E)$ with m edges, we can compute two disjoint sets $C' \subseteq V$ and $V' \subseteq V$ in $O(\sqrt{n} \cdot m)$ time such that the following three properties hold.*

(1) *There is a minimum-size vertex cover of G that contains C'.*
(2) *A minimum vertex cover for the induced subgraph $G[V']$ has size at least $|V'|/2$.*
(3) *If $D \subseteq V'$ is a vertex cover of the induced subgraph $G[V']$, then $C := D \cup C'$ is a vertex cover of G.*

Determining the two sets C' and V' in this theorem involves the computation of a maximum bipartite matching on a graph constructed from G. While Theorem 2.2 may look somewhat unwieldy at first sight, an observation by Chen *et al.* [21] makes it the key to one of the best data reduction procedures known for VERTEX COVER.

THEOREM 2.3. *Let $(G = (V, E),k)$ be an instance of* VERTEX COVER. *In $O(k \cdot |V| + k^3)$ time we can reduce this instance to a kernel $(G' = (V',E'),k')$ with $|V'| \leq 2\,k$.*

Proof. We begin by using the reduction to a problem kernel as sketched in Section 2.1 to get a reduced instance containing at most $O(k^2)$ vertices and edges. This reduction takes $O(k \cdot |V|)$ time. It may decrease the parameter value by putting some vertices into the cover and we let the new parameter value be $k'' \leq k$. On the resulting reduced instance, the two sets C' and V' as described in Theorem 2.2 can be computed in time $O(\sqrt{k^2} \cdot k^2) = O(k^3)$.

The set C' contains vertices that have to be in the vertex cover, and we define $k' := k'' - |C'|$. Observe that due to Theorem 2.2 we directly know that if $|V'| > 2\,k'$, then there is no vertex cover of size k for the original graph G. Otherwise, we let the subgraph induced by V' in G be the problem kernel (of size at most $2\,k' \leq 2\,k$), knowing by Theorem 2.2 that the remaining vertices for a minimum vertex cover of G can be found by searching for a minimum vertex cover in it. □

Can we find even smaller problem kernels for vertex cover of size, say, $1.5\,k$? There is some evidence that this is not possible because all kernelizations we know today are also approximation algorithms: if there were a kernel for VERTEX COVER that guaranteed to have only $1.5\,k$ vertices, we could in polynomial time obtain a vertex cover for G that is at most 1.5 times larger than the optimum k simply by taking all vertices of the kernel. This would be a major breakthrough in approximation theory, since it has been conjectured that it is not possible to polynomial-time approximate vertex cover to within a constant factor smaller than two [22]. Therefore, it seems unlikely that the size of the VERTEX COVER kernel can be further improved.

Theorem 2.2 can be generalized in two ways: one concerns finding minimum *weighted* vertex covers, where vertices have a positive real weight [23]. The second one concerns finding *all* vertex covers up to size$k$; Theorem 2.2 only leads to *one* particular minimum vertex cover, excluding others from further consideration. Recent research shows how to modify these results in order to obtain *all* optimal solutions [23].

Next we present an alternative kernelization for VERTEX COVER which also achieves a problem kernel of size linear in k and is parameter-independent.

2.2.2. *Kernelization based on crown structures*

Many data reduction rules examine only specific local substructures of the input such as a vertex and its neighbourhood (e.g. Reduction Rules VC2 and VC3). Recently, there have been several examples of generalizing such rules to examining arbitrarily large substructures. This can considerably increase their power, as we demonstrate here with the *crown reduction* rule for VERTEX COVER, which generalizes Rule VC2 (the elimination of degree-1 vertices by taking their neighbours into the cover).

A *crown* in a graph consists of an independent set I (that is, no two vertices in I are connected by an edge) and a set H containing all vertices adjacent to I. In order for $I \cup H$ to be a crown, there has to exist a size-$|H|$ maximum matching in the bipartite graph induced by the edges between I and H (i.e. one in which every vertex of H is matched). An example for a crown structure is given in Fig. 2—in a sense, degree-1 vertices are the most simple crowns.

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

Two issues remain to be dealt with, namely how to find crowns efficiently and giving an upper bound on the size of the problem kernel that can be obtained via crown reductions. It turns out that finding crowns can—as with the Nemhauser–Trotter kernelization—be achieved in polynomial time by computing maximum matchings [24]. The size of the thus reduced instance is upper-bounded via the following theorem.

THEOREM 2.4. *A graph that is crown-free and has a vertex cover of size at most k can contain at most 3 k vertices.*

Thus, by generalizing Reduction Rule VC2 which by itself is not a kernelization, we have obtained an efficient

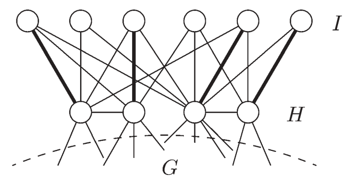

FIGURE 2. A graph G with a crown $I \cup H$. Note how the thick edges constitute a maximum matching of size $|H|$ in the bipartite graph induced by the edges between I and H.

parameter-independent kernelization which yields a linear-size kernel for VERTEX COVER. More examples for reductions for VERTEX COVER that are based on arbitrarily-large graph substructures are given by Chen *et al.* [25].

Crown reductions also demonstrate how structurally very different viewpoints of the same problem (in this case, kernelizing a given instance of VERTEX COVER) lead to similar results (a kernel of size linear in k). Along the same lines, there has recently been interesting research into the interconnections between crown reduction rules and kernelization based on linear programming [18].

2.2.3. *Applications*

VERTEX COVER can be considered the *Drosophila* of fixed-parameter research in that many initial discoveries that influenced the whole field originated from studies of this single problem. It comes as no surprise that also the experimental field is more advanced for this problem than for others from the realm of FPT.

Abu-Khzam *et al.* [17] studied various kernelization schemes for VERTEX COVER and their practical performance both with respect to time as well as with respect to the resulting kernel size. For bioinformatics-related networks derived from protein databases and microarray data, they found that crown reductions turn out to be very fast to compute in practice (almost as fast as an extended version of Reduction Rules VC1–VC3) and are sometimes just as effective as approaches with a theoretically better bound (such as the Nemhauser–Trotter reduction) while at the same time being faster to carry out. Abu-Khzam *et al.* therefore recommend to always use crown reduction as a general preprocessing step when solving VERTEX COVER before attempting other, more costly, reduction schemes.

Another interesting problem where VERTEX COVER reduction rules have successfully been applied is that of searching maximum cliques (that is, maximum-size fully connected subgraphs), making use of the fact that if an n-vertex graph has a maximum clique of size $(n - k)$, then its complement graph has a size-k minimum vertex cover. Details and experimental results with applications to computational biology are given by Abu-Khzam *et al.* [17, 18, 26].

2.3. The DOMINATING SET problem

Domination is a central concept in graphs and can safely be considered a whole research area on its own—already in 1998, more than 200 research papers studied the algorithmic complexity of domination [27]. The most basic domination problem is DOMINATING SET.

DOMINATING SET
INPUT: an undirected graph $G = (V, E)$ and a nonnegative integer k.

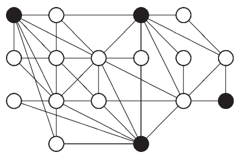

FIGURE 3. A graph with a size-4 dominating set (marked black).

TASK: find a subset $S \subseteq V$ with at most k vertices such that every vertex $v \in V$ is either in S or has at least one neighbour in S.

The problem is illustrated in Fig. 3. As an intuition, one might think of a minimum dominating set as a set of most important vertices that are able to 'observe' or 'control' all other vertices. A straightforward scenario for domination problems is in supply station location tasks, but there are numerous other applications—e.g. in voting scenarios and network analysis [28, 29]—where domination plays a key role.

DOMINATING SET is NP-hard and known to be intractable even from a fixed-parameter point of view, that is, it is highly unlikely that there is a fixed-parameter algorithm for dominating set that confines the exponential part of its running time to the solution size k [1]. By Theorem 2.1, this means that there is also a little hope for a problem kernel for the general DOMINATING SET problem.

Therefore, why are we further discussing DOMINATING SET here in the context of FPT? Interestingly, Alber *et al.* [30] found two data reduction rules for DOMINATING SET that not only prove very effective in practice (especially for sparse graphs), but which also yield a parameter-independent reduction to a linear problem kernel for DOMINATING SET as long as the input graph is restricted to being planar. Hence, this is a good example of how structural insight gained during search for a kernelization proves useful even outside FPT.

Analogously to the crown reduction rules for VERTEX COVER, the DOMINATING SET rules we discuss in this section have recently been generalized to so-called *reduction schemes* from which they can be derived as special cases [31]. Some further reduction rules for DOMINATING SET are also given by Alber *et al.* [32] and Chen *et al.* [33].

2.3.1. *Reduction rules for domination*

Both reduction rules for DOMINATING SET discussed here are based on local neighbourhood considerations. The simpler one considers the neighbourhood of one vertex, the other one the neighbourhood of two vertices.

For the first reduction rule, we partition the neighbourhood $N(v)$ of an arbitrary vertex $v \in V$ in the input graph into three disjoint sets $N_1(v)$, $N_2(v)$ and $N_3(v)$ depending on local neighbourhood structure. More specifically, we define

- $N_1(v)$ to contain all neighbours of v that have edges to vertices that are not neighbours of v,

- N_2 (v) to contain all vertices from $N(v) \backslash N_1$ (v) that have edges to at least one vertex from N_1 (v) and
- $N_3(v)$ to contain all neighbours of v that are neither in N_1 (v) nor in N_2 (v).

