Eötvös Loránd University
Faculty of Science

Ágoston Weisz
Mathematics MSc

FPT Algorithms

Msc Thesis

Supervisor: Zoltán Király
Departement of Computer Science

Budapest, 2016

Acknowledgement

I would like to thank my supervisor, Zoltán Király for introducing me to this interesting topic, for the consultations and for his detailed comments on this thesis.

Contents

Preface

It is a well-known fact that deciding whether a graph has a clique of size 1000 is in P. However, the known algorithm is too slow to allow computers to actually compute it. On the other hand, finding a vertex cover of size k, where k is part of the input is NP-hard, but there are algorithms that can compute it very efficiently for small k's. Fixed parameter tractable algorithms were introduced to make it easier to differentiate polynomial algorithms that are fast enough to be useful in practice. The difference to the classical complexity theory is that with parameterized algorithms we can analyze the complexity of algorithms in a much more detailed way. We not only express the running time as a function of the input, but we introduce one or more parameters, which are also taken into account. There were some important fixed parameter algorithms before the definition and the theory was built. Lenstra's algorithm [29] in integer linear programming and Robertson and Seymour's disjoint paths algorithm [35] are good examples of this. Only in the early 1990s, Downey and Fellows introduced and defined the basics of fixed parameter tractability [17], their book with more detailed results [18] appeared in 2012. This view has lead to a practically very well applicable theory in these days. Now, it is a topic, which produces a huge amount of papers every year. Google scholar gives 10.000 results on the key-word "fixed parameter tractable", 4000 of them are published after 2012. This shows us, how mainstream the topic is in current computer science. Three other detailed books appeared in this topic: [22], [32], [13].

In the first chapter, we can find the basic techniques shown on the example of the vertex cover problem. In the second chapter, other techniques are presented on the feedback vertex set problem. In chapter four, we can read about a randomized algorithm on the exact path problem. In chapter five, a general approach is presented on graphs with small tree-width. So far, we could learn about techniques proving that certain problems are in FPT, while in chapter six, we can find the most famous method to prove the opposite: a problem is not likely to be in FPT. In chapter seven, the author presents his own results on a group of problems about shortest paths in various graphs.

Chapter 1

Definitions

Given a graph $G = (V, E)$, we denote $n = |V|$ and $m = |E|$ throughout the whole thesis. The number of adjacent edges to a vertex v is $d(v)$. For an $X \subseteq V$, we call $G[X]$ the graph induced by X and $N(X) = \{v \in V \mid \exists x \in X : x \text{ and } v \text{ are adjacent}\} - X$. We say that X is independent, if $G[X]$ has no edges. A matching M is a set of disjoint edges.

We proceed with a couple of basic definitions used in this thesis.

Definition 1.1 *A **parameterized problem** is a language $L \subseteq \sum^* \times N$, where $\sum$ is a fixed, finite alphabet. For an instance $(x, k) \in \sum^* \times N$, k is called the parameter. The size of an instance is the number of bits in a description of the instance and is denoted by $|(x, k)|$.*

Definition 1.2 *A parameterized problem $L \subseteq \sum^* \times N$ is called **fixed-parameter tractable** (**FPT**), if there exists an algorithm A (called a fixed-parameter algorithm), a computable function $f : N \to N$, and a constant c such that, given $(x, k) \in \sum^* \times N$, the algorithm A correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^c$. The complexity class containing all fixed-parameter tractable problems is called* FPT.

Definition 1.3 *A parameterized problem $L \subseteq \sum^* \times N$ is called **slice-wise polynomial** (**XP**), if there exists an algorithm A and two computable functions $f, g : N \to N$ such that, given $(x, k) \in \sum^* \times N$, the algorithm A correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^{g(k)}$. That means for every fixed constant k, algorithm A runs in polynomial time. The complexity class containing all slice-wise polynomial problems is called* XP.

Chapter 2

Vertex cover

In this chapter, we are going to see a wide range of methods for solving the vertex cover problem. This serves also as an introduction, where we can find the basic techniques for the topic. The current best algorithm of Chen, Kanj and Xia [10] runs in $O(1.2738^k + kn)$ time.

Definition 2.1 *Let G be a graph and $X \subseteq V(G)$. We say that X is a **vertex cover**, if for every edge of G at least one of its end vertices lies in X. For the vertex cover problem, we are given a graph G and a parameter k. We have to decide if G has a vertex cover of size at most k.*

2.1 Kernelization

The main idea is to solve an NP-hard problem by preprocessing it first. A **reduction rule** for a parameterized problem Q is a function $\phi : \sum^* \times N \to \sum^* \times N$ that maps an instance (I, k) of Q to an equivalent instance (I', k') of Q such that ϕ is computable in time polynomial in $|I|$ and k. We say that two instances of Q are equivalent whenever $(I, k) \in Q$ if and only if $(I', k') \in Q$.

