Advance Access publication on June 18, 2007 doi:10.1093/comjnl/bxm038

An Overview of Techniques for Designing Parameterized Algorithms

Christian Sloper and Jan Arne Telle*

Department of Informatics, University of Bergen, Bergen, Norway
**Corresponding author: telle@ii.uib.no*

A survey of the most important and general techniques in parameterized algorithm design is given. Each technique is explained with a meta-algorithm, its use is illustrated by examples, and it is placed in a taxonomy under the four main headings of branching, kernelization, induction and win/win.

Keywords: Parameterized algorithms; fixed-parameter tractability

Received 8 August 2006; revised 13 November 2006

1. INTRODUCTION

The field of parameterized algorithms continues to grow. It is a sign of its success that in the literature today there exists over 20 differently named techniques for designing parameterized algorithms.[1] Many parameterized algorithms build on the same ideas, and as a problem solver it is important to be familiar with these general themes and ideas. Aside from several survey articles [6, 9], there have been at least two comprehensive texts written on parameterized algorithm design: Downey and Fellows's seminal book 'Parameterized Complexity' [1] and the more recent 'Invitation to Fixed-Parameter Algorithms' [2] by Niedermeier. The first of these books came very early in the history of parameterized complexity, and does therefore not include newer ideas. The second book is dedicated to algorithmic techniques, and singles out bounded search trees and kernelization as the two major ones, while the generic heading 'Further Algorithmic Techniques' is devoted to integer linear programming, color coding, dynamic programming and tree decompositions.

In this paper, we survey what we consider to be the nine most important general techniques, namely: bounded search trees, greedy localization, color coding, local reductions, global reductions (crown reductions), iterative compression, extremal method, well-quasi-ordering (graph minors) and imposing fixed-parameter tractable (FPT) structure (treewidth). We classify these techniques under the four main themes of branching, kernelization, induction and win/win (see Fig. 1). In addition to its pedagogic value, we believe that such a taxonomy could help in developing both new techniques and new combinations of known techniques. The four main themes are described in separate sections. Each technique is introduced with a meta-algorithm, examples of its use are given, and we end with a brief discussion of its practicality. Regarding practicality, let us right away remark that in our opinion the most important decision is the choice of parameter. However, this issue is orthogonal to the algorithmic technique used and will thus not be discussed here.

Some aspects of this classification are novel, for example, the grouping of greedy localization and color coding together with bounded search trees under the heading of Branching algorithms, and the placement of the extremal method as the maximization counterpart of iterative compression under the heading of induction. We give a generic and formal description of greedy localization that encompasses problems hitherto not known to be FPT solvable by this technique. Sometimes a technique is known under different names[2] or it is a variant of a more general technique.[3] Clearly, the fastest FPT algorithm for a problem will usually combine several techniques. In some cases such a combination has itself been given a separate name.[4] We do not consider all these variations of the most

[1]This includes bounded search trees [1], data reduction [2], kernelization [1], The extremal method [3], The algorithmic method [4], catalytic vertices [3], crown reductions [5], modeled crown reductions [6], either/or [7], reduction to independent set structure [8], greedy localization [6], win/win [9], iterative compression [6], Well-Quasi-Ordering [1], FPT through treewidth [1], Search Trees [2], bounded integer linear programming [2], color coding [10], method of testsets [1], interleaving [11].

[2]This is the case with data reduction = local reduction rules, either/or = win/win, search trees = bounded search trees and other names like Hashing [1] = color coding, which do not seem to be in use anymore.

[3]This is the case with catalytic vertices $\subseteq$ local reduction rules, the algorithmic method $\subseteq$ extremal method, modelled crown reductions $\subseteq$ crown reductions, FPT by treewidth $\subseteq$ imposing FPT structure, reduction to independent set structure $\subseteq$ imposing FPT structure, method of testsets $\subseteq$ imposing FPT structure.

[4]This is the case with interleaving, which combines bounded search trees and local reduction rules. The technique known as bounded integer linear

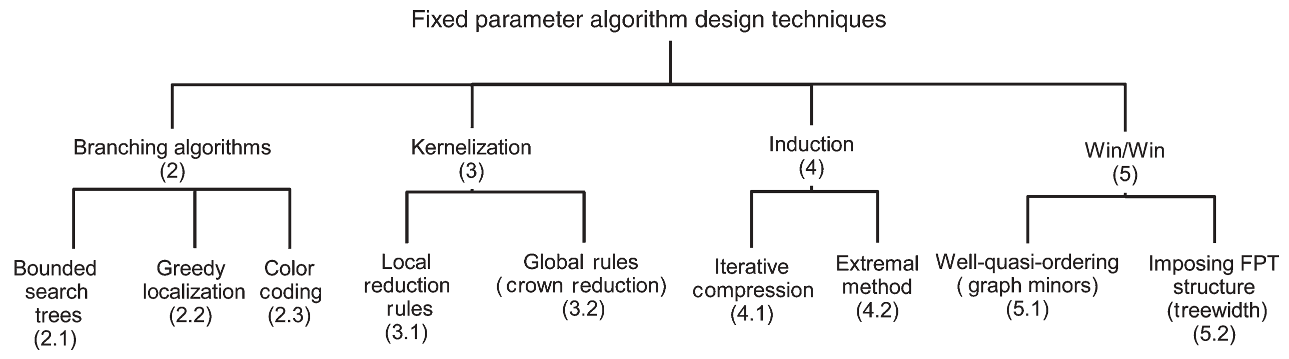
Fixed parameter algorithm design techniques
Branching algorithms (2)
Kernelization (3)
Induction (4)
Win/Win (5)
Bounded search trees (2.1)
Greedy localization (2.2)
Color coding (2.3)
Local reduction rules (3.1)
Global rules (crown reduction) (3.2)
Iterative compression (4.1)
Extremal method (4.2)
Well-quasi-ordering (graph minors) (5.1)
Imposing FPT structure (treewidth) (5.2)

FIGURE 1: The first two levels of the taxonomy, labelled by section number.

general techniques, and our discussion is restricted to the first two levels of the hierarchy in Fig. 1.

A note on notation: the chosen parameter for a problem is usually not stated explicitly, instead we use the convention that the variable name k always denotes the parameter. We use standard notation for graphs and focus on graph problems, particularly the following handful of quite famous ones: k-vertex cover (are there k vertices incident to all edges?), k-dominating set (are there k vertices adjacent to all vertices?), k-independent set (are there k pairwise non-adjacent vertices?), k-feedback vertex set (are there k vertices whose removal gives an acyclic graph?) and k-packing of H (are there k vertex disjoint copies of H?). All these problems are NP-hard, and our goal is an FPT algorithm, meaning that on input (G, k) for a graph G on n vertices it solves the problem in time $f(k)n^{\mathcal{O}(1)}$. Using the $\mathcal{O}^*$ notation that hides not only constants but also polynomial dependencies this runtime is simply expressed as $\mathcal{O}^*(f(k))$.

2. BRANCHING ALGORITHMS

We start by considering algorithm techniques that use a branching strategy to create a set of subproblems such that at least one of the subproblems is a yesinstance if and only if the original problem is a yes-instance. Techniques that create branching algorithms are bounded search trees, greedy localization and color coding. For a branching algorithm to have FPT running time it suffices to require that

- each node in the execution tree has polynomial running time;
- there are $\mathcal{O}((\log n)^{g(k)}f(k))$ branches at each step; and
- it reaches a polynomial time base case in at most $h(k)$ nested calls.

The running time of an algorithm that uses branching is bounded by the number of nodes of the execution tree times the running time for each node. Since we assume polynomial running time at each node in the execution tree, only the size of the tree contributes in $\mathcal{O}^*$-notation. It is normal that an algorithm checks for several different branching rules and branches according to one of the cases that applies. Although branching algorithms can have many branching rules it is usually straightforward to calculate the worst case. For the functions given above, the execution tree will have $\mathcal{O}(((\log n)^{g(k)}f(k))^{h(k)})$ nodes. To show that this is an FPT-function we prove the following.