An example that illustrates such a partitioning is given in the upper part of Fig. 4. A helpful and intuitive interpretation of the partition is to see vertices in N_1 (v) as *exits* because they have direct connections to the world outside the closed neighbourhood of v, vertices in N_2 (v) as *guards* because they have direct connections to exits and vertices in N_3 (v) as *prisoners* because they do not see the world outside $v \cup N(v)$.

Now consider a vertex $w \in N_3$ (v). This vertex is only capable of dominating v and those neighbours of v that only have connections to $N(v)$. Hence, to dominate w, at least one vertex of $\{v\} \cup N_2$ (v) $\cup$ N_3 (v) *must* be contained in a dominating set for the input graph. Since v can dominate all vertices that would be dominated by choosing a vertex from N_2 (v) $\cup$ N_3 (v) into the dominating set, we obtain the following data reduction rule for Dominating Set.

Reduction Rule DS1
If N_3 (v) $\neq \emptyset$ for some vertex v, then remove N_2 (v) and N_3 (v) from G and choose v to be in the dominating set.

Similar to this rule, we can also explore the neighbourhood set $N(v, w) := N(v) \cup N(w)$ of *two* vertices $v, w \in V$ that have distance at most three to each other. Analogously to the single vertex case, we partition $N(v, w)$ into three disjoint subsets as follows.

- N_1 (v, w) to contain all neighbours of v and w that have edges to vertices that are neither neighbours of v nor w.
- N_2 (v, w) to contain all vertices from the set $N(v, w) \backslash N_1$ (v, w) that have edges to at least one vertex from N_1 (v, w).

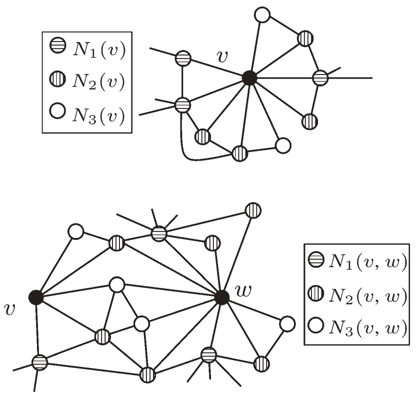

FIGURE 4. Partitioning the input graph of a dominating set instance. The upper part shows the partitioning of the neighbourhood of a single vertex v. Since N_3 (v) $\neq \emptyset$, Reduction Rule DS1 applies. The lower part shows the partitioning of a neighbourhood $N(v, w)$ of two vertices v and w. Since N_3 (v, w) cannot be dominated by a single vertex, Case 2 of Rule DS2 applies.

- N_3 (v, w) to contain all neighbours of $N(v, w)$ that are neither in N_1 (v, w) nor in N_2 (v, w).

The lower part of Fig. 4 shows an example, which illustrates the partitioning of $N(v, w)$ into the subsets $N_1(v, w)$, $N_2(v, w)$ and $N_3(v, w)$. Deferring further details to the literature [30], this neighbourhood partition yields a Reduction Rule DS2 that can be applied whenever $N_3(v, w)$ is nonempty. While the intuition is similar to the one-vertex case, this reduction rule is somewhat more complicated because it includes four different cases to distinguish.

2.3.2. *Reduction rule performance*

As mentioned above, we cannot hope for data reduction rules that yield a problem kernel for dominating set in the general case. However, for planar graphs, it is possible to prove with some technical expenditure that exhaustive application of reduction rules DS1 and DS2 actually is a parameter-independent reduction to a linear-size problem kernel.[3]

Theorem 2.5 [30]. *A planar graph G to which neither Rule DS1 nor DS2 can be applied has size at most* 335 *k where k is the size of a minimum dominating set in G.*

From a practitioner's standpoint, the constant 335 is not too intriguing (although recent efforts have improved it to 67 [33]). However, experimental results presented in several works [30, 35, 36] show a completely different picture in that in practice, these rules are very effective and yield much smaller kernels than the theoretical analysis suggests.

According to Alber *et al.* [30], amending Reduction Rules DS1 and DS2 by four very simple additional rules (which are quite similar to Reduction Rules VC1–VC3, see [35] for details) yields a powerful data reduction scheme that removes, on average, 99.7% of the vertices and 99.8% of the edges when solving dominating set on a random planar graph. Also, an equally high percentage of the vertices belonging to a minimum dominating set is detected. This yields some impressive examples of the power of data reduction such as the one shown in Fig. 5. Subsequent experimental studies [35] on some real-world networks such as power-law Internet topologies showed the rules to be just as effective there, quickly finding optimal dominating sets of up to a thousand vertices for networks consisting of around 10 000 vertices.

2.4. Practising kernelization

The main point of this section is to draw attention to the fact that, compared to the amount of papers on new fixed-parameter algorithms and the accompanying discovery of more and more (improved) problem kernels, very little experimental work has been done so far in order to assess the

[3]Fomin and Thilikos [34] generalize this result beyond planar graphs by showing that Rules DS1 and DS2 yield a linear-size problem kernel for any graph of bounded genus.

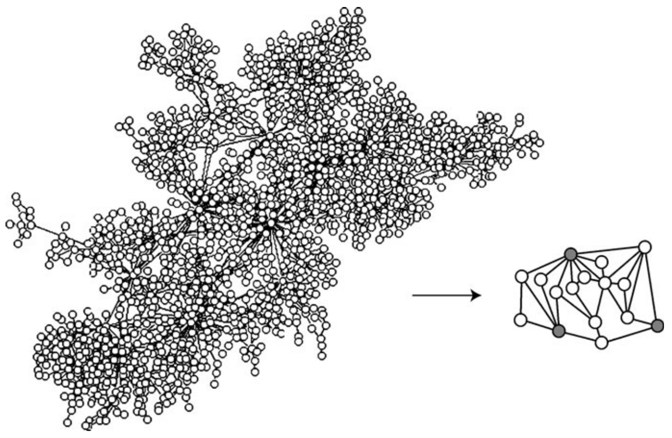

FIGURE 5. Example to illustrate the power of the dominating set Reduction Rules DS1 and DS2. Their application reduces the left-hand (planar) graph with approximately 1500 vertices to the right-hand 16-vertex graph. The black/white colouring of the reduced graph reflects that some vertices (the grey ones) are already known to be dominated due to application of the reduction rules.

practical value of these discoveries. This is somewhat surprising not only in that many fixed-parameter tractable problems are of considerable practical relevance, but also because experimentally studying their known kernelizations is beneficial both to theoretical as well as practical computer scientists in a number of ways as will be outlined here.

While some provable kernel bounds might be very large and thus not always appeal to a practitioner at first sight, the underlying data reduction may still perform very well in practice. One example for this is the reduction rules for Dominating Set we have presented in this section. Another, more recent, example of this is given in the work by Gramm *et al.* [37] concerning the NP-hard Clique Cover problem.

Clique Cover
Input: an undirected graph G and a nonnegative integer k.
Task: find a set of at most k cliques in G such that each edge has both endpoints in at least one such clique.

This problem has numerous applications in diverse fields such as compiler optimization, computational geometry and applied statistics. While Gramm *et al.* only prove an exponential kernel size of $O(2^k)$, their experimental results indicate a much better effectiveness on random and real-world data.

Generally speaking, kernelizations are often based on some deeper insights into the structural properties of a given problem in order to be able to prove that the size of the reduced instance is indeed only dependent on the parameter k. Such insight often leads to very 'well motivated' and hence effective reduction rules (even if only a large kernel size is provably achieved). Performing experimental studies on other fixed-parameter tractable problems and their kernelizations would likely motivate theoreticians to look at the 'hard cases' and devise further data reductions for them. While such an approach certainly is not much younger than data reduction itself, in the realm of FPT it allows not only for further 'average-case' improvements of data reduction but can also achieve provable performance guarantees and thus yield a solid basis for newly devised data reduction schemes.

While there exist some lower bounds on provable kernel sizes (e.g. Chen *et al.* [33] proved that there exists no size-$(2 - \varepsilon)k$ problem kernel with constant $\varepsilon > 0$ for Dominating Set in planar graphs unless $P = \text{NP}$), the gaps among theoretical lower bounds, practically achieved bounds and known theoretical upper bounds are still enormous and thus promise kernelization to provide a fruitful field for future research.

2.5. Current research and future challenges

As to some recent theoretical developments, Damaschke [38] suggests the notion of a *full kernel* that contains *all* small solutions to a given problem (in a compressed form) and thereby allows for their efficient enumeration. This is a recent area of research, and not many full kernels are known.

Another active field of research is the attempt to provide a more systematic way of finding good kernelizations by the *method of extremal structure* (also known as *method of coordinatized kernels*) [39, 40]. Also, a general methodology for developing linear-size problem kernels for planar graph problems has been recently introduced [41]. Finally, it has been tried to obtain data reduction rules in an automated way [42].

Besides the practical challenge of exploring the power of kernelizations, let us name several more theoretical challenges here that we find to be of relevance to the practitioner.

- Can the worst-case provable kernel size for Clique Cover be improved to polynomial?
- Can the technically involved data reduction rules for the network communication problem Multicut in Trees [43] be amended or better analysed in order to obtain a polynomial size problem kernel?
- Recently, data reductions for Feedback Vertex Set (see Section 4) have been reported that yield kernels of size polynomial in k [44, 45]. Are there data reductions that yield size-$O(k)$ (that is, linear-size) kernels?
- Parameterized by the size of a minimum solution, the 3-Hitting Set problem has a problem kernel of size $O(k^3)$ which can be found in linear time [46]. Can the underlying data reduction be generalized to apply to d-Hitting Set (for fixed values of d)?
- Maximum Satisfiability has a problem kernel of size $O(k^2)$ (where k is the minimum number of clauses that have to be satisfied) [47]. This problem kernel is uninteresting from an algorithmic standpoint because it makes

use of the property that we can *always* satisfy half of the clauses of a Boolean formula. Can data reduction be used to prove a better kernel that goes beyond this?