Definition 2.2 *A **kernel** for a parameterized problem Q is a reduction rule such that for the produced instance (I', k') the inequality $|I'| + k' \leq f(k)$ holds, for some computable function $f : N \to N$. A **polynomial kernel** is a kernel of size $p(k)$, where p is a polynomial.*

If the instance is big in terms of the parameter, we can take its kernel, where the new, equivalent instance is thought as being 'small' compared to the new parameter k'. It is particularly useful when $k' \leq k$, as at an intuitive level, this makes the problem easier. In this thesis, we will only deal with this type of kernelization.

First we prove a very important theorem about kernelization.

Theorem 2.3 *[7] Every problem $Q \in FPT$ admits a kernel.*

Proof. From $Q \in FPT$ follows that there is an algorithm A, which runs in time $f(k) \cdot |(I,k)|^c$ for a given instance (I,k) and returns whether $(I,k) \in Q$. The kernelization algorithm starts with running A for at most $|I|^{c+1}$ time. If it terminates, we can return if it is a yes- or a no-instance. Note that $|I|^{c+1}$ is polynomial in the input size. If A does not end within $|I|^{c+1}$ steps, we stop it and return (I,k) as the kernel. In this case $|I|^{c+1} < f(k) \cdot |(x,k)|^c$, so $|I| < f(k)$. This means, that for the computable $g(k) = f(k) + k$, we have $|I| + k \leq g(k)$, which was the definition of a kernel. ■

This example also shows us that kernels might be quite large. So a good question for a problem might be whether it admits a polynomial kernel.

We can find a kernelization algorithm for the above defined vertex cover problem from Buss and Goldsmith in [6]. Let G and k be given. First observe that if G has an isolated vertex v, we can remove it, since it does not change the solution. Also, if for a vertex v $d(v) > k$, we can create a new instance: $G' = G - v$ and $k' = k - 1$. This is because we have to put v into the vertex cover. If not, we would have to add all of its neighbors, but that would be already more than k vertices.

(i) If a vertex v has $d(v) = 0$, remove it.

(ii) If a vertex v has $d(v) > k$, remove it and decrement k by one.

For as long as possible, we apply one of these rules. This algorithm is obviously polynomial in the input size. We can even do it in $O(n+m)$ time, if we loop through all the vertices and process the ones of degree greater than k. After that we loop through all the vertices again and process the ones of degree zero. Let us call (G^*, k^*) the equivalent instance, we got this way.

Lemma 2.4 *If (G^*, k^*) is a yes-instance then $V(G^*) \leq k^*(k^*+1)$.*

Proof. (G^*, k^*) is a yes-instance, so there is a vertex cover X of size maximum k^*. Because of (ii), for every $x \in X$, $d(x) \leq k^*$. (i) implies that there are no isolated vertices, so $V(G^*) = |X| + V(G^* - X) \leq k^* + k^* \cdot k^*$. ■

So far, we reached a polynomial kernel with an $O(n+m)$ algorithm.

Corollary 2.5 *The vertex cover problem is in FPT.*

Proof. First apply the kernelization algorithm to the vertex cover problem and then solve it on the kernel with a brute force algorithm of running time $O(f(k))$, where f is a computable function. ■

For other kernelizations of the vertex cover problem, see [1].

2.2 Bounded search tree

We are going to solve the vertex cover problem in a different way. Also we are going to apply this method to the previous section's vertex cover kernel to achive a fast algorithm. Here we want to solve a modified version of the vertex cover problem: given an $X \subseteq V(G)$ and a parameter k, we would like to find a vertex cover $C \supseteq X$ of size at most k. Melhorn designed a recursive algorithm in [30] for this problem. We start the algorithm with the empty set X.

(i) If there are no edges remaining in $G[V - X]$, return X.

(ii) If $|X| \geq k$ return "NO".

(iii) Take an edge uv whose end vertices are disjoint from the elements X, and run the program recursively on $X' = X + u$, $G' = G - u$ and on $X' = X + v$, $G' = G - v$. If any of these returned "YES", return with "YES", else return "NO".

The proof of the correctness relies on the observation that for every edge in the graph, a vertex cover must contain at least one of its endpoints. Note that the depth of the recursion is at most k, since at every step $|X|$ increases by one and it can never grow larger than k. This provides us an $O(2^k \cdot n)$ algorithm, proving again (Corollary 2.5) that vertex cover is in FPT.

Let us apply this algorithm on the kernel, we constructed in the previous section. There, $n = O(k^2)$, so we have an $O(n + m + 2^k \cdot k^2)$ algorithm. The first part comes from the linear preprocessing, and the second part from the search tree algorithm.

