Parameterized Complexity

Seminar in Algorithms 186.182

Josef Eisl (0625147)
josef.eisl@student.tuwien.ac.at

January 16, 2013

1 Introduction

Complexity theory is about classifying problems into classes of tasks that are apparently *equally difficult* to solve with respect to the *resources* needed to find the solution. In our case the *worst-case* time and space consumption is of interest. The borderline between those problems that are relatively *easy* to solve, and those that are *hard*, lies between the complexity classes P and NP.

The class P contains all problems for which we can find a polynomial-time algorithm. So there is an algorithm with worst-case execution time in the order of $n^{\mathcal{O}(1)}$ where n denotes the size of the input. These problems are also called *tractable* problems. On the other side there is the class NP, which consists of all the problems that can be solved by a *non-deterministic Turing machine* in polynomial time. In a non-deterministic Turing machine the transitions from on state to another are encoded as relations and not as functions like in the ordinary variant. So for each state and symbol the non-deterministic Turing machine can choose between several possible transitions. All these choices can be viewed as a *tree* of different computation paths. In the worst-case all these paths have to be considered which leads to an upper bound on the run-time of $\mathcal{O}(2^n)$. This is referred to as the *combinatorial explosion.* From the definition of NP follows that $\mathsf{P} \subseteq \mathsf{NP}$. There is evidence that P is a proper subset of NP even though this has been an open question for many decades. A problem P is said to be NP-*hard* if it is at least as difficult as every other problem P' in NP. That means every problem P' can be reduced to the problem P by an polynomial-time algorithm. A problem is NP-complete if it is in NP and it is NP-hard. Figure 1 illustrates the connection between these classes under common complexity-theoretic assumptions.

The focus of this paper is on problems that are NP-hard. There is no hope to find an algorithm that performs significantly better than $\mathcal{O}(2^n)$ in worst-case. But on the other hand there are many algorithms for NP-hard problems that perform extremely well in practice for example the simplex method for integer linear programming by Dantzig (1963) or DPLL algorithm for Sat solving by Davis et al. (1962) to name just two. This suggests to take a closer look at a problem and its instances to find out properties that cannot be grasped by classical complexity classes.

Let us consider *conjunctive queries* in database theory. Conjunctive queries are of enormous practical importance because they are a subset of SQL queries. Most SQL queries can be rewritten in form of conjunctive queries. Basically they are SQL queries

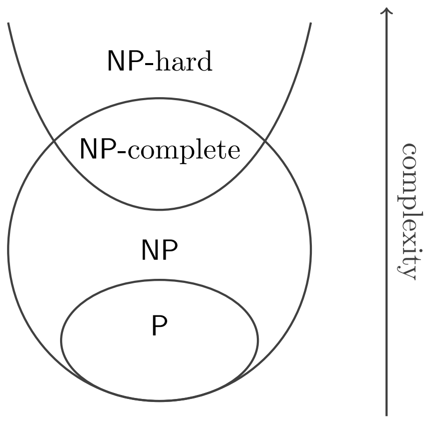


Figure 1 – P vs. NP

without aggregate functions and subqueries. The problem to solve is the following. Given a database $\mathcal{A}$ and a conjunctive query φ, does the query φ yield a non-empty result if executed on database $\mathcal{A}$? The NP-completeness of the *combined complexity*, that is with respect of the size of database $\mathcal{A}$ and the size of the query φ, has been shown by Chandra and Merlin (1977). In the real world it is almost always the case that the queries are small and the database is huge. In fact it has been shown that the whole complexity of this problem lies in the query. If the query has a low structural complexity this problem can be solved efficiently Grohe et al. (2001). This result is an explanation why relational databases are so successful in practice.

In the example above a certain property of the input instance was used to show that in this case an efficient algorithm exists. The complexity result is not only dependent of the *size of the input* but also on a *parameter*. In the following section the informal concepts of this introduction will be put into a formal framework. Afterwards a whole family of methods are presented which build upon the fundamental concepts. These so called fixed-parameter algorithms (FPA) are the main focus of this paper.

2 Basic concepts and definitions

At first we will recall the definition of a problem in the classical complexity theoretic sense.

Definition 1 (Problem)**.** *A* problem *is a* task/question *together with an* infinite set of instances*.*

The most important aspect in definition 1 is the *infinite set of instances*. If we would consider a finite number of instances one could just pre-compute all solutions. This yields a linear time algorithm, with respect to the number of input instances. There are different kinds of problems. For example optimization problems, enumeration problems or decision problems. In complexity theory decision problems, that are questions which can be answered with *yes* or *no*, are most important.

As we have seen in the introduction classical complexity theory do not take structural information about the input instance into account. We need a formal concept that allows us a more fine-grained view at a problem.

Definition 2 (Niedermeier (2006))**.** *A* parameterized problem *is a* task/question *together with an* infinite set of instances *and a* parameter*, often denoted by k.*

We look at the problem from a two-dimensional point of view. The parameterized complexity of a parameterized problem is not only dependent on the size of the input n but also on the parameter k. The parameter can be any property that classifies the input into distinct groups. In almost all cases the parameter is a positive integer. Often the size of the solution set is used for parameterization but also other properties such as treewidth of a graph are very common. With the definition of parameterized problems at hand we can define a new class of tractability.

Definition 3 (Downey and Fellows (1999); Niedermeier (2006); Flum and Grohe (2006))**.** *A parameterized problem is* fixed-parameter tractable *(*FPT*) w.r.t. parameter k if it can be computed in $f(k) \cdot n^{\mathcal{O}(1)}$ time where $f(k)$ is only depending on k.*

As we already mentioned the focus is on NP-hard problems. If such a problem falls into the class FPT as well it has an outstanding property. When considering a fixed parameter k the problem can be solved in polynomial time. It gets *tractable.* The combinatorial explosion has been shifted into the parameter. Note that there are no constraints to the $f(k)$ other than not being dependent on n. This means $f(k)$ can be huge even for small values of k. This is the reason why some FPT results are only of theoretic interest.