3. DEPTH-BOUNDED SEARCH TREES

The previous section studied data reduction and problem kernelization schemes for (pre)processing the given input data and cutting away its 'easy parts'. This leaves behind the 'really hard' problem kernel to be solved. A standard way to explore the huge search space related to optimally solving a computationally hard problem is to perform a systematic exhaustive search. This can be organized in a tree-like fashion, which is the subject of this section.

Certainly, search trees are no new idea and have been extensively used in the design of exact algorithms (e.g. see [48–50]); they are also used extensively in practice to solve problems such as SAT, where the technique is known as the Davis–Putnam–Logemann–Loveland algorithm. Probably most well known as 'backtracking algorithms', search tree algorithms are also referred to as 'splitting algorithms' in some literature.

The main contribution of fixed-parameter theory to search tree approaches is the consideration of search trees whose depth is bounded by the parameter. Combined with insights on how to find useful and possibly nonobvious parameters, this can lead to search trees that are much smaller than those of naive brute-force searches. Additionally, fixed-parameter theory provides means to provably improve the speed of search tree exploration, particularly by exploiting data reduction rules as examined in Section 2.

3.1. Vertex Cover revisited

The most naive search tree approach for solving vertex cover is to just take one vertex and branch into two cases: either this vertex is in the vertex cover or not. This leads to a search tree of size $O(2^n)$. As we outline in this section, we can do much better than that and obtain a search tree whose depth is upper-bounded by k, giving a size bound of $O(2^k)$. Since usually $k \ll n$, this can draw the problem into the zone of feasibility even for large graphs (as long as k is small).

The basic idea is to find a small subset of the input instance in polynomial time such that at least one element of this subset *must* be part of an optimal solution to the problem. In the case of Vertex Cover, the most simple such subset is any two vertices that are connected by an edge. By definition of the problem, one of these two vertices *must* be part of a solution. Thus, a simple search-tree algorithm to solve Vertex Cover on a graph G proceeds by picking an arbitrary edge $e = \{v, w\}$ and recursively searching for a vertex cover of size $k - 1$ both in $G - v$ and $G - w$.[4] That is, the algorithm *branches* into two subcases knowing one of them must lead to a solution of size at most k—if one such solution should exist.

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

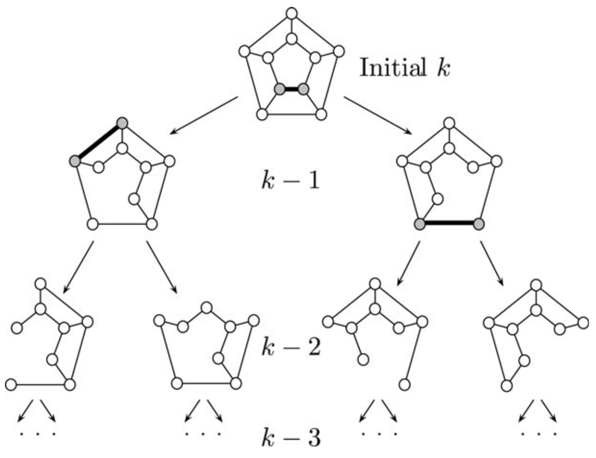

FIGURE 6. Simple search tree for finding a vertex cover of size at most k in a given graph. The size of the tree is upper-bounded by $O(2^k)$.

As shown in Fig. 6, these recursive calls of the simple Vertex Cover algorithm can be visualized as a tree structure. Because the depth of the recursion is upper-bounded by the parameter value and the search always branches into two subcases, the size of this tree is upper-bounded by $O(2^k)$. Note how the size of the tree is independent of the size of the initial input instance and only depends on the value of the parameter k.

The main idea behind fixed-parameter algorithmics is to get the combinatorial explosion as small as possible. For our Vertex Cover example, one can easily achieve a size-$o(2^k)$ search tree by distinguishing more detailed branching cases rather than just picking single endpoints of edges to be in the cover. Note that analogously to the case of data reduction, we are assuming that only *one* minimum solution is sought after.[5]

(1) If there is a vertex of degree one, then put its neighbour into the cover (just as in Reduction Rule VC2 in Section 2.1).
(2) If there is a vertex v of degree two, then either both neighbours of v are in a minimum-size cover or v together with all neighbours of its neighbours.
(3) If there is a vertex v of degree at least three, then either v or all its neighbours are in the cover.

This branching process is recursively repeated until an optimal solution is found. The steps are illustrated in Fig. 7. The correctness of Steps (1) and (3) is rather straightforward to see, but validating (2) needs a little more thought. It is not obvious why the second branch does not only put the vertex v into the cover but also all neighbours of its

[5]Since some graphs can have 2^k minimum-size vertex covers, a size-$o(2^k)$ search tree for enumerating *all* minimum-size vertex covers would require the use of *compact solution representations* as outlined by Damaschke [38] and is beyond the scope of this work.

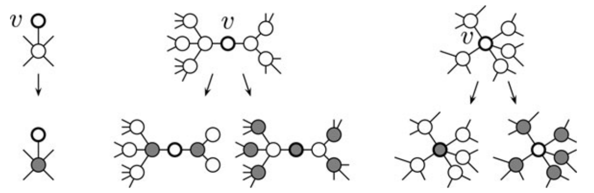

FIGURE 7. Illustration of the branching rules that yield a search tree of size $O(1.47^k)$ for solving vertex cover (the grey vertices are those put into the cover).

neighbours. To show why this is so, assume that there would exist a minimum-size vertex cover containing v and one of its neighbours. Changing this set by replacing v with its second neighbour clearly yields a minimum-size vertex cover as well. This will be found in the second branching case. Hence, if there should be a vertex cover *smaller* than the one that contains both of v's neighbours, then it must contain v and it must *not* contain its neighbours. This implies that all neighbours of v's neighbours have to be part of this vertex cover as well.

In Steps (1) and (3), the search branches into two cases. In Step (2), each branch puts at least two vertices into V'. In Step (3), the first branch puts one vertex into V' and the second branch puts at least three vertices into V'. The recursive construction of the search tree makes it possible to analyse its size with the help of simple recurrences. Solving them with standard mathematical tools, we obtain that if the solution has size k, then the size of the corresponding search tree is bounded from above by $O(1.47^k)$.[6]

The currently 'best' search tree for VERTEX COVER is of size $O(1.28^k)$ [25] and mainly achieved by more extensive case distinguishing than that discussed here. Note, however, that for practical applications it is always concrete implementation and testing that has to decide whether the administrative overhead caused by distinguishing more and more cases pays off. A simpler algorithm with slightly worse search tree size bounds may be preferable. Also, it is conceivable that—instead of having to come up with more and more complicated branching rules—it is possible to find a small set of simple yet generalizing branching schemes that yield good worst-case bounds. Chen *et al.* [25] explore this direction for VERTEX COVER.

In combination with data reduction (see Section 3.4), the use of depth-bounded search trees has proven itself quite useful in practice, allowing to find vertex covers of more than 10 000 vertices in some dense graphs of biological origin [26]. It should also be noted that search trees trivially allow for a parallel implementation: when branching into subcases, each process in a parallel setting can further explore one of these branches with no additional communication required. Cheetham *et al.* [8] expose this in their parallel vertex cover solver to achieve a near-optimum (i.e. linear with the number of processors employed) speedup on multiprocessor systems, solving instances with $k \geq 400$ in mere hours.

3.2. DOMINATING SET revisited

This section introduces a depth-bounded search tree to solve the DOMINATING SET problem (as introduced in Section 2.3) in planar graphs. Again, the size of the tree is upper-bounded by a function of k.[7] However, compared to VERTEX COVER it is much harder to prove the existence of a depth-bounded search tree here.

The problem with finding a search tree in the case of DOMINATING SET is the following: assume that we wanted to argue along the same lines as we did for VERTEX COVER, that is, we identify a set of vertices in the input graph such that at least one of these vertices *must* be in a minimum-size dominating set and then branch on putting parts of it into the dominating set. It turns out that such a set is hard to find: in contrast to VERTEX COVER, for example, it cannot be as simple as the endpoints of an edge—two adjacent vertices can very well be dominated while neither one is part of the DOMINATING SET.

A further challenge with designing a search tree for DOMINATING SET is that we cannot simply remove a vertex v once it has become dominated due to one of its neighbours being put into the dominating set: it might still be necessary to add v into the dominating set in order to dominate other vertices. We can only remove a vertex v once *all* its neighbours are dominated and we no longer need to keep this option open. To circumvent this problem, we consider a more general version of DOMINATING SET [32].

ANNOTATED DOMINATING SET
INPUT: an undirected graph $G = (V, E)$ with vertices coloured black and white and a nonnegative integer k.
TASK: find a subset $S \subseteq V$ with at most$k$ vertices such that every black vertex in V is either contained in S or has at least one neighbour in S.

The strategy for solving DOMINATING SET is to start by colouring all vertices in the input graph black. Then, in each recursion, we take a low-degree black vertex v from the input graph, knowing that v either dominates itself in a minimum solution or it is dominated by one of its neighbours. Being restricted to planar graphs, we can always guarantee the existence of a vertex with degree at most five due to the well-known Euler formula. Unfortunately, however, this vertex need not necessarily be black. The solution to this lies in data reduction!

[6]A more detailed walkthrough of this analysis can be found, e.g. in [3]. For a more refined mathematical analysis of search tree sizes, multivariate recurrences often come into play. Eppstein [51] shows how to treat these.