2.3 Iterative compression

The idea of iterative compression first appeared in [34]. We present the algorithm from [9]. Let's order the vertices of G arbitrarily: $v_1, ..., v_n$. Let

G_i be the induced graph on vertices $v_1, ..., v_i$. We are going to construct a vertex cover of size maximum k for $G_k, ..., G_n$ iteratively, or conclude that it does not exist. For G_k the set $X = \{v_1, ..., v_k\}$ is a vertex cover. In each step, we have $|X| \leq k$ for G_i and we want to construct a vertex cover X' with $|X'| \leq k$ for G_{i+1}, or conclude that it does not exist. First, observe that $Y = X + v_{i+1}$ is obviously a vertex cover for G_{i+1}. Now, we would like to compress it to a smaller vertex cover, if possible. Let us guess (in every possible way) a subset $Z \subseteq Y$ and try to find a vertex cover $|X'| \leq k$, for which $X' \cap Y = Z$. If there is an edge in $G[Y - Z]$, then this is impossible, since we can only add vertices from $V - Y$ to Z. Also, $N(Y - Z) \subseteq X'$ for the same reason.

Lemma 2.6 *$X' = Z \cup N(Y - Z)$ is a vertex cover if $G[Y - Z]$ has no edges.*

Proof. There are no edges in $G[V - Y]$, because Y was a vertex cover. There are no edges in $G[Y - Z]$ either. Every edge entering Y must come from $N(Y - Z)$, or it also enters Z. Thus every edge is covered. ■

If for some guess Z the set $|X'| = |Z \cup N(Y - Z)| \leq k$ and $G[Y - Z]$ is empty, we can return it as a vertex cover for G_{i+1}. If we do not find a vertex cover for any Z we conclude that it is a no-instance. We do this increasingly for every $k \leq i < n$, so at the end we know whether $G_n = G$ has a vertex cover of size at most k.

Theorem 2.7 *If we have a compression algorithm in time $f(k) \cdot n^c$, then we can solve the problem in time $f(k) \cdot n^{c+1}$.*

The running time of the compression is $O(2^{k+1} \cdot m)$, as it is enough to check each guess Z in linear time. We iterate this over n, so the total running time is $O(n \cdot 2^{k+1} \cdot m)$. If we use the previous kernel instead of the whole graph, we get a better time complexity: $O(n + m + k^2 \cdot 2^{k+1} \cdot k^3)$, because the kernel has maximum $O(k^3)$ edges.

2.4 Crown decomposition

The crown rule, which we will present in this section, was introduced by Chor, Fellows and Juedes in [11] and also shown in [26].

Definition 2.8 *A **crown decomposition** of a graph $G = (V, E)$ is a partition of V into three sets: $V = C \cup H \cup B$, such that C is independent, there are no edges between C and B and the bipartite graph between C and H has a matching covering H.*

The following lemma shows us the importance of a crown reduction in solving the vertex cover problem.

Lemma 2.9 *A graph G has a vertex cover of size at most k if and only if the graph $G - H - C$ has a vertex cover of size at most $(k - |H|)$.*

Proof. Suppose that G has a vertex cover X of size at most k. In order to cover $G[H \cup C]$, one needs at least $|H|$ vertices in X from $H \cup C$, because $G[H \cup C]$ has $|H|$ independent edges. To prove the opposite direction, if $G - H - C$ has a vertex cover X' of size at most $(k - |H|)$ then $X = X' \cup H$ is a vertex cover of G. ■

We would like to design an algorithm that either outputs that there is no vertex cover of size at most k, or finds a kernel of size at most $3k$.

Let us delete the isolated vertices of G, this produces an equivalent instance. Take an inclusion-wise maximal matching M in G. If $|M| > k$, we can output "NO", as we have more than k independent edges, which cannot be covered by at most k vertices. Let X be the set of end vertices of M and I the set of the vertices not covered by M. Note that I is independent, since M is maximal and also every vertex in I is connected to X, because we have no isolated vertices.

Take the bipartite graph G^* between X and I, and calculate the maximum matching M^* in G^*. Also find the minimum vertex cover T in G^*. Again, if $M^* > k$, we can return "NO".

If $T \cap X \neq \emptyset$, denote $H = T \cap X$, $C = I - T$ and $B = V - H - C$. By Kőnig's theorem, we know that $|T| = |M^*|$ and exactly one end vertex of each edge in M^* lies in T. We received a crown decomposition, because $|M^*| = |T|$, so M^* is covering H in the bipartite graph between C and H. By Lemma 2.9 we can continue our algorithm on $G[B]$ by setting $k' = k - |H|$ and finding a new crown decomposition of $G[B]$.

On the other hand if $T \cap X = \emptyset$, we get that $T = I$. We already know that $|T| = |M^*| \leq k$ and $|M| \leq k$. Since the set of end vertices of M was X, we also get that $|X| \leq 2k$. Together this means that $|V| = |X \cup I| \leq 3k$, so we arrived at a kernel of size at most $3k$. Applying the bounded search tree algorithm on this kernel results in an $O(k \cdot 2^k)$ algorithm.