Another remark that should be noted is that result in the parameterized complexity theory are always with respect to a concrete parameter. One classical problem can be turned in many parameterized problems by choosing different parameters. For example one can parameterize CNF-Sat by the number of variables, number of clauses or the maximum size of the clauses. These parameterizations form three different parameterized problems and classification results must be established for each of them separately.

3 Fixed-parameter algorithms

In this section we will present a number of techniques 1) to establish FPT membership and 2) that provide us with fixed-parameters algorithms (FPA). Some of the *tools* presented here have been developed in course of parameterized complexity. Other are around for several years and have been retrofitted for this field of research. In the following sections we will describe some of the commonly used approaches. The list is far from complete but should provide an overview.

3.1 Data Reduction and Problem Kernels

If dealing with computationally hard problems the size of the input is of great importance. Before invoking a worst-case exponential algorithm the input should be reduced as much as possible. This can be done by so called *data reductions.* A data reduction is a polynomial-time pre-procession procedure. The goal is to *cut away* all the easy-to-solve parts from an input instance. What remains is the core of the problem that makes it difficult to solve. It is clear that in general this core must have the same hardness as the original problem otherwise P = NP. These technique is universal important not only for FPAs. Almost all approaches profit from reduced input size.

These data reductions are often organized as simple rules. There are *parameter-independent* rules which do not use the parameter value. The other, *Parameter-dependent*, rules need explicit knowledge about the parameter. The first kind is preferable because they can be applied in advance. If the parameter changes in course of the algorithm, for example if a one tries to do a local search for the optimal value using a decision procedure, these rules need not to be undone. In contrast parameter-dependent transformations are not valid in general if the parameter changes. Next we will consider an example for a simple data reduction.

3.1.1 Weihe's Train Problem

The problem was formulated by Weihe (1998). A railroad network is modeled as a bipartite graph $G = (S, T, E)$ with a vertex set of stations S and a vertex set of trains T. If a train $t \in T$ visits a station $s \in S$ it is connected via an edge $\{s, t\} \in E$. Informally speaking the problem is about selecting a subset of the stations $S' \subseteq S$ so that all trains $t \in T$ are visiting at least on station in S'.

Problem 1 (WEIHE'S TRAIN PROBLEM)**.** ***Instance:*** *A bipartite graph $G = (S, T, E)$ with stations S and trains T and a positive integer k.*
Question: *Is there a $S' \subseteq S$ of size k so that every train stops at a station in S'.*

Weihe showed that this problem is NP-complete by proving that it is equivalent to the HITTING SET problem which is known to be NP-complete (Garey and Johnson, 1979). Despite this negative result it is possible to compute the solution for some real-life instances in short time. This can be done using two simple data reduction rules which can shrink the input size tremendously.

Definition 4 (Weihe's reduction rules)**.** *For $s, s' \in S$ and $t, t' \in T$. $N(v)$ denotes the of neighbors of v.*

Station Rule: *if $N(s) \subseteq N(s')$ then delete s.*

Train Rule: *if $N(t) \subseteq N(t')$ then delete t'.*

The first rule is used to remove stations from the instance. If all trains that visit a station s will also stop at station s' we can remove s and all incident edges because we still can cover all trains if we take s' into the solution. The *Train Rule* tries to remove trains. If a train t' stops at all stations another train t stops we can remove t' and all incident edges because if we cover train t by a station, t' is covered as well. It would not make sense to consider t' independently. Because we remove stations from the input set we can lose some (optimal) solutions but we can still find at least one optimum with respect to the size of the solution set.

Example 1. An example instance is depicted in figure 2. At first we apply the Station Rule on $N(s_2) = \{t_2\} \subseteq N(s_1) = \{t_1, t_2, t_4\}$. To cover t_2 it is sufficient to consider only s_1 and we can remove s_2. Next we make use of the Train Rule on $N(t_2) = \{s_1, s_3\} \subseteq N(t_1) = \{s_1, s_3, s_5\}$ to delete t_1. In course of covering t_2 we will also cover t_1 so we do not have to consider it separately. Finally we can apply the Station Rule $N(s_4) = \{t_3\} \subseteq N(s_3) = \{t_2, t_3\}$ once more to remove s_4.

Weihe applied his reduction rules to real world instances from the European railroad network containing several thousands stations and trains. The remaining instance consisted of a set of disconnected components with a size small enough to be solved by enumeration.

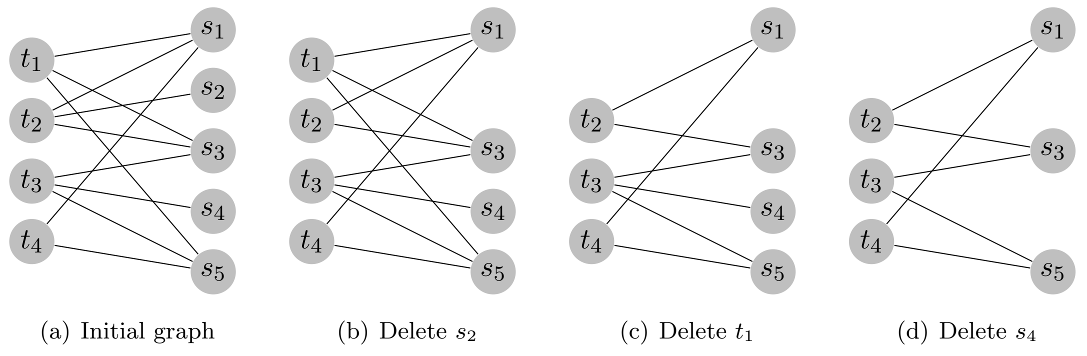


Figure 2 – Weihe's Train problem

Although these rules work very well in practice there is a drawback from the theoretical point of view. The effectiveness of the reduction can not be guaranteed (Niedermeier, 2006). Thinking in terms of classical complexity theory it seems quite hard to find an useful definition to *measure the quality* of a data reduction. In the parameterized setting, on the other hand, this can be expressed very naturally and leads us to the definition of a *problem kernel* which will be discussed next.