[7]As already mentioned in Section 2.3, DOMINATING SET is most likely not fixed-parameter tractable [1], and hence we cannot hope for such a tree when solving DOMINATING SET in general graphs.

Alber *et al.* [32] found seven (rather simple) data reduction rules for annotated DOMINATING SET that allow to establish the following lemma.

LEMMA 3.1. *Let G be a planar graph with vertices coloured black and white such that none of the data reduction rules applies to it. Then there exists a black vertex with degree at most seven in G.*

This directly yields a recursive search strategy for (annotated) DOMINATING SET in planar graphs.[8]

THEOREM 3.1. DOMINATING SET in planar graphs can be solved by a search tree whose worst-case size is $O(8^k)$.

3.3. The CENTRE STRING problem

The string problem we study here has applications in coding theory and computational molecular biology [53, 54]. While we refer to it as CENTRE STRING, it is also known as CONSENSUS STRING or CLOSEST STRING.

CENTRE STRING
INPUT: a set of k length-ℓ strings $s_1, \ldots, s_k$ over an alphabet Σ and a nonnegative integer d.
TASK: find a *centre string* s that satisfies $d_H(s, s_i) \leq d$ for all $i = 1, \ldots, k$.

Here, $d_H(s, s_i)$ denotes the Hamming distance between two strings s and s_i, that is, the number of positions where s and s_i differ.

One application scenario where this problem appears is in *primer design* where we try to find a small DNA sequence called *primer* that binds to a set of (longer) target DNA sequences as a starting point for replication of these sequences. How well the primer binds to a sequence is mostly determined by the number of positions in that sequence that hybridize to it. While often done by hand, Stojanovic *et al.* [55] proposed a computational approach for finding a well-binding primer of length ℓ. First, the target sequences are aligned, that is, as many matching positions within the sequences as possible are grouped into columns. Then, a 'sliding window' of length ℓ is moved over this alignment, giving a centre string problem for each window position. Figure 8 illustrates this (see [56] for details).

In this section, we present a fixed-parameter algorithm for CENTRE STRING by Gramm *et al.* [57]; the parameter is the distance d. The central challenge here lies in *finding* a depth-bounded search tree. Once found, the derivation of the upper bound for the search tree size is straightforward. The underlying algorithm is very simple to implement and so far no better one is known.

The main idea behind the algorithm is to maintain a *candidate string* $\hat{s}$ for the centre string and compare it to the strings

[8]Ellis *et al.* [52] generalize Lemma 3.1 to all graphs of bounded genus.

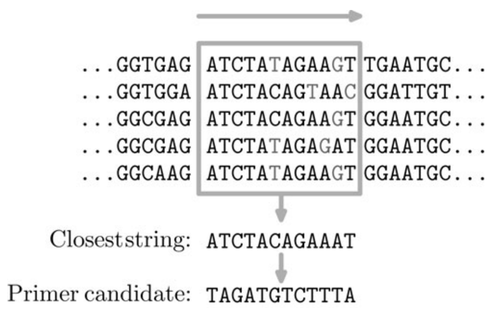
...GGTGAG ATCTATAGAAGT TGAATGC...
...GGTGGA ATCTACAGTAAC GGATTGT...
...GGCGAG ATCTACAGAAGT GGAATGC...
...GGCGAG ATCTATAGAGAT GGAATGC...
...GGCAAG ATCTATAGAAGT GGAATGC...
Closest string: ATCTACAGAAAT
Primer candidate: TAGATGTCTTTA

FIGURE 8. Illustration from Gramm [56] to show how DNA primer design can be achieved by solving centre string instances on length-ℓ windows of aligned DNA sequences. (Note that the primer candidate is not the centre string sought after but its nucleotide-wise complement.).

$s_1, \ldots, s_k$. If $\hat{s}$ differs from some s_i in more than d positions, then we know that $\hat{s}$ needs to be modified in at least one of these positions to match the character that s_i has there. Consider the following observation.

OBSERVATION 3.1. *Let d be a nonnegative integer. If two strings s_i and s_j have a Hamming distance greater than 2d, then there is no string s that has a Hamming distance of at most d to both of s_i and s_j.*

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

We start with an arbitrary string from $\{s_1, \ldots, s_k\}$ as the candidate string $\hat{s}$, knowing that a centre string can differ from it in at most d positions. If $\hat{s}$ already is a valid centre string, then we are done. Otherwise, there exists a string s_i that differs from $\hat{s}$ in more than d positions but at most $2d$. Choosing any $d + 1$ of these positions, we branch into $(d + 1)$ subcases, each subcase modifying a position in $\hat{s}$ to match s_i. This position cannot be changed anymore further down in the search tree (otherwise, it would not have made sense to make it match s_i at that position). Hence, the depth of the search tree is upper-bounded by d, for if we were to go deeper down in the tree, then $\hat{s}$ would differ in more than d positions from the original string we started with. Thus, we obtain the following theorem [57].

THEOREM 3.2. CENTRE STRING *can be solved by exploring a search tree of size $O((d + 1)^d)$.*

This search tree can be combined with the following observation that yields a simple data reduction.

OBSERVATION 3.2. *Given a* CLOSEST STRING *instance with k length-ℓ strings and distance parameter d. If more than $k \times \ell$ character positions are not identical for all k strings, then there is no solution to this instance.*

Combining data reduction and the search tree, we arrive at the following corollary.

Corollary 3.1. Closest String *can be solved in* $O(k \cdot \ell + k \cdot d \cdot (d+1)^d)$ *time*.

Since the term $(d+1)^d$ becomes prohibitively large already for, say, $d = 15$, it might seem as if this result is of pure theoretical interest. However, two things are to be noted in this respect. First, for one of the main applications of Centre String, primer design, d is very small (most often less than four) and an implementation of the search tree algorithm solves corresponding real-world instances in less than a second [57]. Second, a detailed empirical analysis shows that when the algorithm is applied to real-world and random instances, it often beats the proven upper bound by far. The algorithm is also faster than an ILP formulation of Centre String when the input consists of many strings and ℓ is small [57].

This section and the preceding case studies of Vertex Cover and Dominating Set have shown the wide range of occurrences of depth-bounded search trees in fixed-parameter algorithmics. Before making an outlook for this area in Section 3.6, the next two sections mention two useful general techniques when designing fixed-parameter search tree algorithms.

3.4. Interleaving search trees with kernelization

Consider the graph G_i from Fig. 9 for some positive integer i. It is easy to verify that this graph has a minimum vertex cover of size $4i + 1$ (e.g. take all $2i + 1$ vertices in the second row from the top and all $2i$ vertices in the bottommost row). Assume we are using the naive $O(2^k)$ search tree algorithm to solve a Vertex Cover instance $(G_i, 4i + 1)$. None of the Reduction Rules VC1–VC3 from Section 2.1 can be applied to this instance. In a worst-case scenario, the algorithm then first branches for the edges that are not incident to one of the two middle vertices, leading to a search tree of size $O(2^{4i}) = O(8^i)$.

What if we had solved the instance $(G_i, 4i + 1)$ while applying data reduction after every branching? Then, after branching for the first edge (any edge will do), it is easy to verify that an exhaustive application of Reduction Rules VC1–VC3 already solves the instance—the resulting search tree has constant size! This example clearly illustrates the potential power of interleaving kernelization with search trees.

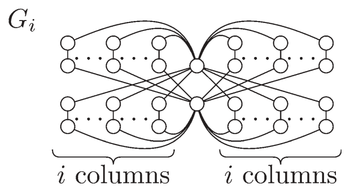

FIGURE 9. Graph to illustrate the importance of interleaving search trees with kernelization.

Intriguingly, the improvement obtained by interleaving search-trees with kernelization is not limited to artificial examples, but yields a *provable* acceleration of the solution finding process. More specifically—provided that a kernelization and a search tree for a fixed-parameter tractable problem are known—Niedermeier and Rossmanith [12] showed that interleaving makes the cost of expanding a node in the search tree constant from an amortized point of view, that is, if the search tree has size $O(\alpha^k)$ then its exploration takes $O(\alpha^k)$ time after an initial run of the kernelization.

While the interleaving technique allows us to amortize the cost of expanding a node in the search tree, the main cost of the algorithm still lies in the base α of the search tree size. The next section discusses an automated approach to improve base α.

3.5. Automated search tree generation and analysis

Many improvements on worst-case search tree size in the realm of fixed-parameter algorithmics stem from extensive case distinguishing (see, e.g. [21, 58])—Section 3.1 has already given a taste of this. Case distinguishing is a tedious and quite error-prone task as the arguments for correctness become more and more complicated and the structures that need to be considered become more complex. Hence, the question arises whether this sort of algorithm design could be automated. The hope is that such an approach would offer the benefits of rapid development and improved upper bounds due to sheer computing power.

In this section, we briefly sketch an approach by Gramm *et al.* [59] that makes use of computers in search tree generation and analysis when dealing with graph modification problems, that is, problems where a graph is to be modified such that the modified graph satisfies certain structural properties.[9] Search tree algorithms for graph modification problems basically consist of a set of branching rules which are usually based on local substructures. The idea behind the automation approach is roughly described as follows.

(1) For a constant integer s, enumerate all 'relevant' subgraphs of size s such that every input instance of the given graph problem has s vertices inducing at least one of the enumerated subgraphs.
(2) For every local substructure that is enumerated in Step (1), check all possible branching rules for this local substructure and select the one corresponding to the *best*, that is, smallest, increase of the search tree size. The

[9]The framework by Gramm *et al.* [59] appears to be the most general approach to automated search tree generation. For satisfiability problems, automated approaches have also been applied by Nikolenko and Sirotkin [60] and Fedin and Kulikov [61].

set of all these best branching rules defines our search tree algorithm.