The only remaining thing is to check the running time of the reduction rule. Finding an inclusion maximal matching takes $O(m)$ time. Finding the maximum matching of a bipartite graph can be done easily in $O(|M^*|m) \leq O(km)$ time. Since k is decreasing in every step, we can only have at most k reduction steps, so the reduction algorithm runs in $O(k^2m)$ algorithm for solving the kernel. These together result in the following theorem.

Theorem 2.10 *There is an $O(k^2m + k \cdot 2^k)$ algorithm for the vertex cover problem.*

The following theorem is a consequence of the theorem of Nemhauser and Trotter [31].

Theorem 2.11 (Nemhauser-Trotter) *There is an algorithm that builds a kernel of size at most $2k$ with complexity $O(m + k^4)$.*

Proof. Let us take $V' = \{v' \mid v \in V\}$ and form a bipartite graph $P = (V, V', \widehat{E})$ by connecting uv' if and only if uv was an edge in G. By applying the algorithm of Hopcroft and Karp, we can find a maximum matching M and a minimum vertex cover T in time $O(\sqrt{n} \cdot m)$. Kőnig's thoerem states that $|M| = |T|$. If $|T| > 2k$, we can stop and output "NO", since a matching M in P corresponds to a vertex-disjoint set of cycles and paths in G. The edges of these cycles and paths need at least $\frac{|T|}{2}$ vertices to be covered.

Let $H = \{v \in V \mid v \in T \wedge v' \in T\}$, $B_1 = T \cap V - H$, $B_2' = T \cap V' - H$ and $B = B_1 \cup B_2$, $C = V - H - B$. Let us observe that both end vertices of an edge in M cannot be in T, since T is a minimum vertex cover and M is a maximum matching and $|M| = |T|$. But M is covering T, so $C = \{v \in V \mid v \in V - T \wedge v' \in V' - T'\}$ is independent in G and there is no edge between C and B. For example, there cannot be a cb_2 edge, since $c'b_2$ would not be covered in P. As M is covering B_1 in P, the other end vertices must lie in B_1'. Similarly the other end vertices of the edges of M covering B_2 lie in B_2'. This implies that in P the other end vertices of the edges of M covering H' are in C. The edges of M covering H' are the matching edges from H to C in graph G.

As $|B| + 2|H| = |T| \leq 2k$, the number $|B|$ of remaining vertices after the crone reduction is not more than $2k$.

First, we use the kernel of size $k^2 + k$, we got in the first part of this chapter after applying the linear-time algorithm, we described there. Also, we know that the degree of each vertex is at most k, so the the number of edges in this kernel is at most $(k^3 + k^2)/2$. As $n = O(k^2)$ and $m = O(k^3)$, the algorithm of Hopcroft and Karp runs in time $O(k^4)$. This immediately gives us the crown decomposition. Altogether the algorithm of Nemhauser and Trotter applied to the kernel (built in $O(m)$ time) in the first part runs in time $O(k^4)$. ■

Chapter 3

Feedback vertex set

Let us turn to another famous problem and see some different techniques applied to that. For the feedback vertex set problem, the running time was improved over the years ([33], [24], [15], [9], [8]) resulting in the current best deterministic algorithm running in time $O^*((2+\phi)^k)\cdot n^{O(1)}$ in [28]. ($\phi < 1.619$ is the golden ratio.) If we allow randomization, then the problem admits an algorithm with running time $3^k \cdot n^{O(1)}$, see [14].

Definition 3.1 *Given a graph G, we say that $X \subseteq V$ is a **feedback vertex set** if $G - X$ contains no cycles.*

The problem here is for a given instance (G, k), where G is an undirected graph and k is the parameter, decide if there exists a feedback vertex set of size at most k. We also allow G to be a multigraph (multiple edges and also loop edges can be present in the graph).

First we would like to reduce the problem to a smaller instance. We observe that in case $v \in V$ has a loop edge, it must be contained in X. Also, if we have an edge of multiplicity more than 2, we can reduce it to 2. As vertices with degree one do not have to be included in X, we can just delete them. We always apply the first possible rule from the following set:

(i) Delete a vertex with loop edges and decrease k by 1.

(ii) Reduce edge multiplicities to a maximum of 2.

(iii) Delete a vertex of degree 1.

(iv) Delete a vertex of degree 2 and connect its two neighbors.

The last rule produces an equivalent instance, because if one included v of degree 2 in X, one could also include one of its neighbors instead, as a cycle has to go through them as well. It also works for double edges, since we get a loop edge and then we include the neighbor in X. Also if once k gets negative, we can return "NO". Observe that this reduction is indeed polynomial.

This way, we reached an equivalent instance, which has the following properties:

(i) G has no loops.

(ii) Every edge multiplicity is at most 2.

(iii) The minimum degree is 3.

3.1 Bounded search tree

Now we prove two lemmas, before describing the actual search tree algorithm from [36]. We would like to branch on vertices with large degrees.