3.1.2 Problem Kernels

Definition 5 (Downey and Fellows (1999); Niedermeier (2006); Flum and Grohe (2006))**.** *Reduction to a* problem kernel *means to replace the instance (I, k) by a reduced instance (I', k') such that*

1. *$k' \le k$ and $|I'| \le g(k)$ where $g(k)$ is a function solely depending on k,*
2. *(I, k) is a positive instance* iff *(I', k') is one,*
3. *the transformation from (I, k) to (I', k') must be computable in polynomial time.*

The first property guarantees that the parameter k' is not increased and that the size of the new instance I' is bounded by the original parameter k. Property 2 ensures that both problems are equivalent in terms of the decision problem. The reduction must be computable in polynomial time otherwise the complexity can be hidden in the transformation routine. This is guaranteed by property 3.

Note that to comply with definition 5 a solution for a reduced instance (I', k') is not required to yield a solution for the original input. But almost always it can be used to construct a solution for the initial instance.

On closer examination the Definition 3 (FPT) and Definition 5 (problem kernel) look very similar. In fact it can be shown that both notations are equivalent.

Theorem 1. *A parameterized problem is in* FPT *iff there exists a problem kernel.*

The *only-if* direction (problem kernel $\Rightarrow$ FPT) is easy to see. If there is a problem kernel of size $f(k)$ enumerate all solutions in a brute force manner. The reverse direction is not trivial. A proof is omitted here but can be found in the book of Flum and Grohe (2006, Theorem 1.39) or in Niedermeier (2006, Proposition 7.2).

In the next section we will present as a concrete example a kernel for the VERTEX COVER problem.

3.1.3 Buss's reduction to a Problem Kernel

The VERTEX COVER problem is one of the most important FPT problems (Niedermeier, 2006). The problem is easy to understand and many approaches to FPA can by presented using VERTEX COVER as an example. The problem is defined as follows.

Problem 2 (VERTEX COVER)**.** ***Instance:*** *A graph $G = (V, E)$ and a nonnegative integer k.*

Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

The first concrete fixed-parameter algorithm we want to present is attributed to Sam Buss (1989) and is called *Buss's reduction to a problem kernel* (Niedermeier, 2006). It consists of three simple rules.

Definition 6 (Buss's reduction to a problem kernel)**.**

VC1 *Remove all isolated vertices.*

VC2 *For every vertex with degree 1, put the neighbor into the cover and delete both vertices from V.*

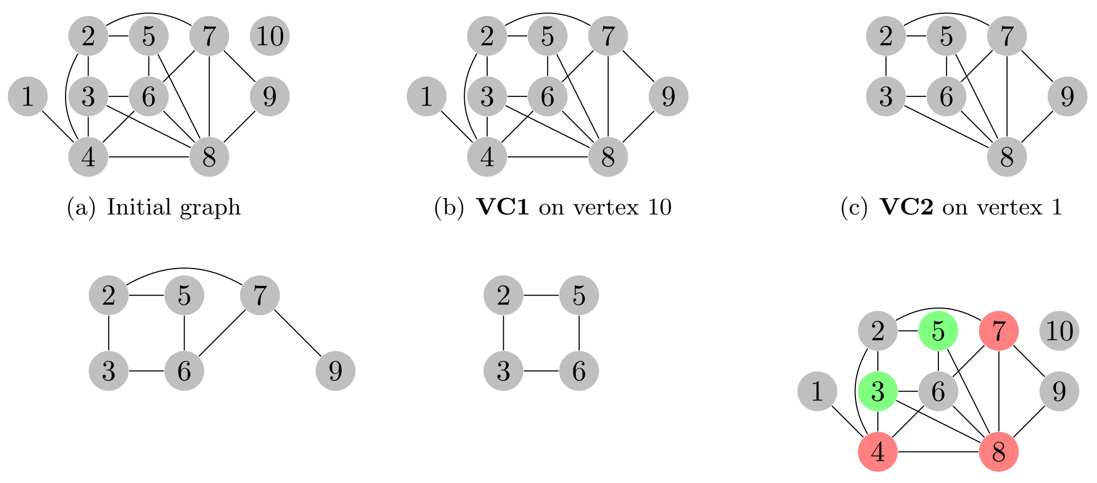
VC3 *For a vertex with degree $> k$, put this vertex into the cover and delete it form the graph.*

The first rule removes all isolated vertices from the instance because they have no influence on the vertex cover. The second rule removes every vertex v which has only one neighbor w. To cover the edge $\{v, w\}$ one of these two vertices muss be in the solution set. Putting v into the cover will never produce a better solution than choosing w. So we can remove v, w and all incident edges from the graph and add w to the solution set. Additionally we have to decrement k by one because we have already fixed one member of the cover. The last rules removes every vertex v which has more than k neighbors. To cover all edges incident to v one could either add v or all vertices adjacent to v (which are denote by $N(v)$) to the cover. We are not allowed to put more than k vertices into the solution set but $|N(v)| > k$. Therefore v must be part of the cover.