(3) Determine the worst-case branching rule among the branching rules stored in Step (2), because this branching rule yields a worst-case bound on the search tree size of the generated search tree algorithm.

In order to be computationally feasible, both Steps (1) and (2) usually make use of further problem-specific rules, e.g. to determine input instances which do not need to be considered in the enumeration of Step (1) because they can be solved in polynomial time, simplified due to data reduction rules, or dealt with by use of a clever manually developed branching rule.

By using their automated search tree generation technique, Gramm *et al.* improved a 'hand-crafted' algorithm for the following problem called cluster editing from $O(2.27^k + n^3)$ to $O(1.92^k + n^3)$ for an n-vertex graph.

CLUSTER EDITING
INPUT: an undirected graph G and a nonnegative integer k.
TASK: by deleting and adding at most k edges, transform G into a graph that consists of a disjoint union of cliques.

Recent experiments [62] used the size-2.27^k search tree, however, which is much simpler in concept than the size-1.92^k tree.

3.6. Current research and future challenges

Concluding our discussion of depth-bounded search trees in the fixed-parameter setting, this section mentions some future challenges that we see in this area. Compared to kernelization, much more experimental work have been done in the area of depth-bounded search trees. Besides the already mentioned case studies, depth-bounded search trees have been successfully employed to improve the solvability of a number of fixed-parameter tractable problems.

Besides in fixed-parameter theory, search tree algorithms are studied extensively in the area of artificial intelligence and heuristic state space search. There, the key to speedups are *admissible heuristic evaluation functions*, which give quickly a lower bound on the distance to the goal, e.g. the number of vertices that still need to be put into the vertex cover. The reason that admissible heuristics are rarely considered by the FPT community[10] is that they typically cannot improve the asymptotic running time [64]. Still, the speedups obtained in practice can be quite pronounced, as demonstrated for vertex cover [65]. Clearly, more theoretical work combining the results on search tree algorithms from the fields of heuristic search and fixed-parameter algorithms is desirable.

As with kernelizations, algorithmic developments outside the fixed-parameter setting can make use of the insights that have been gained in the development of depth-bounded search trees in a fixed-parameter setting. A recent example for this is the MINIMUM QUARTET INCONSISTENCY problem arising in the construction of evolutionary trees. Here, an algorithm that uses depth-bounded search trees was developed by Gramm and Niedermeier [63]. Their insight was recently used by Wu *et al.* [66] to develop a (non-parameterized) faster algorithm for this problem.

Concluding this section, depth-bounded search trees with clever branching rules are certainly one of the first approaches to try when solving hard fixed-parameter tractable problems in practice. Besides the challenge to come up with better branching rules that yield smaller worst-case bounds on the size of the search tree, we also see some more general future challenges in this area that are of interest to the practitioner.

- For VERTEX COVER, we have already mentioned a new approach by Chen *et al.* [25] that replaces complicated branching rules by a smaller set of simple branching schemes that yield the same worst-case bounds. Can this 'reverse engineering approach' (i.e. moving the complicated branching from the actual algorithm description into its analysis) be applied to other problems as well? What are the best bounds on search-tree size achievable with as few different branching rules as possible? Here, recent progress with the analysis of search tree algorithms using multivariate recurrences [51] might help: with this method, it was shown that some simple algorithms perform in fact much better than previously proved [67]. Also, new algorithms were developed guided by the new analysis methods [67]; however, there is no practical experience yet with these approaches.
- The techniques for the automated generation of search trees should be explored beyond CLUSTER EDITING and extended to help find new data reduction rules and kernelization schemes. A first step could be the adaption to weighted CLUSTER EDITING, which has recently gained some interest due to applications in computational biology [68].
- We need to develop a deeper understanding of the order and frequency in which certain branching rules and data reduction attempts are made in order to improve practical efficiency ('make the common case first').
- Since most search tree branchings for graph problems are based on *local* structure, it can happen that certain problem solutions are considered more than once deeper down in the tree. What are computationally efficient ways to avoid such 'double explorations'? Certainly this has the potential to boost efficiency, but few attempts have been made in this direction so far.

[10]See [63] for a counterexample.

4. ITERATIVE COMPRESSION

Of the techniques we survey, iterative compression is by far the youngest, appearing first in a work by Reed *et al.* in 2004 [69]. Although not quite as generally employable as data reduction or search trees, it appears to be applicable to a wide range of problems, and it has already led to significant breakthroughs in showing FPT results.

For instance, the GRAPH BIPARTIZATION problem, that is, the task of finding a minimum set of vertices whose deletion destroys all odd-length cycles, has been shown fixed-parameter tractable with respect to the number of the deleted vertices by means of iterative compression [69]. For years this had been a central open problem in parameterized complexity [47]. Moreover, ongoing research indicates that the corresponding algorithm is competitive in practice [70]. Although currently only a handful of results are known, iterative compression seems promising for a wide range of parameterized problems where the parameter is the size of the solution set.

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

DEFINITION 4.1. *A* compression routine *is an algorithm that, given a problem instance and a solution of size k, either calculates a smaller solution or proves that the given solution is of minimum size*.

Using this routine, one finds an optimal solution to a problem by inductively building up the problem structure and iteratively compressing intermediate solutions. (This is outlined in more detail in Section 4.1 for our running example VERTEX COVER.) The main point is that if the compression routine is a fixed-parameter algorithm, then so is the whole algorithm.

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

However, while the mode of the use of the compression routine is usually straightforward, finding the compression routine itself is not. It is not even clear that a compression routine with interesting running time exists even when we already know a problem to be fixed-parameter tractable. Therefore, the art lies in designing the compression routine.

4.1. Iterative compression for vertex cover

As an introductory example, this section describes an algorithm for VERTEX COVER based on iterative compression. The global structure of the algorithm is as follows.

(1) Set $V' \leftarrow \emptyset$ and $C \leftarrow \emptyset$.
(2) For each vertex $v \in V$, set $V' \leftarrow V' \cup \{v\}$ and $C \leftarrow C \cup \{v\}$ and call the compression routine for the vertex cover instance $(G[V'],C)$, where $G[V']$ is the subgraph induced in G by V'.
(3) Output C.

Here, the compression routine takes a graph G and a vertex cover C for G, and returns a smaller vertex cover for G if there is one; otherwise, it returns C unchanged. Therefore, it is a loop invariant in Step (2) that C is a minimum-size vertex cover for $G[V']$. Since eventually $V' = V$, we obtain an optimal solution for G once the algorithm outputs C.

It remains to implement the compression routine. For this, consider a smaller vertex cover C' as a *modification* of the larger vertex cover C. This modification retains some vertices $Y \subseteq C$ while the other vertices $S := C \setminus Y$ are replaced by $|S| - 1$ new vertices from $V \setminus C$. The idea is to try by brute force all $2^{|C|}$ partitions of C into such sets Y and S (Fig. 10a shows an example). For each such partition, the vertices from Y are immediately deleted since we already decided to take them into the vertex cover, which covers all their adjacent edges (Fig. 10b). In the resulting instance $G' := G[V \setminus Y]$, it remains to find an optimal vertex cover that is disjoint from S. This is easy: since we decided to take no vertex from S into the vertex cover, we have to take that endpoint of each edge which is not in S; if both endpoints of some edge are in S, then this choice of S cannot lead to a vertex cover C' with $S \cap C' = \emptyset$. Note that at least one endpoint of each edge in G' is in S, since S is a vertex cover for G'. Therefore, we can quickly find an optimal vertex cover for G' that is disjoint from S by taking every vertex that is not in S and has degree greater than zero. Together with Y, we obtain a new vertex cover C' for G (Fig. 10c). For one choice of S and Y, this can be done in $O(m)$ time, leading to $O(2^{|C|}m) = O(2^k m)$ time overall required for a call of the compression routine for vertex cover. With n iterations of the algorithm, we get an algorithm for vertex cover running in $O(2^k\, mn)$ time.

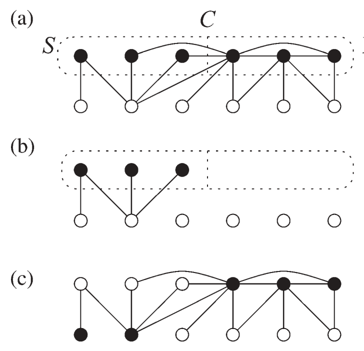

FIGURE 10. Example for the compression routine for solving vertex cover (cover vertices are marked black).

4.2. Iterative compression for feedback set problems

Nearly all of the currently known iterative compression algorithms solve *feedback set problems* in graphs, that is, problems where one wishes to destroy certain cycles in the graph by deleting at most k vertices or edges (see [71] for a survey on feedback set problems).

- GRAPH BIPARTIZATION: destroy all odd cycles by deleting vertices [69, 70].
- EDGE BIPARTIZATION: destroy all odd cycles by deleting edges [72].
- FEEDBACK VERTEX SET: destroy all cycles by deleting vertices [72, 73].
- FEEDBACK VERTEX SET in tournaments: destroy all cycles in a tournament (that is, a directed graph where between any two vertices exactly one directed edge is present) by deleting vertices [74].
- CHORDAL DELETION: destroy all chordless cycles (that is, induced cycles of length at least four) by deleting edges [75].

How do the compression routines work in these cases? Interestingly, most start with the same opening move, namely forcing the new solution to be disjoint from the known one (just as in our VERTEX COVER example). After that, however, they widely diverge.