OBSERVATION 1. *There exists a function $f(k)$ such that $(\log n)^k \leq f(k)\, n$ for all n and k.*

Proof. First recall that a logarithmic function grows slower than any rootfunction. That is, $\log n = o(n^{1/k})\ \forall k$, which implies that

$$\forall k \geq 1 \exists n_0 \geq 1 \text{ such that } \forall n \geq n_0 \ \log n < n^{1/k}.$$

Thus there must be a function $h(k)$: $\mathbb{N} \rightarrow \mathbb{N}^+$ such that $\forall k \forall n \geq h(k)$ we have $\log n < n^{1/k}$ and $\forall k\, \forall n \geq h(k)$ we have $(\log n)^k < n$.

So now the situation is that for all $n > h(k)$ we have that $(\log n)^k < n$ and the theorem holds. If on the other hand $n \leq h(k)$, then $\log n \leq \log h(k)$ and $(\log n)^k \leq (\log h(k))^k$. We thus have $(\log n)^k \leq \max\{1, (\log h(k))^k\} \cdot n$. □

It is easy to extend this to show that $\mathcal{O}((\log n)^{g(k)})$ is also an *FPT*-function.

2.1. Bounded search trees

The method of bounded search trees is arguably the most common and successful parameterized algorithm design technique. We can define it as a branching algorithm which is strictly recursive. A branching rule B_i identifies a certain structure in the graph (that we may call the left-hand side LHS_i) and creates $c \in \mathcal{O}((\log n)^{g(k)}f(k))$ new instances (G_1, k_1), $(G_2, k_2), \ldots, (G_c, k_{c_i})$ such that the instance (G, k) is a 'Yes'-instance if and only if at least one instance (G_j, k_j), $1 \leq j \leq k_{c_i}$ is a 'Yes'-instance. See the meta-algorithm given in Fig. 2.

programming can in a similar way be viewed as a combination of a branching algorithm with local reduction rules.

```
Meta-algorithm BST for rules of type $B_i : LHS_i \rightarrow (G_1, k_1), \ldots, (G_{c_i}, k_{c_i})$
Input Graph $G$, integer $k$

if $G$ contains some $LHS_i$ then Call BST on instances $(G_1, k_1), \ldots, (G_{c_i}, k_{c_i})$
else Solve $(G, k)$ in polynomial time, if 'Yes' then output 'Yes' and halt

Answer 'No'
```

FIGURE 2: A meta-algorithm for bounded search tree.

2.1.1. *Example:* k-*independent set and* k-*vertex cover*

We present two simple examples of bounded search tree algorithms.: an $\mathcal{O}^*(6^k)$ algorithm for k-independent set in a planar graph and an $\mathcal{O}^*(1.466^k)$ algorithm for k-vertex cover.

Observation 2. *k-independent set can for a planar graph G be solved by a BST algorithm in time $\mathcal{O}^*(6^k)$.*

Proof. In an instance (G, k) of k-independent set we know that for any maximal independent set $S \subseteq V(G)$ and for any $v \in V(G)$ it is true that $N[v] \cap S = \emptyset$. If this was not the case, we could include v to obtain a larger set, contradicting maximality. This together with the well-known fact that any planar graph contains a vertex v of degree at most five, allows us to continually branch on such a low-degree vertex v selecting either v or one of its neighbors to be in the indpendent set. When selecting a vertex for the independent set we remove it and all its neighbors from the graph and lower the parameter by one. This leaves us with at most six smaller instances, each with parameter $k' = k - 1$. As the problem is trivial when the parameter is zero, the size of the tree $T(k)$ is bounded by $6 \cdot T(k-1)$. This recurrence relation resolves to $T(k) \leq 6k$ (Fig. 3). □

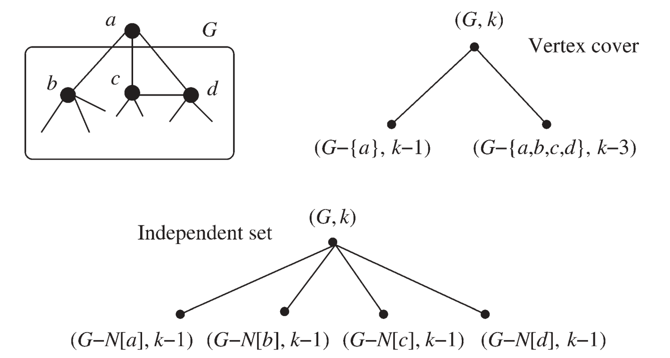

FIGURE 3: Illustrating observations 2 and 3
A graph G with vertex a having 3 neighbors, and two levels of the bounded search trees for k-independent set in planar graphs and for k-vertex cover. Any maximal independent set will contain either vertex a or at least one of its neighbors, while any vertex cover will contain either vertex a or all three neighbors.

Observation 3. *k-vertex cover can be solved by a BST algorithm in time $\mathcal{O}^*(1.466^k)$.*

Proof. To prove this we make two initial observations. First, that any valid vertex cover must contain either a vertex v or all its neighbors $N(v)$. Second, if a graph has only vertices of degree two or less, then k-vertex cover is linear time solvable. The second observation gives us a clear goal: we will try to reach a linear time instance by branching on any vertex of degree higher than two. We can do this as each high-degree vertex v creates only two branches, where one branch decreases the parameter by one and the other decreases the parameter by $|N(v)|$ (three or more). The recursive function $T(k) \leq T(k-1) + T(k-3)$ gives a bound on the size of the tree. This recursive function can be solved by finding the largest root of its characteristic polynomial $\lambda^k = \lambda^{k-1} + \lambda^{k-3}$. Using standard computer tools, this root can be estimated to 1.466, giving the desired result. □

2.1.2. *How is it used in practice?*

This is arguably the most successful parameterized algorithm design technique. The powerful k-vertex cover algorithm by Chen, *et al.* [12], running in time $\mathcal{O}(1.286^k n)$, is a bounded search tree algorithm and remains one of the most cited results in the field.

Also for practical implementations bounded search tree algorithms remain one of the best design techniques, achieving very good running times (M. Langston, Personal Communication). However, we would like to note that the effectiveness of a practical implementation of a bounded search tree algorithm may be dependent on the number of cases then the algorithm attempts. If the algorithm checks for just a handful of cases, then the running time of the algorithm is usually low, and its implementation feasible. However, there are cases in the literature where the number of cases reach the tens of thousands [13]. Although this allows for very good theoretical performance the unavoidable overhead will slow down any practical implementation. Some very good implementations of the vertex cover algorithm deliberately skip some of the more advanced cases as they in most practical instances only slow the algorithm down, as reported in (M. Langston, Personal Communication).

2.2. Greedy localization

Greedy localization is a technique that uses a clever first branching to start off the recursive search for the solution. It was introduced in [14, 15] and popularized in an IWPEC'04 paper [6]. Our aim is to show that if a parameterized problem satisfies the following conditions then greedy localization will give an FPT algorithm.

(1) The problem can be formulated as that of finding k pair-wise non- overlapping 'objects' in an input instance G, with objects being special subsets of size depending only on k of a ground set W of G.
(2) For any $R \subseteq W$ and $X \subseteq W$ we can decide in FPT time if there exists $S \subseteq W \setminus X$ such that $S \cup R$ is an object.

Not all bounded-size subsets of W are objects, and an obvious requirement for the problem to have an FPT algorithm is that for any $R \subseteq W$ we must be able to decide in FPT time if R is an object or not. Condition 2 can be seen as a strengthening of this obvious requirement and we will refer to S as an 'extension' of the 'partial object' R to a 'full object' $R \cup S$ avoiding X.

Figure 4 gives the greedy localization algorithm, in nondeterministic style, for a problem satisfying these two conditions. It uses the notation that for a set of partial objects $B = \{B_1, B_2, \ldots, B_k\}$ the ground elements contained in B are denoted by $W_B = \cap_{B_i \in B} B_i$.