Example 2. An example graph G is depicted in Figure 3(a). Let us apply Buss's reduction on the instance $(G, k = 5)$. The steps are illustrated in Table 1.

Rule	Vertex	Removed	k	Cover	Figure
–	–	–	5	$\{\}$	3(a)
VC1	10	10	5	$\{\}$	3(b)
VC2	1	$1, 4$	4	$\{4\}$	3(c)
VC3	8	8	3	$\{4, 8\}$	3(d)
VC2	9	$7, 9$	2	$\{4, 7, 8\}$	3(e)

Table 1 – Application of Buss's reduction. The first and the second column indicates which rule is applied to which vertex. The third column shows which vertices are removed in this step (together with all incident edges). The columns k and Cover show the value of the parameter and the partial cover after the step. The last column contains a reference to the subfigure that illustrates the graph after the application.



(a) Initial graph (b) **VC1** on vertex 10 (c) **VC2** on vertex 1

(d) **VC3** on vertex 8 (e) **VC2** on vertex 9 (f) Solution

Figure 3 – Buss's reduction

What remains is the problem kernel $(G', k = 2)$. To give the correct yes/no answer to the decision problem one still has to find a cover of size 2 in the remaining graph G'. This can be done with an exhaustive search or any other *exact* algorithm for the parameterized Vertex Cover problem. Figure 3(f) shows one possible solution. The original input $(G, k = 5)$ is a yes-instance of the problem.

If all rules are applied exhaustively we can make some statements about the problem kernel $(G' = (V', E'), k')$.

Proposition 1. *If at some point $k' = 0$ and $E' \neq \emptyset$ the original instance does not have a solution of size $\leq k$.*

Proof. We have already k vertices in the cover but there are still some edges in E which are not covered yet. □

Corollary 1. *If $k' = 0$ and $E' \neq \emptyset$ then $k + |E'|$ gives an upper bound on the optimal solution.*

Proof. We know that each edge $e = \{v, w\} \in E'$ is not covered yet. So we can arbitrary choose one endpoint of e and add it to the solution. The result is a vertex cover of size $k + |E'|$ for the graph G. □

From now on we assume that $k' \geq 0$.

Corollary 2. *There is no vertex with degree $> k$.*

Proof. This is a direct consequence of VC3. □

Proposition 2. *If $|E'| > k^2$ then there is no solution.*

Proof. By Corollary 2 we know that each vertex can cover at most k edges. A solution can contain at most k vertices. Therefore we can cover at most k^2 edges. □

Corollary 3. *Every vertex has at least degree 2.*

Proof. All vertices with degree 0 (1) are eliminated by rule VC1 (VC2). □

Proposition 3. *If $|V'| > k^2$ then there is no solution.*

Proof. By Corollary 3 we know that each vertex has at least two neighbors. This implies that if there are more than k^2 vertices there must be more than k^2 edges. By Proposition 2 we know that there cannot be a solution in this case. □

Theorem 2. *Applying Buss's reduction to an instance $(G = (V, E))$ yields a problem kernel $(G' = (V', G'), k')$ where $|V'| \leq k^2$, $|E'| \leq k^2$.*

Proof. The size of the kernel is a direct consequence of Proposition 2 and Proposition 3. As already mentioned the size is only guaranteed for yes-instances but we can check the bounds and return a trivial no-instance if these properties are violated. Therefore the size is valid even for no-instances. In order to fulfill the complete problem kernel definition we still must ensure that $k' \leq k$ and that it is computable in polynomial time. The first property follows directly from the reduction rules. The parameter is never incremented. It can be shown that the reduction has a polynomial runtime (Niedermeier, 2006). □

The complete FPA using this kernel and a search tree technique for finding the final solution has a runtime of $\mathcal{O}(kn + 2^k \cdot k^2)$ (Downey and Fellows, 1999).

3.1.4 Conclusion problem kernels

We have seen in the examples in the previous sections that problem kernels and in general data reductions are powerful tools even outside complexity theory. Problem kernels are a formal framework to prove properties of reductions which would be difficult to show in a classical setting. In both examples introduced in this part of the paper the reduction rules are very simple but the proof of their effectiveness is demanding (Buss' reduction) or even impossible (Weihe's train problem).

In the proof of Theorem 1 (which was omitted in this work) one can see that there are trivial (in sense of huge) kernels for all problems in FPT. So some of the kernelization results are not useful at all. Even in Example 2, where Buss's reduction yields a tiny kernel we can only guarantee a kernel of size $|V'| \leq k^2 = 25, |E'| \leq k^2 = 25$ which is way bigger then the initial graph.

For Vertex Cover there is a kernelization by Chen et al. (2001) which is based on a famous theorem by Nemhauser and Trotter (1975) which yields a kernel of size $|V'| \leq 2k$. Due to connections to approximation theory there is evidence that there does not exist a smaller kernel than that (Niedermeier, 2006; Khot and Regev, 2003).

One problem of the definition of problem kernels from the practical point of view is that the size of the problem kernel can be huge. Therefore the concept of *polynomial-size problem kernels* has been introduced (Flum and Grohe, 2006). It refines the definition so that the size is bound by $k^{\mathcal{O}(1)}$. This notion has been used to describe *efficient problem kernels*. Again, it are the negative results that dominate this field (Bodlaender et al., 2008). The outstanding masters thesis by Misra (2009) provides a great reference on polynomial kernels.

3.2 Depth-Bounded Search Trees

The method of *depth-bounded search trees* is one of the most important approaches to attack (the core of) hard tasks. It is based on the observation that many combinatorial problems can be solved by navigating through the search space in a tree-like fashion. For NP-hard problems the search space is often a tree with exponential size. This is not a new idea for solving computationally hard problems. For instance the famous DPLL algorithm (Davis et al., 1962) uses such an approach.