- For GRAPH BIPARTIZATION and EDGE BIPARTIZATION, it is possible to reduce the remaining task to finding a vertex (respectively edge) cut set, a task which can be accomplished in polynomial time by maximum flow techniques. We describe this in detail for EDGE BIPARTIZATION in Section 4.3.
- For FEEDBACK VERTEX SET, data reduction rules allow to shrink the remaining instance so it can be solved by brute force.
- For FEEDBACK VERTEX SET in tournaments, a data reduction constrains the possible solution such that it can be found with a polynomial-time LONGEST INCREASING SUBSEQUENCE algorithm.
- For CHORDAL DELETION, the graph is reduced until it has bounded treewidth, a property that allows fixed-parameter tractable algorithms.

4.3. Iterative compression for EDGE BIPARTIZATION

We now present in detail an iterative compression algorithm for EDGE BIPARTIZATION [72].

EDGE BIPARTIZATION
Input: an undirected graph G and a nonnegative integer k.
Task: find a size-k subset X of edges such that each odd cycle in G contains at least one edge from X.

Being the only fixed-parameter algorithm known for EDGE BIPARTIZATION, this algorithm is similar to the iterative

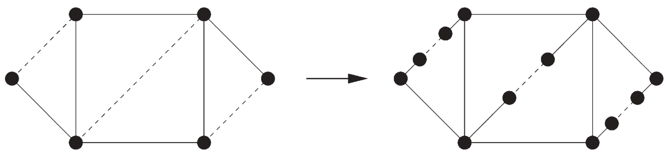

FIGURE 11. Graph (left) with edge bipartization set X (dashed lines). To be able to assume without the loss of generality that a bipartization set smaller than X is disjoint from X, we subdivide each edge in X by two vertices and choose the middle edge from each thus generated path as the new X (right).

compression algorithm that Reed *et al.* [69] gave for GRAPH BIPARTIZATION, yet simpler.

The global algorithm structure is almost the same as for VERTEX COVER, except that we add edge-by-edge instead of vertex-by-vertex. It is helpful for the compression routine to assume that an edge bipartization set smaller than the given edge bipartization set X is disjoint from X. Unlike with VERTEX COVER, we do not need brute force: this assumption can be made without loss of generality by applying a simple input transformation (see Fig. 11). Since this transformation preserves the parity of the length of cycles, it is easy to see that the thus transformed graph has an edge bipartization set with i edges iff the original graph has an edge bipartization set with i edges. Moreover, for each edge bipartization set Y for the transformed graph there is an edge bipartization set of the same size that is disjoint from X, which can be obtained by replacing every edge in $Y \cap X$ by one of its two adjacent edges.

The idea for the compression routine is to compare the two-colourings induced by the known bipartization set and the (yet unknown) compressed solution (Fig. 12a and b) and mark a vertex black when the two colourings coincide, or white when they differ (Fig. 12c). The key observation is then that the two bipartization sets together form an edge cut between the black and the white vertices, that is, removing them destroys all paths from a black to a white vertex.

The following simple definition is the only remaining prerequisite for the central lemma for the EDGE BIPARTIZATION compression routine.

DEFINITION 4.2. *Let $G = (V, E)$ be a graph and let $X \subseteq E$ be a set of edges, with $V(X)$ denoting the set $\cup_{\{u,v\} \in X} \{u, v\}$ of their endpoints. A mapping Φ: $V(X) \to \{A, B\}$ is called valid partition of $V(X)$ if for each $\{u, v\} \to X$, we have $\Phi(u) \neq \Phi(v)$.*

LEMMA 4.1. *Consider a graph $G = (V, E)$ and an edge bipartization set X for G without redundant edges. For a set of edges $Y \subseteq E$ with $X \cap Y = \emptyset$, the following are equivalent:*

(1) *Y is an edge bipartization set for G.*
(2) *There is a valid partition Φ of $V(X)$ such that Y is an edge cut in $G \backslash X$ between $A_\Phi := \Phi^{-1}(A)$ and $B_\Phi := \Phi^{-1}(B)$.*

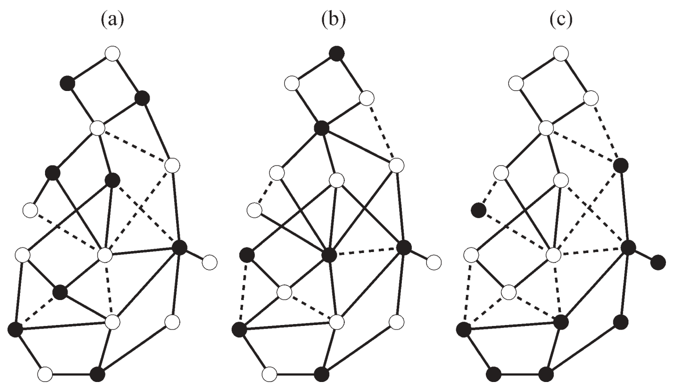

FIGURE 12. Comparing disjoint edge bipartization sets.

How can we make use of Lemma 4.1 to obtain the compression routine that, given a graph and a with respect to set inclusion minimal edge bipartization set X of size k', either computes a smaller edge bipartization set Y in $O(2^{k'} \cdot k'm)$ time or proves that no such Y exists? First, we apply the input transformation from Fig. 11, which allows us to assume the prerequisite of Lemma 4.1 that $Y \cap X = \emptyset$. We then enumerate all $2^{k'}$ valid partitions Φ of $V(X)$ and determine a minimum-size edge cut between A_Φ and B_Φ until we find an edge cut Y of size $k' - 1$. This is illustrated in Fig. 13: on the left side, we have a graph with an edge bipartization set X (dashed lines) and a valid partition for $V(X)$. A smaller edge bipartization set is obtained as minimum-size edge cut between the black and the white vertices (right). Each of the Minimum Cut instances can individually be solved in $O(k'm)$ time with the Ford–Fulkerson algorithm that goes through k' rounds, each time finding a flow augmenting path [76]. By Lemma 4.2, Y is an edge bipartization set; furthermore, if no such Y is found, we know that k' is of minimum size.

Analogous to the procedure we used to solve Vertex Cover via iterative compression, using the compression routine m times and each time compressing a bipartization set of size $k' \leq k + 1$, we obtain the following theorem.

Theorem 4.1. Edge Bipartization *can be solved in* $O(2^k \cdot m^2)$ *time*.

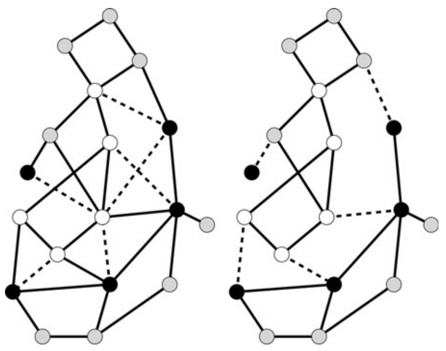

FIGURE 13. A valid partition leading to a compressed solution (black and white: value of valid partition; grey: not in domain of valid partition).

4.4. Further remarks

So far, little research has been done on the range of applicability of iterative compression (see [77] for some results). In particular, little is known about characterizing problems amenable to solution compression, that is, those for which we can find a compression routine. For Vertex Cover this was trivial, while the Edge Bipartization compression routine is not quite as easy to find.

As Dehne *et al.* [78] point out, iterative compression can also be used as a tool to obtain kernelizations.

First experimental results for iterative compression-based algorithms appear quite encouraging. An implementation of the Graph Bipartization algorithm, improved by heuristics, can solve all problems from a testbed from computational biology within minutes, whereas established methods are only able to solve about half of the instances within reasonable time [70]. Further, an iterative compression-based approach for the Balanced Subgraph problem, which generalizes Edge Bipartization, is able to find optimal solutions to instances for which previously only approximate solutions could be given [79].

A particular advantage of iterative compression is the flexibility in the use of the compression routine. An appealing mode for practical purposes is to start with a suboptimal initial solution (found by a heuristic, by an approximation algorithm or even manually supplied), and then to repeatedly compress this solution. We can abort the compression when either the solution is 'good enough' or we are not willing to invest any more calculation time; when waiting long enough, we will eventually get an optimal solution. Although the theoretical bound on the running time of this approach is worse than with an inductive buildup of the input instance, experiments with Graph Bipartization show that it is not substantially slower in practice [70].

4.5. Future challenges

Besides the general goal to find more applications of iterative compression, in particular outside the realm of feedback set problems, we mention some specific challenges.

- The Minimum 2CNF Deletion problem (given a formula in 2CNF, find a minimum number of clauses to delete such that the formula becomes satisfiable) is a natural generalization of Graph Bipartization; as such, it would be interesting to try to find a fixed-parameter algorithm for Minimum 2CNF Deletion based on iterative compression (see also [47]). Currently, the problem is not even known to be fixed-parameter tractable.

- Since the technique seems to work so well for feedback set problems, it is tempting to try to apply it to DIRECTED FEEDBACK SET, for which famously no fixed-parameter algorithm is known. Because this is probably difficult, the subclasses of directed graphs could be examined first, for example, generalizing tournaments for which an iterative compression algorithm is already known [74].
- There is a $2^k \cdot n^{O(1)}$ time algorithm for DIRECTED FEEDBACK VERTEX SET in tournaments using iterative compression [74]. Recently, the related DIRECTED FEEDBACK EDGE SET in tournaments has received increased attention, for example, due to applications in rank aggregation [80]. Can one find, using iterative compression, a $2^k \cdot n^{O(1)}$ time algorithm for DIRECTED FEEDBACK EDGE SET in tournaments?
- It is known that the CLIQUE COVER problem is fixed-parameter tractable, since it has an exponential-size kernel [37]; however, no fixed-parameter algorithm beyond the brute-force exploration of the kernel is known. Iterative compression might lead to a more practical upper bound on the running time here.