Lemma 3.2 *For every graph G and feedback vertex set X*

$$\sum_{v \in X} (d(v) - 1) \geq m - n + 1,$$

where n is the number of vertices and m is the number of edges.

Proof. Since $F = G - X$ is a forest, it has at most $n - |X| - 1$ edges. Notice that every edge not in F has at least one of its end vertices in X. So by counting every edge

$$m \leq \sum_{v \in X} d(v) + (n - |X| - 1),$$

which ends our proof. ■

Lemma 3.3 *Let B contain the $3k$ vertices of V with largest degrees. Every feedback vertex set X of size at most k has to contain at least one vertex from B.*

Proof. Let us order the vertices of V is descending order of their degree: $(v_1, v_2, ..., v_n)$ for which $d(v_1) \geq d(v_2) \geq ... \geq d(v_n)$. Suppose that $B \cap X = \emptyset$, we get from our previous lemma that

$$\sum_{i=1}^{3k} (d(v_i) - 1) \geq 3 \cdot (\sum_{v \in X} (d(v) - 1)) \geq 3 \cdot (m - n + 1).$$

Also because of our assumption

$$\sum_{i>3k}(d(v_i)-1) \geq \sum_{v\in X}(d(v)-1) \geq m-n+1.$$

Together they give us

$$2m-n=\sum_{i=1}^{n}(d(v_i)-1) \geq 4\cdot(m-n+1).$$

The first equality holds, because $\sum_{i=1}^{n} d(v_i) = 2m$. This means $2m < 3n$, which is a contradiction with all degrees being at least 3. ■

Theorem 3.4 *There is an $O\big((3k)^k \cdot n^{O(1)}\big)$ algorithm for feedback vertex set.*

Proof. First, we reduce our instance by the rules above. If we get $k \geq 0$ and G is a forest, we can instantly return our X. Otherwise, we branch on the first $3k$ largest degree vertices. One of them must be in X by our previous lemma, so we choose one of them: v (we examine all possibilities) and do a recursion on $(G-v,\ k-1,\ X+v)$. If once we arrive at a forest, we found a solution. If our parameter gets under zero, we do not search any more on that path. As k decreases by one at each recursive call, we can see that the tree has depth at most k. If we keep track of the degrees of the vertices, at a function call, we have to find one of the $3k$ maximum degree vertices in $O(n)$ and modify its neighbor's degrees in $O(n)$. ■

3.2 Iterative compression

First, we are going to present some generic steps in designing iterative compression algorithms. Here we follow the definitions from section 2.3. During our iterative compression, we have an instance (G, Z, k), where Z is a feedback vertex set of size $k+1$ and we would like to produce a feedback vertex set in G of size k. Following the idea presented in section 2.3, we are going to guess a $X_Z \subseteq Z$ and we would like to find a feedback vertex set X in $G - X_Z$, such that $|X| \leq k - |X_Z|$ and $X \cap (Z - X_Z) = \emptyset$.

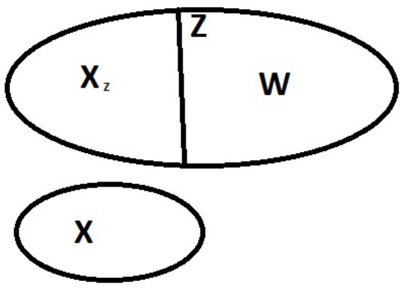

Figure 3.1: Graph G

In this step, we have $W = Z - X_Z$ of size $k + 1 - |X_Z|$, so our goal is for a given W to find a feedback vertex set in $G' = G - X_Z$ that is disjoint from W and of size $\ell = |W| - 1 = k - |X_Z|$. This is the **disjoint feedback vertex set** problem.

Theorem 3.5 *If we can solve the disjoint feedback vertex set with parameter ℓ in time $\alpha^\ell \cdot n^c$, then we can solve the compression problem in time $(k+1) \cdot (\alpha+1)^k \cdot n^c$.*

Proof. By looping through each guess X_Z of size i, we can add up the running time of our algorithm:

$$\sum_{i=0}^{k} \binom{k+1}{i} \cdot (\alpha^{k-i} \cdot n^c) \leq (k+1) \cdot (\alpha+1)^k \cdot n^c,$$

which completes our proof. ■

From now on, we denote by G' the graph $G - X_Z$, as discussed above, we need to examine this graph. Note that W is a feedback vertex set. The only thing remaining is to actually solve the disjoint feedback vertex set problem for an instance (G', W) that is find a feedback vertex set X disjoint from W. Let $\ell = |W| - 1$, so we would like a vertex set X for which $|X| \leq \ell$. This is a recursive solution, so formally, we write (G', W, ℓ, X), where X is the empty set at the beginning and a solution if the program exits with "YES". As usual, first we make some observations about the structure of the graph, while reducing it to a smaller equivalent instance. If $G'[W]$ is not a forest, return "NO", since we cannot add any vertices from W to the feedback vertex set. Let $H = G' - W$ and apply the following reduction rules:

(i) Delete vertices in G' of degree 1.