In the course of fixed-parameter algorithms the goal is to bind the depth of the search tree by a function solely depending on k. If it is possible to design such a search tree this yields a FPA. The rule of thumb is again: a small k leads to a short runtime.

Let us again consider the VERTEX COVER-problem (Problem 2). In the classical setting the naïve search tree might branch on the vertices. The choice to made is whether a vertex v is part of the cover or not. This forms a search tree of size $\mathcal{O}(2^n)$.

By recalling the definition of the parameterized VERTEX COVER problem it is clear that a positive solution is required to contain at least one endpoint of each edge $\{v, e\} \in E$. So instead of branching on the vertices one can branch on edges. For an edge $\{v, w\} \in E$ put one of the endpoints into the cover, remove it and all incident edges from the graph and repeat the procedure. After at most k steps we have either found a cover or can stop because we are not allowed to put anymore nodes into the cover. The corresponding search tree has a size of $\mathcal{O}(2^k)$.

Although, the $\mathcal{O}(2^k)$ search tree is sufficient for showing FPT membership it might be practically advantageous to find an even better (smaller) search tree. The key to smaller search trees are case distinctions in the branching step. Instead of only adding (removing) only one vertex to the cover (from the graph) several vertices can be considered by exploiting certain properties of the graph.

One possibility is to define branching rules according to the degree of a vertex.

1. For a vertex of degree 1 put the neighbor into the cover.
2. For a vertex v of degree 2 either
   - both neighbors, or
   - v together with all the neighbors of the neighbors

   are in the cover.
3. Vertex v of degree at least 3 either
   - v is in the cover, or
   - all its neighbors are.

Visual representation of these rules are depicted Figure 4. The first rule is similar to the first rule in Buss's reduction (Definition 6). The last rule is the *catch all* rule and can be seen as an generalization of the edge-branching described previously. The correctness of the rule for degree 2 vertices is not so easy to see. Although we might lose some optimal solution it can be shown that one of these sub cases will also lead to a solution at least as good. A more detailed argumentation can be found in Hüffner et al. (2008). The resulting search tree has a size bounded by $\mathcal{O}(1.47^k)$. The result can be computed using the mathematical tool of *linear recurrences with constant coefficient.* A walk-though guide on the presented bound and how it is computed can be found in the book of Niedermeier (2006, Chapter 8).

The basis of the exponential factors can be improved even more. The main approach is to use finer case distinctions and to *"discuss away certain situations"* (Niedermeier,

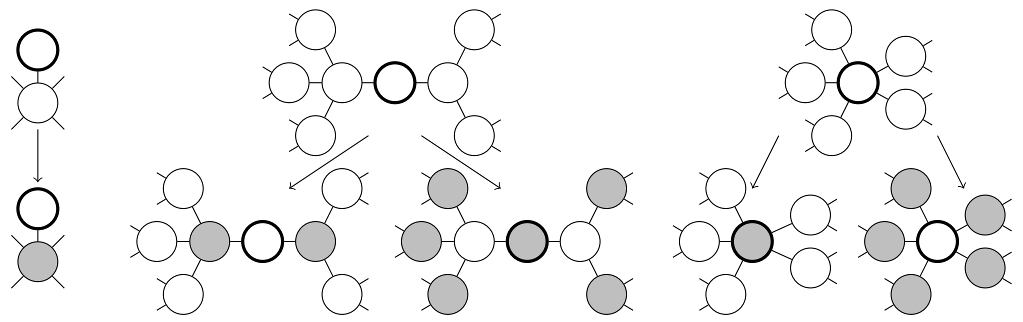


Figure 4 – Depth-bounded search tree

2006, p. 101). The best search tree for the Vertex Cover-problem currently known was proposed by Chen et al. (2006). The size is bounded by $\mathcal{O}(1.28^k)$.

Regarding practical application it is worth noting that the $\mathcal{O}$-notation hides the administrative overhead caused by extensive case distinctions. In many cases approaches with fewer branching rules perform better in practice (and are easier to implement) (Niedermeier, 2006).

3.2.1 Interleaved kernelization

It was already remarked that a preceding kernelization can improve the runtime of any approach. The method of depth-bounded search trees allows us to kernelize in between branching steps. Empirical studies have shown that this approach is highly beneficial in practical applications. Some times it is also possible to reduce the search tree size because certain cases can not occur after kernelization and therefore can be neglected. The overhead of the data reduction is compensated by the decreased size in most cases. Niedermeier (2006) also argues that interleaving is not only possible but necessary to achieve certain improvements.

3.2.2 Conclusion search trees

Depth-bounded search trees are a key tool for designing fixed-parameter algorithms. In most cases the analysis of the runtime and the correctness proof are the challenging tasks. Interleaving kernelization and branching can provide provable benefits.

It is also worth mentioning that search algorithms are very suitable for parallel machines because each branch nodes is split into several independent sub problems which can be distributed easily.

3.3 Iterative Compression

Iterative compression is a technique first proposed by Reed et al. (2004) for showing FPT membership and deriving fixed-parameter algorithms. It is not as general as kernelization or depth-bounded search trees but it have been shown that it is a valuable method for theoretic as well as practical results (Hüffner, 2005).

Iterative compression can be used for minimization problems which are parameterized by the size of the solution set. The core concept is the so-called *compression routine*.

Definition 7 (Hüffner et al. (2008))**.** *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.*

Note that it is not required to find an optimal solution but only a better one if one exists. To find an optimal solution one starts with a trivial subinstance of the problem instance together with a solution. In every step the instance is increased towards the original instance. After that the compression routine is used to find the optimal solution for the partial problem by iterative application. The key proposition is that if the compression routine is fixed-parameter tractable, the whole algorithm is in FPT too (Hüffner et al., 2008).