5. CONCLUSION

In this survey, we have chosen to concentrate on three key techniques of fixed-parameter algorithmics. Some further techniques with potential for practical application are the following.

- *Colour-coding* [81] is an interesting technique that helps to find small substructures in graphs. A concrete application in computational molecular biology can be found in [82]; experimental and algorithm engineering results are reported in [83].
- *Tree decompositions* of graphs are a very fundamental tool of modern graph theory [5]. For graphs of bounded treewidth, the combinatorial explosion can often be confined to the parameter 'treewidth' (e.g. see [84] for an efficient dynamic programming algorithm for dominating set).
- Exponential-time *dynamic programming* is a well-known technique that has proven useful also in the context of fixed-parameter algorithmics (see [85–87] for recent examples with more or less practical flavour).
- Also *enumerative techniques* can be useful in fixed-parameter solutions, e.g. in the context of finding longest arc-preserving subsequences motivated by RNA structure comparison in biology [88]. In addition, note that the question of enumerating all solutions of a particular parameterized problem is still in its infancy [88, 89].

We hope to have fulfilled the promise made in the introduction, namely showing that the concept of FPT belongs into the toolkit of all algorithm designers, and to have provided the reader with enough material to raise an interest in pursuing further studies of fixed-parameter algorithmics, we recommend to study the monographs [1–3] for a more thorough treatment of FPT algorithmics. Further, an upcoming thesis [90] focuses on algorithm engineering issues of FPT methods, that is, the systematic use of implementation and experiments to design and improve FPT algorithms.

ACKNOWLEDGEMENTS

This work was supported by the Deutsche Forschungsgemeinschaft, Emmy Noether research group PIAF (fixed-parameter algorithms), NI 369/4 (Falk Hüffner) and the Deutsche Telekom Stiftung (Sebastian Wernicke). The authors' FPT research involved collaboration with the following (partially former) PhD students: Jochen Alber, Nadja Betzler, Michael Dom, Jens Gramm, Jiong Guo and Hannes Moser.

REFERENCES

[1] Downey, R.G. and Fellows, M.R. (1999) *Parameterized Complexity*. Springer.

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

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

[4] Fellows, M.R. (2002) Parameterized complexity: The main ideas and connections to practical computing. *Experimental Algorithmics: From Algorithm Design to Robust and Efficient Software*, Lecture Notes in Computer Science, Vol. 2547, 51–77. Springer.

[5] Diestel, R. (2005) *Graph Theory* (3rd ed). Springer.

[6] Park, K. and Lee, H. (2001) On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets. *Proc. 8th COMM*, 15–26. ACM.

[7] Chesler, E.J. *et al*. (2005) Complex trait analysis of gene expression uncovers polygenic and pleiotropic networks that modulate nervous system function. *Nat. Genet.*, **37**, 233–242.

[8] Cheetham, J., Dehne, F.K.H.A., Rau-Chaplin, A., Stege, U. and Taillon, P.J. (2003) Solving large FPT problems on coarse-grained parallel machines. *J. Comput. Syst. Sci.*, **67**, 691–706.

[9] Mecke, S. and Wagner, D. (2004) Solving geometric covering problems by data reduction. *Proc. 12th ESA*, Lecture Notes in Computer Science, Vol. 3221, 760–771. Springer.

[10] Weihe, K. (1998) Covering trains by stations or the power of data reduction. *Proc. Algorithms and Experiments (ALEX-'98)*, pp. 1–8.

[11] Quine, W.V. (1952) The problem of simplifying truth functions. *Am. Math. Mon.*, **59**, 512–531.

[12] Niedermeier, R. and Rossmanith, P. (2000) A general method to speed up fixed-parameter-tractable algorithms. *Inf. Process. Lett.*, **73**, 125–129.

[13] Bixby, R.E. (2002) Solving real-world linear programs: a decade and more of progress. *Oper. Res.*, **50**, 3–15.

[14] Cai, L., Chen, J., Downey, R. and Fellows, M.R. (1997) Advice classes of parameterized tractability. *Ann. Pure Appl. Log.*, **84**, 119–138.

[15] Fernau, H. (2005) *Parameterized Algorithmics: A Graph-Theoretic Approach*. Habilitationsschrift. Wilhelm-Schickard-Institut für Informatik, Universität Tübingen, Germany.

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

[17] Abu-Khzam, F.N., Collins, R.L., Fellows, M.R., Langston, M. A., Suters, W.H. and Symons, C. T. (2004) Kernelization algorithms for the vertex cover problem: theory and experiments. In Arge, L., Italiano, G.F. and Sedgewick, R. (eds), *Proc. 6th Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmcs and Combinatorics (ALENEX-04)*, pp. 62–69. SIAM.

[18] Abu-Khzam, F.N., Langston, M.A. and Suters, W.H. (2005) Fast, effective vertex cover kernelization: a tale of two algorithms. *Proc. 3rd ACS/IEEE Int. Conf. Computer Systems and Applications (AICCSA-'05)*, 16 pp. ACS/IEEE.

[19] Díaz, J., Petit, J. and Thilikos, D.M. (2006) Kernels for the vertex cover problem on the preferred attachment model. *Proc. 5th WEA*, Lecture Notes in Computer Science, Vol. 4007, 231–240. Springer.

[20] Nemhauser, G.L. and Trotter, J.L.E. (1975) Vertex packing: structural properties and algorithms. *Math. Program.*, **8**, 232–248.

[21] Chen, J., Kanj, I.A. and Jia, W. (2001) Vertex Cover: further observations and further improvements. *J. Algorithms*, **41**, 280–301.

[22] Khot, S. and Regev, O. (2003) Vertex Cover might be hard to approximate to within $2 - \epsilon$. *Proc. 18th IEEE Annual Conf. Computational Complexity*, pp. 379–386. IEEE.

[23] Chlebík, M. and Chlebíková, J. (2004) Improvement of Nemhauser–Trotter theorem and its applications in parameterized complexity. *Proc. 9th SWAT*, Lecture Notes in Computer Science, Vol. 3111, pp. 174–186. Springer.

[24] Chor, B., Fellows, M.R. and Juedes, D.W. (2004) Linear kernels in linear time, or how to save k colors in $O(n^2)$ steps. *Proc. 30th WG*, Lecture Notes in Computer Science, Vol. 3353, pp. 257–269. Springer.

[25] Chen, J., Kanj, I.A. and Xia, G. (2006) Improved parameterized upper bounds for vertex cover. *Proc. 31st MFCS*, Lecture Notes in Computer Science, Vol. 4162, pp. 238–249. Springer.

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

[27] Kratsch, D. (1998) Algorithms. In Haynes, T.W., Hedetniemi, S.T. and Slater, P.J. (eds), *Domination in Graphs: Advanced Topics*, chapter 8, pp. 191–231. Marcel Dekker.

[28] Roberts, F.S. (1979) *Graph Theory and Its Applications to Problems of Society*. SIAM Press, Odyssey Press.

[29] Valente, T.W., Hoffman, B.R., Ritt-Olson, A., Lichtman, K. and Johnson, C.A. (2003) Strategies on peer-led tobacco prevention programs in schools. *Am. J. Public Health*, **93**, 1837–1843.

[30] Alber, J., Fellows, M.R. and Niedermeier, R. (2004) Polynomial time data reduction for dominating set. *J. ACM*, **51**, 363–384.

[31] Alber, J., Dorn, B. and Niedermeier, R. (2006) A general data reduction scheme for domination in graphs. *Proc. 32nd SOFSEM*, Lecture Notes in Computer Science, Vol. 3831, pp. 137–147. Springer.

[32] Alber, J., Fan, H., Fellows, M.R., Fernau, H., Niedermeier, R., Rosamond, F. and Stege, U. (2005) A refined search tree technique for Dominating Set on planar graphs. *J. Comput. Syst. Sci.*, **71**, 385–405.

[33] Chen, J., Fernau, H., Kanj, I.A. and Xia, G. (2005) Parametric duality: Kernel sizes & algorithmics. *Proc. 22nd STACS*, Lecture Notes in Computer Science, Vol. 3404, pp. 269–280. Springer. Long version to appear in *SIAM J. Comput.*

[34] Fomin, F.V. and Thilikos, D.M. (2004) Fast parameterized algorithms for graphs on surfaces: linear kernel and exponential speed-up. *Proc. 31st ICALP*, Lecture Notes in Computer Science, Vol. 3142, pp. 581–592. Springer.

[35] Alber, J., Betzler, N. and Niedermeier, R. (2006) Experiments on data reduction for optimal domination in networks. *Ann. Oper. Res.*, **146**, 105–117.

[36] Alber, J. (2003) Exact algorithms for NP-hard problems on networks: design, analysis, and implementation. PhD Thesis WSI für Informatik, Universität Tübingen, Germany.

[37] Gramm, J., Guo, J., Hüffner, F. and Niedermeier, R. (2006) Data reduction, exact, and heuristic algorithms for clique cover. *Proc. 8th ALENEX*, 86–94. SIAM. Long version to appear in *ACM J. Exp. Algorithmics.*

[38] Damaschke, P. (2006) Parameterized enumeration, transversals, and imperfect phylogeny reconstruction. *Theor. Comput. Sci.*, **351**, 337–350.

[39] Prieto, E. (2005) Systematic kernelization in FPT algorithm design. PhD Thesis University of Newcastle, Australia.

[40] Estivill-Castro, V., Fellows, M.R., Langston, M.A. and Rosamond, F.A. (2005) FPT is P-time extremal structure I. *Proc. 1st ACiD*, Texts in Algorithmics, Vol. 4, pp. 1–41. College Publications.