Theorem 1. *If a parameterized problem satisfies conditions 1 and 2 above then the algorithm* 'greedy localization' *is an FPT algorithm for this problem.*

Proof. The algorithm starts by computing an inclusion maximal non-overlapping set of objects A. By condition 2 this first step can be done in FPT time as follows: repeatedly extend the emptyset to a full object while avoiding a set X, by calling subroutine EXTEND($\varnothing, X$) with X initially empty, and adding the extension to X before the next iteration. When no extension exists we are assured that the sequence of extensions found must be an inclusion maximal non-overlapping set of objects A.

```
Meta-algorithm GREEDY LOCALIZATION /* non-deterministic */
Input: Instance $G$ with ground set $W$, and an integer $k$
Output: Yes if $G$ has $k$ non-overlapping objects, otherwise No
compute an inclusion maximal non-overlapping set of objects $A$
if $A$ contains at least $k$ objects then halt and answer 'Yes'
else if $|W_A| < k$ then halt and answer 'No'
else guess $\{v_1, v_2, \ldots, v_k\} \subseteq W_A$ and let $B_i$ be partial object on $v_i$ $(1 \le i \le k)$
BRANCHING($B = \{B_1, B_2, \ldots, B_k\}$)

Subroutine BRANCHING
Input: Set of partial objects $B = \{B_1, B_2, \ldots, B_k\}$
Output: Yes if $B$ can be extended to a set of full objects, otherwise No
$F = S = \emptyset$
for $j = 1$ to $k$
    if $B_j$ not full then {
        $S =$EXTEND($B_j, W_B \cup F$)
        if $S == \emptyset$ then break
        else $F = F \cup S$
        }
if all objects could be extended then halt and answer 'Yes'
else if $F == \emptyset$ then halt and answer 'No'
else guess $v \in F$ and add $v$ to $B_j$ /* $j$ is value of parameter at break */
BRANCHING($B = \{B_1, B_2, \ldots, B_k\}$)

Subroutine EXTEND
Input: Partial object $B_i$, unavailable ground elements $X$
Output: Ground elements $S \subseteq W \setminus X$ whose addition to $B_i$ gives a full object
        or $S = \emptyset$ if this is not possible
```

FIGURE 4: A meta-algorithm for greedy localization, in non-deterministic style.

The crucial aspect that makes the algorithm correct is that if A and B are two inclusion maximal non-overlapping sets of objects then for any object B_i in B there is an object A_j in A such that A_j and B_i overlap, since otherwise A is not maximal. Thus, if the instance contains such a set B of at least k objects, then we can guess k ground elements appearing in A, with A constructed in the first step of the algorithm, such that these ground elements belong to k separate objects of B. The branching subroutine is called on these k one-element partial objects and we greedily try to extend them to full objects. If this fails for some object B_j, after having added extension elements F to objects $B_1, B_2, \ldots, B_{j-1}$, then there must exist an element v from F that should have instead been used to extend B_j. We then simply guess the element v and try again.

For a deterministic algorithm, the guesses are replaced by branchings, and we give a No answer iff all branches answer No. The first branching is of size $\binom{|W_A|}{k}$, the remainder of the branches are of size $|F|$, and the total height of the tree is bounded by k times the maximum size of an object since at each level one new ground element is added to W_B. All these are bounded by a function depending on k as we assumed in condition 1 that each object had size depending on k only. The calls to the extend subroutine are FPT by condition 2. Hence the algorithm is FPT. □

2.2.1. *Example:* k*-packing of* k*-cycles*

Using Theorem 1, we can easily argue that deciding if a graph G contains k vertex-disjoint cycles on k vertices is FPT by greedy localization. The ground set W will be the vertex set of G and the objects will be subsets of k vertices inducing a subgraph containing a k-cycle, to satisfy condition 1. Given R, $X \subseteq W$ we can by an FPT algorithm designed using the Color Coding technique, see the next section, decide if there exists $S \subseteq W \setminus X$ such that $R \cup S$ induces a subgraph containing a k-cycle, to satisfy condition 2. By Theorem 1 the greedy localization meta-algorithm therefore solves the problem in FPT time. For packing of edge-disjoint cycles, a similar argument holds with W being the edge set of the graph.

2.2.2. *How is it used in practice?*

Since we do not know of any practical implementation of this technique we simply give our opinion on the practical usefulness of this design technique. Observe that the running time of this algorithm depends on the running time of the EXTEND subroutine and the maximum size of the objects. The size of objects determine the size of our branching tree and also the number of times the EXTEND subroutine is executed. This, together with the inherent running time of $\mathcal{O}^*(c^{k \log k})$, lead us to believe that greedy localization is impractical for anything but small constant sized objects.

2.3. Color coding

Color coding is a technique that was introduced by Alon *et al.* in their paper 'color coding' [10] and is characterized by a powerful first branching step. For example, given an input to a parameterized graph problem we may in a first branching color the vertices with k colors such that the structure we are looking for will interact with the color classes in a specific way. To do this we create many branches of colored graphs using a family of perfect hash functions for the coloring.

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

Schmidt and Siegal [16] describe a construction of a k-perfect family of hash functions of size $2^{\mathcal{O}(k)} \log^2 n$, and [10] describes how to obtain an even smaller one of size $2^{\mathcal{O}(k)} \log n$.

The technique could for example apply a family of perfect hash functions to partition vertices of the input graph into k color classes. By the property of perfect hash families we know that for any k-sized subset S of the vertices, one of the hash functions in the family will color each vertex in S with a different color. Thus, if we seek a k-set C with a specific property (e.g. containing a k-cycle), we know that if there is such a set C in the graph then its vertices will, for at least one function in the hash family, be colored with each of the k colors. See the meta-algorithm in Fig. 5. The color coding technique gives an FPT algorithm whenever this colored subproblem can be solved in FPT time.

2.3.1. *Example:* k*-cycle*

To give a simple example of how to use color coding we give an algorithm for the k-cycle problem, which asks if an input graph has a cycle of length exactly k. This problem is obviously NP-complete since it is equivalent to Hamiltonian cycle for $k = n$. Let us consider the 'k-cycle algorithm' in Fig. 6.

Observation 4. *The 'k-cycle algorithm' is correct and runs in time $\mathcal{O}^*(2^{\mathcal{O}(k)} k!)$.*

Proof. Given an instance (G, k) we prove that the algorithm outputs a k-cycle if and only if G contains a k-cycle.

In one direction the algorithm answers 'Yes' and outputs a cycle. As the edges not deleted from the graph go from color-class c_i to $c_i + 1 (\text{mod } k)$, the shortest cycle in the graph is of length k and since the breadth first search only test for lengths up to k, we will not find cycles longer than k. Thus, if the algorithm outputs a cycle, it must be of length k.

For the other direction, assume in contradiction that G has a k-cycle $S = \langle s_1, s_2, \ldots, s_k, s_1 \rangle$, while the algorithm produces a 'No' answer. Since $\mathcal{F}$ is a perfect hash family, there exists a function $f \in \mathcal{F}$ such that the vertices $\{s_1, s_2, \ldots, s_k\}$ all

```
Meta-algorithm Color Coding
Input: A graph $G = (V, E)$ and an integer $k$.
Output: A 'Yes' or a 'No' answer

Let $\mathcal{F}$ be a family of perfect hash functions from $\{1, \ldots, |V|\}$ to $\{1, \ldots, k\}$

For each function $f \in \mathcal{F}$
    Use $f$ to color $V$ using $k$ colors.
    run algorithm for the colored problem
    if 'Yes' then Answer 'Yes' and halt
End For
Output a 'No' Answer.
```

FIGURE 5: A meta-algorithm for 'color coding'.

```
Algorithm $k$-cycle
Input: A graph $G = (V, E)$ and an integer $k$.
Output: A subgraph of $G$, isomorphic to a cycle of size $k$, or a 'No' answer

Let $\mathcal{F}$ be a family of perfect hash functions from $\{1, \ldots, |V|\}$ to $\{1, \ldots, k\}$

For each function $f \in \mathcal{F}$
    Use $f$ to color $V$ using $k$ colors
    For each ordering $c_1, c_2, \ldots, c_k$ of the $k$ colors
        Construct a directed graph as follows:
        for each edge $(u, v) \in E$
            if $f(u) = c_i$ and $f(v) = c_{i+1 (\bmod k)}$ for some $i$
                then replace edge with arc $\langle u, v \rangle$
            else delete edge $uv$

        For all $v$ such that $f(v) = c_1$
            Use breadth first search to look for cycle $C$ from $v$ to $v$ of length $k$
            If such $C$ exists then output $C$ and halt
        End For
    End For
End For
Output 'No'
```

FIGURE 6: An algorithm for the k-cycle problem.

received different colors when f was used as the coloring function. Since the algorithm tries every possible ordering of the color classes, it will try $\langle f(s_1), f(s_2), \ldots, f(s_k) \rangle$. Under this ordering, none of the edges in the cycle S will be removed, and since we test every vertex of one color class $f(s_i)$, we will at some point test if there exists a cycle from s_i to itself and output a 'Yes'-answer, contradicting the assumption.