When using iterative compression the challenges are 1) to find a compression routine and 2) to show its FPT membership.

The Vertex Cover-problem will serve again as a demonstration model for iterative compression. Algorithm 1 shows the entry point of the iterative compression routine. We start with the empty vertex set and the trivial cover, also the empty set. For the loop starting at line 4 we can assert the invariant that C is a minimum vertex cover for the graph $G[V']$[a]. This property clearly holds at the loop entry. The empty set is the minimum vertex cover for the empty graph. In each iteration of the loop in line 4 we add an arbitrary vertex v to the graph and to the cover. It is easy to see that C is still a vertex cover of $G[V']$. All edges that were introduced by the new vertex v are covered because v is also part of C. But we cannot ensure that it is a minimum cover. In order to achieve minimality the compression routine is called in line 9. This is repeated until it reaches a fix-point, that is the minimum vertex cover for the induced subgraph. Therefore the invariant holds. Furthermore the loop in line 7 will end after at most $|C|$ iterations because the loop is only continued if the compression routine returns a covers smaller than the current solution. If the optimal solutions for the current subgraph already bigger than k there is no solution to the original problem.

Let us now take a close look at the compression routine in Algorithm 2. The idea is that we want to find a better cover C' which is modification of the current cover C. The vertices in $Y \subseteq C$ remain in the cover and the vertices in $S = C \setminus Y$ are replaced by vertices from $V \setminus C$ that were not part of the cover.

Now we just consider all $2^{|C|}$ partitions of C into Y and S. This is done in the loop on line 3. Two illustrations of the sets that are used in this loop are given in Figure 5 which might be helpful when studying the algorithm. Note that we have already decided to take the vertices in Y into the cover so it is sufficient to only consider all uncovered edges (line 6). In line 7 we check if there is an edge where both endpoints are in S. If this is the case we can not find a cover without using a vertex in S so we can continue with the next partition. If this is not the case we can construct the vertex cover by choosing one endpoint for each remaining edge. Note that at line 10 we know that one endpoint $v \in S$ and the other in $V \setminus C$. The case that both endpoints are in S is covered by the check in line 7. It is also not possible that both endpoints are in $V \setminus C$ because this would contradict the precondition that C is a vertex cover of G. It is easy to see that the vertex cover that is constructed in this way is the optimum for the current partition. Every vertex in the cover is needed. The vertices in Y by definition and the edge endpoints to fulfill the vertex cover property. The smallest cover of all partitions will be returned. If no better cover is found the original solution is returned. It is also

[a] $G[V']$ denotes the subgraph of G *induced* by V'. That is if $G = (V, E)$ and $V' \subseteq V$ then $G[V'] = (V', \{\ \{v, w\} \in E \mid v, w \in V'\})$.

Algorithm 1: IterativeCompressionVC

```
Data: A graph $G = (V, E)$ and $k$.
Result: A vertex cover $C \subseteq G$ with $|C| \leq k$ or nil.
begin
    $V' \longleftarrow \emptyset$
    $C \longleftarrow \emptyset$
    foreach $v \in V$ do
        $V' \longleftarrow V' \cup \{v\}$
        $C \longleftarrow C \cup \{v\}$
        repeat
            $C' \longleftarrow C$
            $C \longleftarrow$ CompressVC($G[V'], C$)
        until $|C'| = |C|$
        if $|C| > k$ then
            return nil
    return $C$
```

a minimum vertex cover for G because all possibilities where covered, from taking all vertices from C to taking non of them, and constructed the optimum cover for all these partitions (Niedermeier, 2006).

Let us analyze the runtime of the algorithm. For every partition we have to check at most all edges which is bounded by $m = |E|$. Checking all partitions can be done in $\mathcal{O}(2^{|C|} \cdot m)$ time. In Algorithm 1 we abort if $|C| > k$ so the compression routine is upper bound by $\mathcal{O}(2^k \cdot m)$. Note that the compression routine finds the optimum solution at once so the loop in line 7 of Algorithm 1 is not really necessary in our case. Anyway assumed that the compression routine would only yield a smaller solution this loop would be bound by $\mathcal{O}(|C|)$ (and therefore by k). The outer loop is executed for all vertices $n = |V|$. Therefore the iterative compression algorithm for Vertex Cover find an optimal solution within $\mathcal{O}(2^k \cdot m \cdot n)$ which is clearly in FPT.

3.3.1 Conclusion iterative compression

It is worth noting that the iterative compression approach for Vertex Cover is not competitive to the search tree algorithms described previously but there are examples where iterative compression yields algorithms that are useful in practice. There are even problems where iterative compression is the only known way for showing FPT membership (Niedermeier, 2006; Hüffner et al., 2008).

In contrary to problem kernels, where we know that there exists a (possibly trivial) problem kernel for every problem in FPT, the same is not true for iterative compression. It is not clear that a compression routine exists for every FPT problem (Hüffner et al., 2008). Therefore, the challenge is to find a suitable compression routine.