(ii) If there is a vertex $v \in V(H)$, such that $G'[W+v]$ contains a cycle, proceed with $(G'-v, W, \ell-1, X+v)$, as v has to be included in the solution.

(iii) If there is a vertex $v \in V(H)$ of degree 2, such that at least one neighbor of v is from $V(H)$, then delete this vertex and connect its neighbors (the graph can become a multigraph).

We can easily see that these reduction rules produce an equivalent instance, indeed. After applying the rules above as long as possible, if $l < 0$ return "NO". If X is a feedback vertex set, return it. Let us suppose that none of these are applicable. Also, we know that H is a forest, because W is a feedback vertex set in G'. Take a leaf q of H. We will branch on q being in X or not. Recursively, solve the problem for $(G'-q, W, \ell-1, X+q)$ and for $(G, W+q, \ell, X)$. If one of them has a solution, return it as the answer. $G'[W+q]$ is a forest, as we could not apply rule (ii). This algorithm is correct, because we always got an equivalent instance by applying reduction rules.

Observe that by the choice of q, it has at least two neighbors in W, because it cannot have 0 by (i) or 1 by (iii). Those neighbors have to be in different connected components of $G[W]$, by (ii). Let $\mu(I) = l + \text{comp}(I)$ be a measure for an instance I, where $\text{comp}(I)$ is the number of connected components of $G[W]$. At the recursion, when we include q in X, l decreases by one, whereas the number of components stay the same. If we do not include q in X, then l stays the same but the number of components strictly decreases, since the neighbors of q in W were in different components. So, by every recursion $\mu(I)$ decreases. At the beginning $\mu(I) \leq k + |W| \leq k + (k+1)$, so we have a solution for the disjoint version of the problem in time $O(2^{2k+1} \cdot n^2) = O(4^k \cdot n^2)$. So with our previous theorem, we have a reduction algorithm in time $O(5^k \cdot n^2)$. Combining it with Theorem 2.7 we arrived at the following theorem.

Theorem 3.6 *Feedback vertex set has a solution in time $O(5^k \cdot n^3)$.*

3.3 Randomized algorithm for feedback vertex set

In this section, we are going to present a randomized algorithm for the feedback vertex set defined above. We are going to deal with multigraphs, again. First apply the reduction from the beginning of this chapter, so every vertex is of degree 3 at least. The key observation is the following lemma:

Lemma 3.7 *Let G be a multigraph with minimum degree at least 3 and X be a feedback vertex set. Then at least half of the edges in G are adjacent to X.*

Proof. Again, let $H = G - X$ and observe that H is a forest, because X is a feedback vertex set. So, $|E(H)| < |V(H)|$. We would like to prove that $|E(G)| - |E(H)| > |E(H)|$, since every edge not in H is by definition adjacent to X.

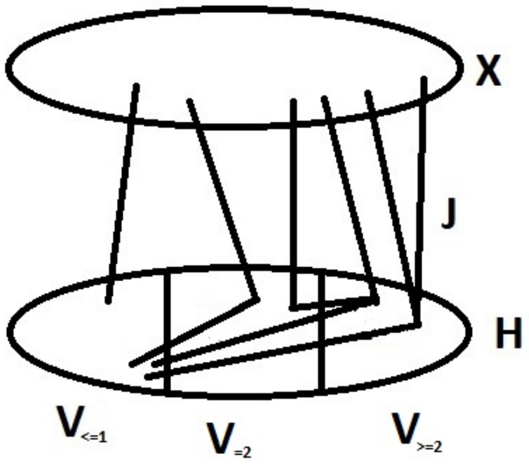

Figure 3.2: Graoh G

With our previous observation, it is enough to see that $|E(G)| - |E(H)| \geq |V(H)|$. Let J denote the edges connecting X and H. Furthermore let $V_{\leq 1}$, $V_{=2}$, $V_{\geq 3}$ be the vertex sets of H with degree (in H) as indicated in the index. As the minimum degree in G is at least 3, every vertex in $V_{\leq 1}$ contributes at least 2, every vertex in $V_{=2}$ contributes at least 1 edge to J. Since H is a forest, we get that $|V_{\leq 1}| > |V_{\geq 3}|$. So we obtain the inequality

$$|E(G)| - |E(H)| \geq |J| \geq 2 \cdot |V_{\leq 1}| + |V_{=2}| > |V_{\leq 1}| + |V_{=2}| + |V_{\geq 3}| = |V(H)|,$$

which finishes our proof. ■

Theorem 3.8 *There exists a polynomial time randomized algorithm for the feedback vertex set problem that for a yes-instance as an input, returns a feedback vertex set of size at most k with probability at least 4^{-k} and for a no-instance, it always returns "NO".*