To calculate the running time, we know by Alon *et al.* [10] that we have a perfect hash family of size $2^{\mathcal{O}(k)} \log n$. Thus, the result follows as the number of orderings of the k color classes is $k!$, and the rest is a polynomial factor. □

Note that instead of investigating each possible ordering of the color classes in order to find a cycle we could use a dynamic programming strategy. This would improve the running time, but we have chosen this simpler version because we wish to emphasize the color coding part of the algorithm.

2.3.2. *How is it used in practice?*

Perhaps the strongest theoretical results using color coding are obtained in [17] where it is combined with kernelization to give FPT algorithms for a large variety of packing problems. In general, a major drawback of these algorithms is that while the hash family has an asymptotically good size, the $\mathcal{O}$-notation hides a large constant. Thus, from a practical viewpoint the color coding algorithms would for reasonable input sizes normally be slower than a $2^{k \log k}$ algorithm obtained through other techniques. We believe that this algorithm design technique is unsuitable in practice.

3. KERNELIZATION

Under the heading of kernelization, we combine techniques that reduce a general instance into an equivalent *kernel*, i.e. an instance whose total size is bounded by a function depending only on the parameter. We distinguish between *local reductions* and *global reductions*.

3.1. Local reductions

The main tool to facilitate the reduction is the *reduction rule*. Each reduction rule identifies a certain structure LHS (the left-hand side) in the instance and modifies it to RHS (the right-hand side), possibly by deletion (RHS $= \emptyset$). This must be done in such a way that the original instance (G, k) has a positive solution iff the reduced instance (G', k') has one. The identified structure and resulting change is usually of a fixed size, and we then call it a local reduction rule.

If a reduction rule A cannot be applied to an instance we say that the instance is *irreducible for A*. That an instance is irreducible implies that it does not have the structure the reduction rule applies to. In this way, we can use reduction rules to remove a certain structure from an instance. This allows us to shed away trivial and/or polynomial solvable parts of an instance, thus revealing the combinatorial hard core.

Normally, a single reduction rule is not sufficient to reduce a problem to a kernel. We usually require a set of reduction rules, where each rule in the set removes one type of problematic structure. If a set of reduction rules is sufficient to reduce any instance of a problem to a kernel we call it a *complete set of reduction rules*. Given such a set we can devise a simple algorithm as seen in Fig. 7.

3.1.1. *Example:* k*-vertex cover and* k*-dominating set*

Here we will give the classical example of a quadratic kernelization algorithm for vertex cover. It is by far the simplest algorithm using reduction rules known to us and illustrates the technique very well. We will make use of a reduction rule applicable to vertices of degree larger than k [1]. It is easily seen to be true and says that

Rule 1. *Assume $v \in V(G)$ with $deg(v) > k$. Then G has a vertex cover of k vertices if and only if $G - v$ has a vertex cover of $k - 1$ vertices.*

Observation 5. *The singleton set containing Reduction Rule 1 is a complete set of reduction rules for k-vertex cover and will in linear time give a kernel of $\mathcal{O}(k^2)$ vertices.*

Proof. Let us examine the reduced graph G' that remains after we have deleted all vertices of degree more than k. This instance G' (with the parameter k') has been obtained from the input (G, k) by repeatedly applying Reduction Rule 1. Thus by correctness of Reduction Rule 1, we know that G' has a vertex cover of size k' if and only if G has a vertex cover of size k. Since the reduced graph G' has vertices of degree at most k, any vertex in $V(G')$ can cover at most k edges. Thus the total number of edges a vertex cover of size k' can cover is at most $k' \cdot k$, thus it is at this point safe to reject any graph G' with more edges than $k' \cdot k$. It is easy to see that the algorithm works in linear time. It simply scans through the vertices and deletes the vertices of degree more than k. □

For some problems obtaining a kernel is trivial. In the following example, we consider k-dominating sets in cubic graphs, i.e. where all vertices have degree three. Thus no vertex can dominate more than four vertices (itself and its three neighbors) and we can safely answer No whenever the input graph has more than $4k$ vertices. Note that we did not apply any reduction rule at all to find this kernel of $4k$ vertices and $12k$ edges. For many problems this would be a very good result, but here it is terrible. By the same argument we see that no cubic graph has a dominating set of size less than $n/4$. Thus, for any non-trivial problem instance we have $k \geq n/4$ and thus $4k \geq n$, and the bound $4k$ obtained from the kernel is larger than the size of the instance itself. This shows that it is

```
Meta-algorithm Complete set $R$ of reduction rules $R_i : LHS_i \rightarrow RHS_i$
Input Graph $G$, integer $k$

while $G$ contains some $LHS_i$
    remove $LHS_i$ from $G$ and replace by $RHS_i$
end while
run a brute force algorithm on the reduced graph
```

FIGURE 7: A meta-algorithm for a complete set of reduction rules.

important to be aware of the lower and upper bounds on interesting instances of the problem one is working on. This can be of great help in finding trivial kernels and estimating the quality of a suggested kernel.

3.1.2. How is it used in practice?

The idea of pre-processing an input forms a heuristic for many real-world problems. Kernelization is a sibling of this idea where we have a guarantee on its effectiveness. It can can be extremely useful in practical implementations as it allows for a fast reduction of the input size. For many instances, the reduction rules work much better than the worst case analysis would indicate [18].

3.2. Global reduction rules—crown reduction

Lately there has been a focus on reduction rules that do not follow the pattern of finding a local structure of constant size. In this section we describe reduction rules based on finding *crown decompositions* in graphs. (see Fig. 8).

Definition 2. *A crown decomposition of a graph $G = (V, E)$ is a partitioning of V into sets C,H,R, where C and H are both non-empty, such that:*

(1) *C is an independent set.*
(2) *There is no edge between a vertex in C and a vertex in R.*
(3) *There exists an injective map m: $H \rightarrow C$, such that $m(a) = b$ implies that ab is an edge. We call ab a matched edge if $m(a) = b$.*

When using a crown decomposition (C, H, R) in a reduction rule for a graph G we usually show that we can remove or modify $(C \cup H)$ to obtain a reduced instance (G', k') which is a Yes-instance if and only if (G, k) is a Yes-instance. For example, it is easy to see that G has a vertex cover of size k iff the graph G' resulting from removing $C \cup H$ has a vertex cover of size $k - |H|$. Usually, more complicated reduced instances and arguments are necessary. For example, an FPT algorithm for *k*-internal spanning tree [8] uses crown reduction rules that remove only the vertices of C not incident to a matched edge.

Although it is possible to determine if a graph has a crown decomposition in polynomial time [19], this technique is often combined with the following lemma by Chor *et al.* [5].

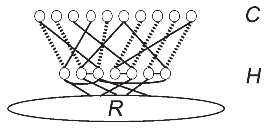

FIGURE 8: Example of a crown decomposition. The matched edges are dashed.