Algorithm 2: CompressVC

```
Input  : A graph $G = (V, E)$.
Input  : A vertex cover $C \subseteq V$ for $G$.
Output: A (minimal) vertex cover $C' \subseteq V$ with $|C'| \leq |C|$.
begin
    $C' \longleftarrow C$
    foreach $Y \subset C$ do                                   // for all partitions of $C$
        $C'' \longleftarrow Y$                                 // $Y$ will remain in the cover
        $S \longleftarrow C \setminus Y$                       // $S$ is removed from the cover
        $G' = (V', E') \longleftarrow G[V \setminus Y]$        // consider the remaining graph
        if $\exists v, w \mid \{v, w\} \in E' \wedge v, w \in S$ then   // both endpoints are in $S$
            continue                                           // no cover in this partition
        for $\{v, w\} \in E'$ do                               // ensure cover properties
            // w.l.o.g. we can assume that $v \in S$ and $w \in V \setminus C$
            $C'' \longleftarrow C'' \cup \{w\}$
        if $|C''| < |C'|$ then                                 // better cover found
            $C' \longleftarrow C''$
    return $C'$
```

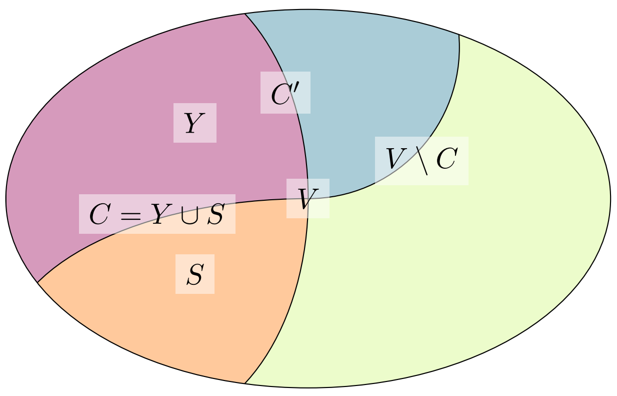
3.4 Further Techniques

In this section some further fixed-parameter techniques are presented. Due to lack of space only the key aspects are mentioned. Every topic would deserve a paper on its own.

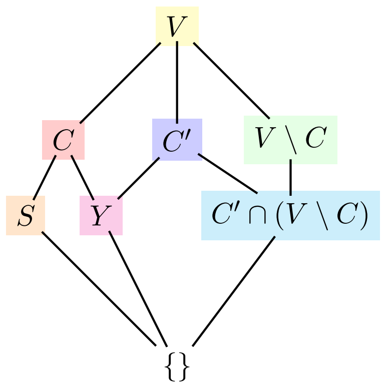
3.4.1 Tree Decomposition

Tree decomposition is a very important method for finding new FPT results. The motivation is that many problems that are hard on general graphs can be solved efficiently on trees. Tree decompositions of graphs were discovered by Halin (1976) and later reinvented by Robertson and Seymour (1984). It can be seen as a "tree representation" of a graph. There are different decompositions of the same graph. The *width* of a decomposition is somehow a measure how "precise" a representation is. The *treewidth* of a graph is the minimum width of all possible tree decompositions. It is a measure how tree-like a graph is. Trees have a treewidth of 1. The FPA approach is to construct the tree decomposition for instances that are *almost* trees and solving the problem efficiently on that decomposition. The treewidth is used as parameter value.

There is a major setback in this approach. Finding a tree decomposition of minimum width is a hard problem. Even the task of finding a decomposition of size at most k is NP-complete. Bodlaender (1996) proposed a linear time fixed-parameter algorithm for this problem but the hidden constants are preventing practical application (Niedermeier, 2006). But never the less it is a theoretic basis for FPT results. Another important remark has to be made. There are many heuristic and approximation approaches which are able to find "good" tree decompositions in short time. This is sufficient for the tree decomposition based FPAs. The price that one has to pay is that a non-optimal tree decomposition increases the runtime of the algorithm.



(a) *Venn* diagram



(b) *Hasse* diagram

Figure 5 – Visualizations of the vertex sets that occur in the compression routine (Algorithm 2) of the iterative compression algorithm for Vertex Cover. (A label on the border between two or more subsets indicates that the it denotes the union of them.)

A deeper introduction into treewidth as FPT tool is given in all tree books on the topic by Niedermeier (2006); Flum and Grohe (2006); Downey and Fellows (1999).

3.4.2 Dynamic Programming

Dynamic programming is a well known technique where the goal is to prevent recomputation by storing intermediate results. In most cases search trees have parts with *overlapping, independent subproblems*. For example recall the simple $\mathcal{O}(2^k)$ depth-bounded search tree for Vertex Cover. We branch on edges but in both paths the same subgraph might occur. It does not matter which vertices are already in the cover. Only the remaining graph is of interest. If we store the result in a look-up table we do not need to recompute it. The major challenge is to decide which results to store and which to recompute. It is a trade of space for time. If the space needed by the table can be bound by the parameter it is possible to create fixed-parameter algorithms. One example is Binary Knapsack with parameter weight W which can be done on $\mathcal{O}(W \cdot n)$. Further pointers to applications of dynamic programming in course of fixed-parameter algorithms can be found in Niedermeier (2006); Hüffner et al. (2008).

3.5 Even further

Due to restriction in time and space this paper only scratches the surface of parameterized algorithms. In this section some remarks are made on topics that deserve attention but did not yet find their way into this paper.

The only problem presented here was the Vertex Cover problem but there many other problems where interesting parameterized solutions exists.

An other important note is that there is a close connection to approximation. These two topics complement each other in various ways. For example kernelizations can be used as approximation algorithms and results from approximation theory give evidence on lower bounds for fixed-parameter approaches.

In all examples it is assumed that the parameter is *small enough* to claim that the FPT approach is practically useful. But in many cases one can not hope that a certain parameter is small. For theoretic results this does not matter but in practice this is a major problem. The good news it that there are methods to tackle that problem. For example in MaxSat it is trivial to satisfy half of the clauses. Guess an assignment and check if the half of the clauses are fulfilled. If not just invert the assignment. So instead of parameterizing the overall number of satisfied clauses the difference to the trivial case can be used as parameter. Niedermeier (2006, Chapter 5) spends a whole chapter on this important topic.