Proof. Apply the reductions seen at the beginning of this chapter, so we get (G', k'). If $k' < 0$ return "NO". Otherwise, G' has minimum degree at least 3. We already have some vertices, we got from the reduction that are surely inside of any solution X. Denote these by X_0. We only need to find a solution X' for G' and return $X_0 + X'$. If G' is a forest, we can instantly return $X = X_0$. Lemma 3.7 tells us that if we choose an edge in G uniformly randomly, then with probability at least $\frac{1}{2}$, one of its endpoints will be in X. So if we choose one of its endpoints q uniformly randomly, it will be in X with probability at least $\frac{1}{4}$. Now we do a recursion on $(G' - q, k' - 1)$. If this returns a solution X', we return $X_0 + X' + q$, otherwise we return that we have not found a solution. We can easily verify that $X = X_0 + X' + q$ will be indeed a solution, since X' is a feedback vertex set of $G' - q$ of size at most $k' - 1$ and we also included q in the solution. If there exists a feedback vertex set of G of size at most k, then because of the reduction rules there must be a feedback vertex set of G' of size at most k'. As we do the recursion at most k' times and $k' \leq k$, we arrive that the probability that we find a solution is at least 4^{-k}. ■

Theorem 3.9 *There exists a randomized algorithm for the feedback vertex set problem that runs in time* $4^k \cdot n^{O(1)}$ *and returns "NO" if there is no feedback vertex set of size at most* k*. If there is a solution, it returns a solution with constant probability.*

Proof. We have to run the previous algorithm 4^k times to achieve a constant error probability. ■

Chapter 4

Color coding

The technique of color coding was introduced by Alon, Yuster and Zwick to handle the problem of detecting a small subgraph in a large input graph. If the input graph is a forest, or more generally a graph of constant tree-width, then we can achieve an FPT algorithm of running time $2^{O(k)}n^{O(1)}$.

4.1 Exact path

We are going to show the color coding technique on the exact path problem, which given an instance (G, k) asks, if there is a path of length k in G. The **length** of the path is the number of edges, it consists of. The main idea of color coding is to color all the vertices of G uniformly randomly with k colors. Then, we are only looking for a **colorful** path, meaning a path, that has each of the k colors amongst its vertices. First we prove a lemma, which is essential in dealing with this problem.

Lemma 4.1 *Let $X \subseteq V(G)$ be of size k. Let us color all the vertices in $V(G)$ uniformly randomly with k colors. The probability that all the vertices in X got pairwise distinct colors is at least e^{-k}.*

Proof. There are k^n colorings of $V(G)$ and $k! \cdot k^{n-k}$ colorings for which X is colorful. Since $k! > (\frac{k}{e})^k$, we proved the lemma. ■

Now we can present the algorithm of finding a colorful path.

Theorem 4.2 *There is an $O(2^k \cdot n^{O(1)})$ algorithm for finding a colorful path in a graph G, whose vertices are colored with k colors.*

Proof. Let us denote the colors with numbers $1, ..., k$ and let $col(q)$ be the color of vertex q. We use dynamic programming. For $C \subseteq \{1, ..., k\}$ let $D[C, q]$ be a boolean value, which is the answer for the following question: does there exist a path of length $|C|$ with pairwise distinct colors as indicated in C, whose endpoint is q? For $C = \{c\}$ this value is true, if and only if $\mathrm{col}(q) = c$. We can apply the following recursion, if $|C| > 1$:

$$D[C, q] = \bigvee_{r \in N(q)} \big(\{D[C - \mathrm{col}(q), r]\} \wedge \{\mathrm{col}(r) \in C - \mathrm{col}(q)\}\big).$$

We look for all neighbors r of q, which could be the last but one vertex of this path. If $D[C, q]$ is true, there had to be a correct last but one vertex for which the appropriate $D[C - col(q), r]$ is true. At the end, we return true, if and only if $D[\{1, ..., k\}, v]$ is true for any $v \in V$. We only calculate every $D[C, q]$ once, so there are $2^k \cdot n$ subproblems to calculate. One state can be calculated in $O(n)$ time, so the overall time complexity is $2^k \cdot n^2$. One can observe that we can calculate the D's for every C, by looping through every edge once as indicated in the formula. This gives us a better $2^k \cdot m$ time complexity. Also we can return the colorful path, if we memorize the last vertex for each true D. Beginning for $|C| = 1$: $\mathrm{Last}[C, q] = \emptyset$, we can also see that

$$\mathrm{Last}[C, q] = r, \text{ if}\big(\{D[C - col(q), r]\} \wedge \{\mathrm{col}(r) \in C - \mathrm{col}(q)\}\big).$$

So we always know the last vertex in the path. This way we can output the whole path in $O(k)$ time complexity. ■

Theorem 4.3 *There is a randomized algorithm for the exact path problem that, given az instance (G, k), if it is a no-instance, returns "NO", otherwise it returns a path of length k with constant probability. This algorithm runs in time $(2e)^k \cdot m$.*

Proof. We will run the previous algorithm e^k times, re-coloring all the vertices before each run uniformly randomly and independently. By Lemma 4.1, if it was a yes-instance, at each run, we return a path of length k with probability at least e^{-k}. Repeating this k times gives us a constant error probability. ■

Chapter 5

Tree-width

In this chapter, we are going to give an overview of FPT problems in relation with tree-width (tw(G)). We refer to [13], where you can find all the necessary definitions and the proofs of the following theorems. From now on, we suppose that the reader is familiar with the basics of tree-decomposition.