Lemma 1. *If a graph $G = (V, E)$ has an independent set I such that $|N(I)| < |I|$, then a crown decomposition (C,H,R) for G such that $C \subseteq I$ can be found in time $\mathcal{O}(|V| + |E|)$.*

The notation $N(I)$ denotes vertices in $V \setminus I$ that are adjacent to a vertex in I. Since it is $W[1]$-hard to find a large independent set we cannot directly apply Lemma 1. To see how the lemma can be used, we consider *k*-vertex cover on a graph G in the next section.

Although crown reduction rules were independently discovered by Chor *et al.* [5] one should note that a similar type of structure has been studied in the field of boolean satisfiability problems (SAT). An *autarky* is a partial truth assignment (assigning true/false to only a subset of the variables) such that each clause that contains a variable determined by the partial truth assignment is satisfied. In a *matching autarky*, we require in addition that the clauses satisfied and the satisfying variables form a matching cover in the natural bipartite graph description of the SAT. It is easy to see that the matching autarky is a crown decomposition in this bipartite graph. The main protagonist in this field is Kullmann [20, 21], who has developed an extensive theory on different types of autarkies.

3.2.1. Example: k-*vertex cover*

We have already remarked the following quite simple crown reduction rule.

Rule 2. *Given a crown decomposition (C,H,R) of a graph $G = (V,E)$, then G has a vertex cover of size k if and only if $G' = G[V - (C \cup H)]$ has a vertex cover of size $k' = k - |H|$.*

In Fig. 9, this rule is applied to *k*-vertex cover with a simple win/win argument.

Observation 6. *The algorithm crown kernelization in Fig. 9 either terminates with a correct answer or produces a kernel of at most 4k vertices for k-vertex cover.*

Proof. To see that the algorithm always terminates, observe that the graph either gives an output or reduces the graph. Since we can have at most $\mathcal{O}(n)$ reductions, the algorithm will eventually terminate.

We first show that a 'No' answer is always correct. The algorithm will only output 'No' if there is a maximal matching M in G where $|V(M)| > 2k$. Since we have to pick at least one vertex from each edge, the vertex cover for this graph is greater than k.

The algorithm modifies the instance (G, k), so we have to make sure we do not introduce or remove any solutions. At any stage of the algorithm the current graph and parameter has been obtained by the repeated applications of the crown reduction rule. By Rule 2, we are thus guaranteed that the reduced instance is a 'Yes'-instance if and only if the input instance is a 'Yes'-instance.

```
Algorithm Crown Kernelization for Vertex Cover
Input: A graph $G = (V, E)$ and an integer $k$.
Output: A kernel of size at most $4k$ or a 'No' answer

do Create a maximal matching $M$ in $G$
    if $|V(M)| > 2k$ then output answer 'No' and halt
    else
        if $|V(G) - V(M)| \leq 2k$ then output $(G, k)$ and halt
        else
            Create crown decomposition $(C, H, R)$
            Let $G = G[V - (C \cup H)]$, and $k = k - |H|$
repeat
```

FIGURE 9: A $4k$ kernelization algorithm for vertex cover.

We have $|V(G)| = |V(M)| + |V(G) - V(M)| \leq 2k + 2k = 4k$, so the algorithm outputs only graphs of at most $4k$ vertices. Thus, the observation is correct. □

3.2.2. How is it used in practice?

Crown decompositions have been used with great success in the vertex cover problem. Although it does not always reach as good kernel as the competitive linear programming or network flow algorithms it uses only a fraction of the time and it is often worth using a crown reduction rule as a preprocessing for the more time consuming methods, as reported by implementations in [19].

4. FPT BY INDUCTION

We discuss techniques closely related to mathematical induction. If we are provided a solution for a smaller instance (G, k) then we can for some problems use the information to determine the solution for one of the larger instances $(G + v, k)$ or $(G, k + 1)$. We will argue that the two techniques iterative compression and extremal method are actually two facets of this inductive technique, depending on wether the problem is a minimization problem or a maximization problem. In his book *Introduction to Algorithms* [22], Manber shows how induction can be used as a design technique to create remarkably simple algorithms for a range of problems. He suggests that one should always try to construct a solution based on the inductive assumption that we have a solution to smaller problems. For example, this leads to the well-known insertion sort algorithm by noting that we can sort sequences of n elements by first sorting $n - 1$ elements and then inserting the last element at its correct place.

This inductive technique may also be applied to the design of FPT algorithms but more care must be taken on two accounts: (i) we have one or more parameters and (ii) we are dealing with decision problems. The core idea of the technique

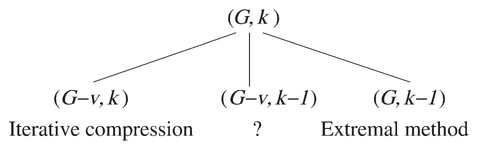

FIGURE 10: Three ways, of which only two correspond to known techniques, to design an FPT algorithm by induction.

is based on using the information provided by a solution for a smaller instance. When an instance contains both a main input and a parameter input, we must be clear about what we mean by 'smaller' instances. Let (G, k) be an instance, where G is the main input and k the parameter. We can now construct three distinctly different 'smaller' instances $(G - v, k)$, $(G, k - 1)$ and $(G - v, k - 1)$. Which one of these to use?

We first show that using smaller instances of the type $(G - v, k)$ is very suitable for minimization problems and leads to a technique known as iterative compression. Then we show that using smaller instances of the type $(G, k - 1)$ can be used to construct algorithms for maximization problems and is in fact the technique known as the extremal method (Fig. 10).

4.1. For minimization—iterative compression

In this section, we present iterative compression which works well on certain parameterized minimization problems. Let us assume that we can inductively (recursively) compute the solution for the smaller instance $(G - v, k)$. Since our problems are decision problems, we get either a 'Yes'-answer or a 'No'-answer. In both cases, we must use the information provided by the answer to compute the solution for (G, k). We must assume that for a 'Yes'-instance we also have a certificate that verifies that the instance is a 'Yes'-instance and it is this certificate that must be used to compute the solution for (G, k). However, for a 'No'-answer we may receive no extra information. A class of problems where 'No'-answers carry sufficient

```
Meta-algorithm Iterative Compression
Input: Graph $G$, integer $k$

if $|V(G)| < k$ then run brute force algorithm, return answer and halt
Choose arbitrary vertex $v$
Recursively call problem on instance $(G - v, k)$, obtain solution $S$
if $S =$ 'No' then answer 'No'
else  Compute in FPT time a solution for $(G, k)$ based on $S$
```

FIGURE 11: A meta-algorithm for iterative compression on a monotone graph minimization problem.

information is the class of *monotone* problems in which the 'No'-instances are closed under element addition. Thus, if a problem is monotone we can immediately answer 'No' for (G, k) whenever $(G - v, k)$ is a 'No'-instance. Given a monotone problem we can use a meta-algorithm as seen in Fig. 11.

The algorithm will recursively call itself at most $|V(G)|$ times, thus the running time of an algorithm of this type is $\mathcal{O}(n)$ times the time it takes to compute a solution given a smaller solution.

Note that many minimization problems are not monotone, like k-dominating set where the addition of a universal vertex always changes a 'No' answer to 'Yes' (unless $k \geq n$). For such problems, we believe that iterative compression is ill suited.

4.1.1. *Example: k-feedback vertex set*

We will give an algorithm for the problem k-feedback vertex set using the technique described in the previous section. Feedback vertex set (is there a k-set S such that $G[V - S]$ is acyclic?) is a monotone problem as adding new vertices will never help to cover existing cycles. Consider the 'Feedback Vertex Set Algorithm' in Fig. 12.

Observation 7. *Algorithm 'FVS' is correct, and solves k-feedback vertex set in FPT time.*

Proof. We will first prove that the algorithm does not incorrectly decide the answer before running the treewidth subroutine. If the algorithm answers 'Yes' because $|V(G)| \leq k$, it is correct as we can select V as our feedback vertex set. If the algorithm answers 'No' because $(G - v, k)$ is a 'No'-instance, it is correct as k-feedback vertex set is a monotone problem.