This paper is about the positive results of parameterized complexity theory: fixed-parameter algorithms. But as it is in classical complexity theory the negative results are at least as interesting if not even more. This *parameterized complexity theory* consist of several hierarchies and classes that are build upon conjectures such as $\mathsf{P} \neq \mathsf{NP}$ and similar. It gives an even more detailed view of the parameterized complexity of a problem.

4 Bibliographical remarks

Downey and Fellows laid the corner stone of parameterized complexity theory by proposing the central definitions in the mid-90s. Their monograph (Downey and Fellows, 1999) collects all this knowledge from a decade of research in one book.

As the title *Invitation to Fixed-Parameter Algorithms* suggests the focus of Niedermeier (2006) is on application and algorithm design. It is almost written in a textbook manner which spends much time in communicating intuition and enthusiasm about the topic. It is a great springboard to jump into the deep of parameterized complexity theory. It also served as the main inspiration for this paper.

Flum and Grohe (2006) on the other hand gives a comprehensive reference book on parameterized complexity theory. Rather than giving a long speech of motivations it presents the concrete mathematical facts in form of definitions, theorems and proofs. This makes it a great basis for everyone how is working on intractable parameterized problems.

The Computer Journal Special Issue on Parameterized Complexity features several outstanding articles that provide a broader, more up to date view on the field of parameterized complexity theory. Especially the inspiring foreword by Downey et al. (2008) as well as the two FPA overview articles by Hüffner et al. (2008) and Sloper and Telle (2008) give a short but fascinating insight into this field.

Last but no least the International Workshop on Parameterized and Exact Computation (IPEC) is maybe the most important annual event with the focus on the positive results and indicates the direction of the current development in this, still very dynamic, area.

5 Conclusion

In parameterized complexity the focus is on the question: What makes the problem computationally difficult? (Downey and Fellows (1999))

Parameterized complexity is a very young and active direction of research. It takes a multi dimensional point of view on the problem. Not only the instance size but also other structural parameters matter. It can be used to find out where the hardness of a problem yields from. It also narrows the gap between theory an practice. On the one hand it can give us proofs why certain approaches work in practice. On the other hand, and even more important, it can provide new, exact algorithms for hard problems. In the future we will see many more applications of the results from parameterized complexity.

This is a subject that every computer scientist should know about.
(Foinn Murtagh, Editor-in-Chief of The Computer Journal)

References

H. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on computing*, 25(6):1305–1317, 1996.

H. Bodlaender, R. Downey, M. Fellows, and D. Hermelin. On problems without polynomial kernels. *Automata, languages and programming*, pages 563–574, 2008.

S. Buss, 1989. unpublished.

A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 77–90, New York, NY, USA, 1977. ACM. doi: 10.1145/800105.803397.

J. Chen, I. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41(2):280–301, 2001.

J. Chen, I. Kanj, and G. Xia. Improved parameterized upper bounds for vertex cover. In R. Královič and P. Urzyczyn, editors, *Mathematical Foundations of Computer Science 2006*, volume 4162 of *Lecture Notes in Computer Science*, pages 238–249. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-37791-7. doi: 10.1007/11821069_21.

G. Dantzig. Linear programming and extensions. 1963.

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962. ISSN 0001-0782. doi: 10.1145/368273.368557.

R. Downey and M. Fellows. *Parameterized complexity*, volume 3. Springer New York, 1999.

R. G. Downey, M. R. Fellows, and M. A. Langston. The computer journal special issue on parameterized complexity: Foreword by the guest editors. *The Computer Journal*, 51(1):1–6, 2008. doi: 10.1093/comjnl/bxm111.

J. Flum and M. Grohe. *Parameterized complexity theory*, volume 3. Springer Berlin, 2006.

R. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W. H. Freeman, 1979.

M. Grohe, T. Schwentick, and L. Segoufin. When is the evaluation of conjunctive queries tractable? In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, STOC '01, pages 657–666, New York, NY, USA, 2001. ACM. ISBN 1-58113-349-9. doi: 10.1145/380752.380867.

R. Halin. S-functions for graphs. *Journal of Geometry*, 8:171–186, 1976. ISSN 0047-2468. doi: 10.1007/BF01917434.

F. Hüffner. Algorithm engineering for optimal graph bipartization. In S. Nikoletseas, editor, *Experimental and Efficient Algorithms*, volume 3503 of *Lecture Notes in Computer Science*, pages 240–252. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-25920-6. doi: 10.1007/11427186_22.

F. Hüffner, R. Niedermeier, and S. Wernicke. Techniques for practical fixed-parameter algorithms. *The Computer Journal*, 51(1):7–25, 2008. doi: 10.1093/comjnl/bxm040.

S. Khot and O. Regev. Vertex cover might be hard to approximate to within 2- epsiv;. In *Computational Complexity, 2003. Proceedings. 18th IEEE Annual Conference on*, pages 379 – 386, july 2003. doi: 10.1109/CCC.2003.1214437.

N. Misra. Infeasibility of polynomial kernelization. Master's thesis, The Institute of Mathematical Sciences, 2009.

G. Nemhauser and L. Trotter. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8(1):232–248, 1975.

R. Niedermeier. *Invitation to fixed-parameter algorithms*, volume 3. Oxford University Press, 2006.

B. Reed, K. Smith, and A. Vetta. Finding odd cycle transversals. *Operations Research Letters*, 32(4):299 – 301, 2004. ISSN 0167-6377. doi: 10.1016/j.orl.2003.10.009.

N. Robertson and P. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

C. Sloper and J. A. Telle. An overview of techniques for designing parameterized algorithms. *The Computer Journal*, 51(1):122–136, 2008. doi: 10.1093/comjnl/bxm038.

K. Weihe. Covering trains by stations or the power of data reduction. pages 1–8, 1998. doi: 10.1.1.57.2173.