[41] Guo, J. and Niedermeier, R. (2007) Linear problem kernels for NP-hard problems on planar graphs. *Proc. 34th ICALP*, Lecture Notes in Computer Science, Vol. 4596, pp. 375–386. Springer.

[42] Kulikov, A.S. (2005) Automated generation of simplification rules for SAT and MAXSAT. *Proc. 8th SAT*, Lecture Notes in Computer Science, Vol. 3569, pp. 430–436. Springer.

[43] Guo, J. and Niedermeier, R. (2005) Fixed-parameter tractability and data reduction for multicut in trees. *Networks*, **46**, 124–135.

[44] Burrage, K., Estivill-Castro, V., Fellows, M.R., Langston, M.A., Mac, S. and Rosamond, F.A. (2006) The undirected feedback

vertex set problem has a *poly*(k) kernel. *Proc. 2nd IWPEC*, Lecture Notes in Computer Science, Vol. 4169, pp. 192–202. Springer.

[45] Bodlaender, H.L. (2007) A cubic kernel for feedback vertex set. *Proc. 24th STACS*, Lecture Notes in Computer Science, Vol. 4393, pp. 320–331. Springer.

[46] Niedermeier, R. and Rossmanith, P. (2003) An efficient fixed-parameter algorithm for 3-Hitting Set. *J. Discret. Algorithms*, **1**, 89–102.

[47] Mahajan, M. and Raman, V. (1999) Parameterizing above guaranteed values: MaxSat and MaxCut. *J. Algorithms*, **31**, 335–354.

[48] Davis, M., Logemann, G. and Loveland, D.W. (1962) A machine program for theorem-proving. *Commun. ACM*, **5**, 394–397.

[49] Mehlhorn, K. (1984) Data Structures and Algorithms, Volume 2: NP-Completeness and Graph Algorithms. In *EATCS Monographs on Theoretical Computer Science*. Springer.

[50] Drori, L. and Peleg, D. (2002) Faster exact solutions for some NP-hard problems. *Theor. Comput. Sci.*, **287**, 473–499.

[51] Eppstein, D. (2006) Quasiconvex analysis of multivariate recurrence equations for backtracking algorithms. *ACM Trans. Algorithms*, **2**, 492–509.

[52] Ellis, J., Fan, H. and Fellows, M.R. (2004) The dominating set problem is fixed parameter tractable for graphs of bounded genus. *J. Algorithms*, **52**, 152–168.

[53] Cohen, G.D., Karpovsky, M.G., Mattson, H.F., Jr. and Schatz, J.R. (1985) Covering radius—survey and recent results. *IEEE Trans. Inf. Theory*, **31**, 328–343.

[54] Pevzner, P.A. (2000) *Computational Molecular Biology: An Algorithmic Approach*. MIT Press.

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

[56] Gramm, J. (2003) Fixed-parameter algorithms for the consensus analysis of genomic sequences. PhD Thesis WSI für Informatik, Universitäat Tübingen, Germany.

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

[58] Fernau, H. and Niedermeier, R. (2001) An efficient, exact algorithm for constraint bipartite vertex cover. *J. Algorithms*, **38**, 374–410.

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

[60] Nikolenko, S.I. and Sirotkin, A.V. (2003) Worst-case upper bounds for SAT: automated proof. *15th European Summer School in Logic Language and Information* (*ESSLLI 2003*).

[61] Fedin, S.S. and Kulikov, A.S. (2006) Automated proofs of upper bounds on the running time of splitting algorithms. *J. Math. Sci.*, **134**, 2383–2391.

[62] Dehne, F.K.H. A., Langston, M.A., Luo, X., Pitre, S., Shaw, P. and Zhang, Y. (2006) The cluster editing problem: implementations and experiments. *Proc. 2nd IWPEC*, Lecture Notes in Computer Science, Vol. 4169, pp. 13–24. Springer.

[63] Gramm, J. and Niedermeier, R. (2003) A fixedparameter algorithm for minimum quartet inconsistency. *J. Comput. Syst. Sci.*, **67**, 723–741.

[64] Korf, R.E., Reid, M. and Edelkamp, S. (2001) Time complexity of iterative-deepening-A*. *Artif. Intell.*, **129**, 199–218.

[65] Felner, A., Korf, R.E. and Hanan, S. (2004) Additive pattern database heuristics. *J. Artif. Intell. Res.*, **21**, 1–39.

[66] Wu, G., You, J.-H. and Lin, G. (2005) A lookahead branch-and-bound algorithm for the maximum quartet consistency problem. *Proc. 5th WABI*, Lecture Notes in Computer Science, Vol. 3692, pp. 65–76. Springer.

[67] Fomin, F.V., Grandoni, F. and Kratsch, D. (2005) Some new techniques in design and analysis of exact (exponential) algorithms. *Bull. EATCS*, **87**, 47–77.

[68] Rahmann, S., Wittkop, T., Baumbach, J., Martin, M., Truß, A. and Böcker, S. (2007) Exact and heuristic algorithms for weighted cluster editing. *Proc. 6th CSB*. To appear.

[69] Reed, B., Smith, K. and Vetta, A. (2004) Finding odd cycle transversals. *Oper. Res. Lett.*, **32**, 299–301.

[70] Hüffner, F. (2005) Algorithm engineering for optimal graph bipartization. *Proc. 4th WEA*, Lecture Notes in Computer Science, Vol. 3503, pp. 240–252. Springer.

[71] Festa, P., Pardalos, P.M. and Resende, M.G.C. (1999) *Handbook of Combinatorial Optimization*. Kluwer.

[72] Guo, J., Gramm, J., Hüffner, F., Niedermeier, R. and Wernicke, S. (2006) Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *J. Comput. Syst. Sci.*, **72**, 1386–1396.

[73] Dehne, F.K.H. A., Fellows, M.R., Langston, M.A., Rosamond, F.A. and Stevens, K. (2005) An $O(2^{O(k)}n^3)$ FPT algorithm for the undirected feedback vertex set problem. *Proc. 11th COCOON*, Lecture Notes in Computer Science, Vol. 3595, pp. 859–869. Springer.

[74] Dom, M., Guo, J., Hüffner, F., Niedermeier, R. and Truß, A. (2006) Fixed-parameter tractability results for feedback set problems in tournaments. *Proc. 6th CIAC*, Lecture Notes in Computer Science, Vol. 3998, pp. 321–332. Springer.

[75] Marx, D. (2006) Chordal deletion is fixed-parameter tractable. *Proc. 32nd WG*, Lecture Notes in Computer Science, Vol. 4271, pp. 37–48. Springer.

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

[77] Guo, J. (2006) Algorithm design techniques for parameterized graph modification problems. PhD Thesis Institut füur Informatik, Friedrich-Schiller-Universitäat Jena, Germany.

[78] Dehne, F., Fellows, M.R., Rosamond, F.A. and Shaw, P. (2004) Greedy localization, iterative compression, and modeled crown reductions: new FPT techniques, an improved algorithm for set splitting, and a novel 2 k kernelization for Vertex *Cover*. *Proc.*

1st IWPEC, Lecture Notes in Computer Science, Vol. 3162, pp. 271–280. Springer.

[79] Hüffner, F., Betzler, N. and Niedermeier, R. (2007) Optimal edge deletions for signed graph balancing. *Proc. 6th Workshop on Experimental Algorithms* (*WEA '07*), Lecture Notes in Computer Science, Vol. 4525, pp. 297–310. Springer.

[80] Ailon, N., Charikar, M. and Newman, A. (2005) Aggregating inconsistent information: ranking and clustering. *Proc. 37th STOC*, pp. 684–693. ACM.

[81] Alon, N., Yuster, R. and Zwick, U. (1995) Color-coding. *J. ACM*, **42**, 844–856.

[82] Scott, J., Ideker, T., Karp, R.M. and Sharan, R. (2006) Efficient algorithms for detecting signaling pathways in protein interaction networks. *J. Comput. Biol.*, **13**, 133–144.

[83] Hüffner, F., Wernicke, S. and Zichner, T. (2007) Algorithm engineering for color-coding to facilitate signaling pathway detection. *Proc. 5th APBC*, Advances in Bioinformatics and Computational Biology, Vol. 5, pp. 277–286. Imperial College Press.

[84] Alber, J., Bodlaender, H.L., Fernau, H., Kloks, T. and Niedermeier, R. (2002) Fixed parameter algorithms for dominating set and related problems on planar graphs. *Algorithmica*, **33**, 461–493.

[85] Guo, J. and Niedermeier, R. (2006) A fixed-parameter tractability result for multicommodity demand flow in trees. *Inf. Process. Lett.*, **97**, 109–114.

[86] Guo, J. and Niedermeier, R. (2006) Exact algorithms and applications for tree-like weighted set cover. *J. Discret. Algorithms*, **4**, 608–622.

[87] Fuchs, B., Kern, W., Mölle, D., Richter, S., Rossmanith, P. and Wang, X. (2006) Dynamic programming for minimum Steiner trees. *Theory Comput. Syst.* To appear.

[88] Alber, J., Gramm, J., Guo, J. and Niedermeier, R. (2004) Computing the similarity of two sequences with nested arc annotations. *Theor. Comput. Sci.*, **312**, 337–358.

[89] Fernau, H. (2002) On parameterized enumeration. *Proc. 8th COCOON*, Lecture Notes in Computer Science, Vol. 2383, pp. 564–573. Springer.

[90] Hüffner, F. (2007) Algorithm engineering for parameterized approaches to hard graph problems. PhD Thesis. Institut für Informatik, Friedrich-Schiller-Universität Jena, Germany. In preparation.