We assume the treewidth subroutine is correct so it remains to show that the algorithm computes a tree decomposition of the graph with bounded treewidth. The algorithm computes the graph $T = G - (S \cup v)$ which is a forest, and it is easy to construct a tree decomposition of width one from this forest, having one bag for each edge. The algorithm then adds $S \cup v$ to each bag to obtain a correct tree decomposition of width $k + 2$.

It follows from results in [23] that k-feedback vertex set is solvable in FPT time if the treewidth is the parameter. This gives the desired result. □

Two other papers that use this type of induction are [6, 24]. In [24], Reed *et al.* managed to show that the problem k-odd cycle cover (is there a k-set S such that $G[V - S]$ is bipartite?) is in

```
Algorithm FVS
Input: Graph $G$, integer $k$

if $|V(G)| \leq k$ then return 'Yes' and halt
Choose arbitrary vertex $v \in G$
$S = FVS(G - v, k)$
if $S =$ 'No' then answer 'No' and halt

Let $T = G - (S \cup v)$, note that $T$ is a forest.
Create tree decomposition $(X, I)$ of $T$.
Add $S \cup v$ to each bag in $X$ to obtain tree dec. $(X', I)$ of width at most $k + 2$.
run Treewidth algorithm for feedback vertex set on tree decomposition $(X', I)$
output answer from treewidth algorithm.
```

FIGURE 12: An inductive algorithm for k-feedback vertex set.

```
Meta-algorithm Extremal Method
Input: Graph $G$, integer $k$

if $k = 0$ then run brute force algorithm, return answer and halt
Inductively call problem on instance $(G, k-1)$, obtain solution $S$
if $S$ = 'No' then answer 'No'
else Compute in FPT time a solution $S'$ for $(G, k)$ based on $S$ and return $S'$
```

FIGURE 13: A meta-algorithm for parameter monotone problems.

FPT, thus settling a long open problem in the field. By the induction hypothesis, assume that we can determine if $(G - v, k)$ is a 'No' or a 'Yes' instance for k-odd cycle cover. The induction step is then in two parts. First, the trivial part: to show that if $(G - v, k)$ is a 'No'-instance then the inclusion of another vertex cannot improve the situation. The second part deals with the situation when we have a positive certificate S consisting of the k vertices to be deleted from $(G - v, k)$ to make it bipartite. If we have such a certificate S, we can conclude that $S \cup \{v\}$ is a solution of size at most $k + 1$ for G. The authors then show, by a fairly complicated argument which we will not explain here, that given a solution of size at most $k + 1$ it is possible to determine in FPT time whether or not there exists a solution of size k.

A very similar use of induction is seen in [6] where Dehne *et al.* give a $2k$ kernel for k-vertex cover without using the complicated Nemhauser–Trotter results [25]. By induction assume we can determine if $(G - v)$ has a k-vertex cover, if no such cover exists we cannot find one for G either. On the other hand if $(G - v)$ has a k-vertex cover S then $S \cup \{v\}$ is a $k + 1$-vertex cover for G and by implication has a $n - (k + 1)$-independent set. As long as $|V(G)| > 2k + 2$ we know by Lemma 1 that G has a crown decomposition, which in turn leads to a reduction in G as seen in Lemma 2. This reduction continues until the graph has size at most $2k + 2$, at which point we can test for k-vertex cover using a brute force algorithm.

4.1.2. How is it used in practice?

The general practical effectiveness of this design technique depends directly on how well the induction step can be implemented. For the vertex cover problem above we use an effective crown reduction algorithm and thus the accompanying algorithm would run well in practice. We hope that the implementations testing this technique in practice will appear soon.

4.2. For maximization—the extremal method

For maximization problems we consider smaller instances of the type $(G, k - 1)$, and induct on k instead of n. We say that a problem is *parameter monotone* if the 'No'-instances are closed under parameter increment, i.e. if instance (G, k) is a 'No'-instance then (G, k') is also a 'No'-instance for all $k' > k$. For a parameter monotone problem, we can modify the meta-algorithm from the previous section to obtain an inductive algorithm for maximization problems. (see Fig. 13).

The *Method of Extremal Structure*[5] is a design technique that works well for parameter monotone maximization problems. In this technique, we do not focus on any particular instance (G, k), but instead investigate the structure of graphs that are 'Yes'-instances for k, but 'No'-instances for $k + 1$. Let $\mathcal{G}(k)$ be the class of such graphs, i.e. $\mathcal{G}(k) = \{G \mid (G, k)$ is a 'Yes'-instance, and $(G, k + 1)$ is a 'No'-instance$\}$.

Our ultimate goal is to prove that there exists a function $f(k)$ such that $\max\{|V(G)| \; |G \in G(k)\} \leq f(k)$. This is normally not possible without some refinement of $G(k)$, to do this we make a set of observations E of the following type:

> Since (G, k) is a 'Yes'-instance, but $(G, k + 1)$ is a 'No'-instance, G has property p. (1)

Given a set of such observations E and consequently a set of properties P we try to devise a set of reduction rules R that apply specifically to large graphs having the properties P. We call our refined class $\mathcal{G}_R(k) = \{G|$ no reduction rule in R applies to (G, k), and (G, k) is a 'Yes'-instance, and $(G, k + 1)$ is a 'No'-instance$\}$. If we can add enough observations to E and reductions rules to R to prove that there is a function $f(k)$ such that max $\{|V(G)| \; |G \in \mathcal{G}_R(k)\} \leq f(k)$ we have proven that:

> If (i) no rule in R applies to (G, k) and (ii) (G, k) is a 'Yes'-instance and (iii) $(G, k + 1)$ is a 'No'-instance, then $|V(G)| \leq f(k)$.

Given such a boundary lemma and the fact that the problem is a *parameter monotone* maximization problem a *kernelization* lemma follows, saying that if no rule in R applies to (G, k) and $|V(G)| > f(k)$, then (G, k) is a 'Yes'-instance.

It is not immediately obvious that this can be viewed as an inductive process, but we will now make this clear by presenting the *Algorithmic Method*, a version of the 'Extremal

[5]An exposition of this design technique can be found in Prieto's PhD thesis [4].

Method'. Here the 'Extremal Method' can be used as the inductive step, going from k to $k+1$, in an inductive algorithm.

As its base case, the algorithm decides $(G, 0)$, which is usually a trivial 'Yes'-instance for a maximization problem. Our induction hypothesis is that we can decide (G, k'). Then as long as $k'+1 \leq k$ we try to compute $(G, k'+1)$. If (G, k') is a 'No'-instance we can immediately answer 'No' for $(G, k'+1)$ as the problem is parameter monotone. Otherwise we can now make an algorithmic use of observations of the type defined for extremal method ((1) above). For each of the properties $p \in P$ we check if G has the property p. If G does not have property p then since (G, k') is a 'Yes'-instance it follows that $(G, k'+1)$ is also a 'Yes'-instance. By the same reductions and observations (although the reader should observe that we here require properties to be FPT time verifiable), we obtain that

If no observation in E or reduction rule R applies to $(G, k'+1)$ then $|V(G)| < f(k)$.

At that point we can invoke a brute force algorithm to obtain either a solution S or a 'No'-answer for $(G, k'+1)$. This answer for $(G, k'+1)$ can then be used in the next step, $k'+2$, of our inductive algorithm.

4.2.1. *Example:* k*-packing of* $K_{1,3}$

Here we will give an inductive algorithm for deciding if a graph has a subgraph isomorphic to k vertex disjoint copies of $K_{1,3}$. First note the obvious reduction rule that removes edges whose endpoints both have degree at most two and that can therefore not participate in any $K_{1,3}$.

RULE 3. *If $\exists vu \in E(G)$ such that $deg(v) \leq deg(u) \leq 2$ then (G, k) is a 'Yes'-instance if and only if $(G' = (V(G), E(G) \setminus vu), k)$ is a 'Yes'-instance.*

Vertices of degree larger than $4(k-1)+3$ can on the other hand always be the center node in a $K_{1,3}$-star as there cannot be more than $4(k-1)$ remaining vertices used in the k-packing.

RULE 4. *If $\exists v \in V(G)$ such that $deg(v) > 4k-1$ then (G, k) is a 'Yes'-instance if and only if $(G' = G[V \setminus v], k-1)$ is a 'Yes'-instance.*

OBSERVATION 8. *k-packing of $K^{1,3}$ is FPT.*

Proof. We prove this by inducting on k. $(G, 0)$ is a trivial 'Yes'-instance. Let the induction hypothesis be that we can decide instance $(G, k'-1)$ in FPT time and in case of a 'Yes'-instance also give the $(k'-1)$ packing. We now prove the induction step from an arbitrary $k'-1$ to k'. By the induction hypothesis, we can decide $(G, k'-1)$. If $(G, k'-1)$ is a 'No'-instance then (G, k') is also a 'No'-instance as the problem is parameter monotone. Otherwise we have, by the induction hypothesis, a $(k-1)$ $-K_{1,3}$-packing. Let W be the vertices of this packing. Reduce the graph G until none of the two rules above apply.

First observe that the max degree in $G[V-W]$ is at most two, as otherwise W is not inclusion-maximal and (G, k') becomes a trivial 'Yes'-instance. This together with the two reduction rules is enough to prove that the reduced instance is a kernel.

From the second rule above it follows that $N(W) \leq 4k(4k-1)$. From the first rule we can conclude that $R = V - (W \cup N(W)$ is an independent set. Since R is an independent set, each vertex in R must have at least one adjacent vertex in $N(W)$. Thus, from the observation above we have that $|R| \leq 2|N(W)|$. In total: $|V| = |W| + |N(W)| + |R| \leq 4k + 3{\cdot}4k(4k-1) = 48k^2 - 8k$. We can now compute a solution on this kernel in a brute force manner to determine a solution for (G, k'). This completes the induction step. □

4.2.2. *How is it used in practice?*

This technique, either the 'Extremal Method' or its variant the 'Algorithmic Method', has been applied successfully to a range of problems, such as: k-max Cut [26], k-leaf spanning tree [4], k-non-blocker [4], k-edge-disjoint triangle-packing [27], k-$K_{1,s}$-packing [28], k-K_3-packing [29], k-set splitting [30] and k-internal spanning tree [8].

We do not know of any practical implementation of algorithms of this type, but the kernelization results that can be obtained from 'Extremal Method' argumentation is generally quite good. This, together with the general simplicity of the technique, indicates that the technique could be successful in practice.

5. WIN/WIN

Imagine that we solve our problem by first calling an FPT algorithm for another problem and use both its 'Yes' and 'No' answer to decide in FPT time the answer to our problem. Since we then 'win' if its answer is 'Yes' and we also 'win' if its answer is 'No', this is called a 'win/win' situation. In this section, we focus on techniques exploting this behavior.

Consider the meta-algorithm shown in Fig. 14. We run our decision algorithm Φ for problem B on input I. If Φ returns a 'No' answer, we know by the relationship between A and B that A is a 'Yes'-instance (this can sometimes yield a non-constructive result where we know that A is a yes-instance but we have no certificate). Otherwise we have obtained the knowledge that I is a 'Yes'-instance for B, and if Φ is constructive, we also have a certificate. We then proceed to solve A with the extra information provided. In the following subsection, we give examples of how this has been done in the literature.

```
**Meta-algorithm** Win/Win algorithm for problem $A$ using problem $B$
**Input:** Graph $G$, parameter $k$
**run** FPT algorithm $\Phi$ for $B$ on $(G,k)$
**if** $(G,k)$ is a 'No'-instance for $B$ **then** Output 'Yes'
**else run** use structure given by $\Phi$ to solve $A$ on $G$.
```

FIGURE 14: The classical win/win algorithm structure, although other yes/no relationships between *A* and *B* may be used.

```
**Meta-algorithm** Graph Minors
**Input:** A graph $G=(V,E)$ and an integer $k$.
**Output:** A 'Yes' or a 'No' answer

Let $MFM$ be the forbidden minors for the specified problem and given $k$.
**for** each minor $H \in MFM$
    **if** $H \preceq_m G$ **then answer** 'No' and **halt**
**end for**

**answer** 'Yes'
```

FIGURE 15: A meta-algorithm for the graph minors technique, applicable when the class $\{G \mid (G, k)$ is a 'Yes'-instance$\}$ is closed under minors.

5.1. Well-quasi-ordering and graph minors

Robertson and Seymour have shown that (i) the set of finite graphs are wellquasi-ordered under minors and (ii) the *H*-minor problem that checks if *H* is a minor of some input graph, with $k = |V(H)|$, is FPT. These two facts suffice to prove that any parameterized graph problem whose Yes-instances (or Noinstances) are closed under minors is FPT. If analogous structural results could be shown for some other relation, besides minors, then for problems closed under this other relation we would also get FPT algorithms. Thus, the general technique is called 'well-quasi-ordering'. We consider this a win/win algorithm as we relate the problem we wish to solve to the FPT problem of checking if one of the forbidden minors (or whatever other relation is involved) appear in our problem instance.

Let us briefly explain the main ideas. A well-quasi-ordering is a reflexive and transitive ordering which has no infinite antichain, meaning that any set of elements, which are comparable in the ordering must be finite. A graph *H* is a *minor* of a graph *G*, denoted $H \preceq_m G$, if a graph isomorphic to *H* can be obtained from contracting edges of a subgraph of *G*. The graph minors theorem [3] states that 'The set of graphs are well-quasi-ordered by the minor relation'. Combined with *H*-minor testing this can be used to prove existence of an FPT algorithm for any problem *A* with the property that for any *k* the 'Yes'-instances are closed under minors. In other words, let us assume that if A_k is the class of graphs *G* such that (*G*, *k*) is a 'Yes'-instance to problem *A* and $H \preceq_m G$ for some $G \in A_k$ then $H \in A_k$ as well. Consider the minimal forbidden minors of A_k, denoted MFM(A_k), defined as follows: MFM(A_k) $= \{G \mid G / \notin A_k$ and $(\forall H \preceq_m G, H \in A_k \vee H = G)$ (Fig. 15)$\}$.

By definition, MFM(A_k) is an antichain of the $\preceq m$ ordering of graphs so by the graph minors theorem it is finite. Beware that the non-constructive nature of the proof of the graph minors theorem implies that we can in general not construct the set MFM(A_k) and thus we can only argue for the existence of an FPT algorithm. We do this by noting that (*G*, *k*) is a Yes-instance of problem *A* iff there is no $H \in$ MFM(A_k) such that $H \preceq_m G$. Since MFM(A_k) is independent of $|G|$, though dependent on *k*, we can therefore decide if (*G*, *k*) is a Yes-instance in FPT time by |MFM(A_k)| calls of *H*-Minor.

5.1.1. *Example: k-feedback vertex set*

Armed with this powerful tool, all we have to do to prove that a parameterized graph problem is FPT, is to show that the Yes-instances are closed under the operations of edge deletion, vertex deletion and edge contraction. Consider the problem *k*-feedback vertex set, where we ask for *k* vertices whose removal creates an acyclic graph. Deleting an edge or vertex does not create any new cycles in the graph. After contracting the edge *uv* into a new vertex *x* the vertex *x* will be a part of any cycle that was previously covered by both *u* and *v*, and no new cycles apart from these are introduced by the contraction.

OBSERVATION 9. *If (G, k) is a 'Yes'-instance for k-feedback vertex set and $H \preceq_m G$, then (H, k) is a 'Yes'-instance for k-feedback vertex set.*

We now have enough information to conclude that k-Feedback Vertex Set is in FPT.

5.1.2. *How is it used in practice?*

Although its extreme simplicity makes the technique very useful as a method for proving membership in FPT, the number of forbidden minors is often quite large and even worse, usually unknown. Thus, this technique is rarely viable as a tool for designing practical algorithms.

5.2. Imposing FPT structure and bounded treewidth

In the literature on parameterized graph algorithms there are several notable occurrences of a win/win strategy that imposes a tree-like structure on the class of problematic graphs, in particular by showing that they must have treewidth bounded by a function of the parameter. This is then combined with the fact that many NP-hard problems are solvable in FPT time if the parameter is the treewidth of the input graph.

5.2.1. *Example: k-dominating set*

Let us briefly explain this technique in the case of finding k-dominating sets in planar graphs, where a very low-treewidth bound on Yes-instances gives very fast FPT algorithms. In [32], it is shown that a planar graph that has a k-dominating set has treewidth at most $c\sqrt{k}$ for an appropriate constant c. Thus, we have a win/win relationship, since we can check in polynomial time if a planar graph has treewidth at most $c'\sqrt{k}$ [33], for some slightly larger constant c', and if so find a tree-decomposition of this width. If the treewidth is higher we can safely reject the instance, and otherwise we can run a dynamic programming algorithm on its tree-decomposition, parameterized by $c'\sqrt{k}$, to find in FPT time the optimal solution. In total this gives a $\mathcal{O}^*(c''\sqrt{k})$ algorithm for deciding if a planar graph has a dominating set of size k.

A series of papers have lowered the constant c'' of this algorithm, by several techniques, like moving to branchwidth instead of treewidth, by improving the constant c and by improving the FPT runtime of the dynamic programming stage. Yet another series of papers have generalized these 'subexponential in k' FPT algorithms from dominating set to all so-called bidimensional parameters and also from planar graphs to all graphs not having a fixed graph H as minor [34].

5.2.2. *How is it used in practice?*

According to [34] the only known algorithms with subexponential running time $\mathcal{O}^*(c^{\sqrt{k}})$ are algorithms based on imposing treewidth and branchwidth structure on the complicated cases, and these fall into the win/win category. We cannot say much about the practical usefulness of this design technique in general. The running time of a win/win algorithm depends on two things: the nature of the observation that ties one problem to another, and the running time to solve the other problem. Normally the observation can be checked in polynomial time, but to have a practical implementation it must also give a reasonable parameter value for the other problem.

6. REFERENCES

[1] Downey, R. and Fellows, M. (1999) *Parameterized complexity*, Springer-Verlag.

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

[3] Fellows, M.R., McCartin, C., Rosamond, F. and Stege, U. (2000) Coordinatized kernels and catalytic reductions: an improved FPT algorithm for max leaf spanning tree and other problems. *Foundations of Software Technology and Theoretical Computer Science.*

[4] Prieto, E. Systematic kernelization in FPT algorithm design. PhD Thesis, University of Newcastle, Australia.

[5] Chor, B., Fellows, M. and Juedes, D. (2004) Linear kernels in linear time, or how to save k colors in $\mathcal{O}$ (n^2) steps. *Proc. WG2004*, Lecture Notes in Computer Science.

[6] Dehne, F., Fellows, M., Rosamond, F. and Shaw, P. (2004) Greedy localization, iterative compression and modeled crown reductions: new FPT techniques and improved algorithms for max set splitting and vertex cover. *Proc. IWPEC04*, Lecture Notes in Computer Science, vol. 3162, 271–281.

[7] Prieto, E. and Sloper, C. (2003) Either/or: using vertex cover structure in designing FPT-algorithms—the case of k-internal spanning tree. *Proc. WADS 2003*, Lecture Notes in Computer Science, vol. 2748, pp. 465–483.

[8] Prieto, E. and Sloper, C. (2005) Reducing to independent set structure—the case of k-internal spanning tree', *Nordic J. Comput.*, vol 12, 308–318.

[9] Fellows, M. (2003) Blow-ups, win/wins and crown rules: some new directions in FPT. *Proc. WG 2003*, Lecture Notes in Computer Science, vol. 2880, pp. 1–12, Springer-Verlag. pp. 1–12.

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

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

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

[13] Robson. Finding a maximum independent set in time $\mathcal{O}(2^{n/4})$? Manuscript.

[14] Chen, J., Friesen, D., Jia, W. and Kanj, I. (2004) Using nondeterminism to design efficient deterministic algorithms. *Algorithmica*, **40**, 83–97.

[15] Jia, W., Zhang, C. and Chen, J. (2004) An efficient parameterized algorithm for m-set packing. *J. Algorithms*, **50**, 106–117.

[16] Schmidt, J.P. and Siegel, A. (1990) The spatial complexity of oblivious k-probe hash functions. *SIAM J. Comput.*, **19**, 775–786.

[17] Fellows, M., Knauer, C., Nishimura, N., Ragde, P., Rosamond, F., Stege, U., Thilikos, D. and Whitesides, S. (2004) Faster fixed-parameter tractable algorithms for matching and packing problems, *Proc. 12th Annual European Symp. Algorithms* (*ESA 2004*).

[18] Weihe, K. (2001) On the differences between practical and applied (invited paper). *Proc. WAE 2000*, Lecture Notes in Computer Science, vol. 1982, pp. 1–10, Springer-Verlag.

[19] Abu-Khzam, F., Collins, R., Fellows, M. and Langston, M. (2004) Kernelization algorithms for the vertex cover problem: theory and experiments. *Proc. ALENEX 2004*, Lecture Notes in Computer Science, Springer-Verlag, to appear.

[20] Kullmann, O. (2000) Investigations on autark assignments. *Discret. Appl. Math.*, **107**, 99–138.

[21] Kullmann, O. (2003) Lean clause-sets: generalizations of minimally unsatisfiable clause-sets. *Discret. Appl. Math.*, **130**, 209–249.

[22] Manber, U. (1989) *Introduction to Algorithms, A Creative Approach.* Addison Wesley Publishing.

[23] Arnborg, S., Lagergren, J. and Seese, D. (1991) Easy problems for treedecomposable graphs. *J. Algorithms*, **12**, 308–340.

[24] Reed, B., Smith, K. and Vetta, A. (2003) Finding odd cycle transversals, *Oper. Res. Lett.*, **32**, 299–301.

[25] Nemhauser, G. and Trotter, L., Jr. (1975) Vertex packings: structural properties and algorithms. *Math. Program.*, **8**, 232–248.

[26] Prieto, E. (2005) The method of extremal structure on the k-maximum cut problem. *Proc. Computing: The Australasian Theory Symposium (CATS 2005). Conferences in Research and Practice in Information Technology*, vol. **41**, pp. 119–126.

[27] Mathieson, L., Prieto, E. and Shaw, P. (2004) Packing edge disjoint triangles: a parameterized view. *Proc. IWPEC 04, Lecture Notes in Computer Science*, vol. 3162, pp. 127–137.

[28] Prieto, E. and Sloper, C. (2004) Looking at the stars, *Proc. Int. Workshop on Parameterized and Exact Computation (IWPEC ? 04)*, Lecture Notes in Computer Science, vol. 3162, pp. 138–149.

[29] Fellows, M., Heggernes, P., Rosamond, F., Sloper, C. and Telle, J.A. (2004) Finding k disjoint triangles in an arbitrary graph. *Proc. 30th Workshop on Graph Theoretic Concepts in Computer Science* (*WG '04*), Springer Lecture Notes in Computer Science. To appear.

[30] Dehne, F., Fellows, M. and Rosamond, F. (2004) An FPT algorithm for set splitting. *Proc. WG2004—30th Workshop on Graph Theoretic Concepts in Computer science*, Lecture Notes in Computer Science, Springer Verlag.

[31] Robertson, N. and Seymour, P.D. Graph minors XX Wagner's conjecture. To appear.

[32] 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.

[33] Seymour, P.D. and Thomas, R. (1994) Call routing and the ratcatcher. *Combinatorica*, **14**, 217–241.

[34] Demaine, E. and Hajiaghayi, M. (2005) Bidimensionality: new connections between FPT algorithms and PTASs. *Proc. 16th Annual ACM-SIAM Symp. Discrete Algorithms (SODA 2005).* January 23–25, pp. 590–601.

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

[36] Sloper, C. (2006) Techniques in parameterized algorithm design. qPhD Thesis, University of Bergen, (http://www.ii.uib.no/ sloper).