5.1 Results with dynamic programming

We are going to present the most important results.

Definition 5.1 *The* ***weighted independent set*** *problem in a graph G is to find a maximum weight of an independent set in G. An independent set is a set of vertices that are pairwise non-adjacent.*

Let G be an undirected graph on n vertices and $K \subseteq V(G)$ be a set of terminals. A ***steiner tree*** *for K in G is a connected subgraph H of G containing K. In the* ***weighted steiner tree*** *problem, we are given an undirected graph G, a weight function $w : E(G) \to R_{>0}$ and the goal is to find a steiner tree H, whose weight is minimized.*

The ***maxcut*** *problem asks for a partition of $V(G)$ into sets A and B such that the number of edges between A and B is maximized. The* ***q-coloring*** *problem asks whether G can be properly colored using q colors, while in* ***chromatic number*** *the question is to find the minimum possible number of colors needed to properly color G.* ***Exact cycle*** *asks for existence of a cycle on at least l vertices in G, for a given integer l. In* ***cycle packing*** *the question is whether one can find l vertex-disjoint cycles in G.* ***Dominating set*** *problem is to find a vertex set X of size at most l, for which $N(X) = V(G) - X$. In the* ***odd cycle transversal*** *problem, the objective is to find at most l vertices whose removal makes the resulting graph bipartite. Problems* ***connected*** *vertex cover, connected dominating set, connected feedback vertex set* *differ from their standard (non-connected) variants by additionally requiring that the solution induces a connected graph in G.*

Theorem 5.2 *Let G be an n-vertex graph with weights on vertices given together with its tree decomposition of width at most k. Then in G one can solve*

(i) ***vertex cover*** *in time $2^k \cdot k^{O(1)} \cdot n$,*

(ii) ***dominating set*** *in time $3^k \cdot k^{O(1)} \cdot n$,*

(iii) ***odd cycle transversal*** *in time $3^k \cdot k^{O(1)} \cdot n$,*

(iv) ***maxcut*** *in time $2^k \cdot k^{O(1)} \cdot n$,*

*(v) q-**coloring** in time $q^k \cdot k^{O(1)} \cdot n$.*

Also one can solve the problems: steiner tree, feedback vertex set, hamiltonian path and exact path, hamiltonian cycle and exact cycle, chromatic number, cycle packing, connected vertex cover, connected dominating set, connected feedback vertex set in time $k^{O(k)} \cdot n$.

These solutions come from dynamic programming on the given tree decomposition. For these algorithms, we need to be given a tree of width at most k. Therefore, we need an algorithm that provides us a tree-decomposition.

Theorem 5.3 *[4] There exists an algorithm that, given an n-vertex graph G and integer k, runs in time $k^{O(k^3)} \cdot n$ and either constructs a tree decomposition of G of width at most k, or concludes that $\mathrm{tw}(G) > k$.*

5.2 MSO$_2$ logic and Courcelle's theorem

Courcelle's theorem provides us with FPT algorithms for certain problems on a graph with given tree-width. Monadic second order logic (MSO_2) is a formal language of expressing properties of graphs and objects inside these graphs.

Definition 5.4 *Formulas of MSO_2 can use four types of variables: for single vertices, single edges, subsets of vertices, and subsets of edges. Every formula ϕ of MSO_2 can have free variables, which often will be written in parentheses besides the formula. The formula together with the free variables is called the signature: S. The evaluation of the formula on a given graph can result in a true/false value. Formulas of MSO_2 are constructed inductively from smaller subformulas.*

(i) *If $u \in S$ is a vertex (edge) variable and $X \in S$ is a vertex (edge) set variable, then we can write formula $u \in X$.*

(ii) *If $u \in S$ is a vertex variable and $e \in S$ is an edge variable, then we can write formula $inc(u, e)$, which denotes incidence.*

(iii) *For any two variables $x, y \in S$ of the same type, we can write formula $x = y$.*

(iv) *Suppose that ϕ_1, ϕ_2 are two formulas over the same signature S. Then one can write $\neg\phi_1$; $\phi_1 \vee \phi_2$; $\phi_1 \wedge \phi_2$; $\phi_1 \Rightarrow \phi_2$.*

(v) *One can also use quantifiers: $\forall_{v \in X}\phi$; $\exists_{v \in X}\phi$.*

Let us see an example.

Example 5.5

$$\begin{aligned} 3-\text{colorability} &= \exists_{X_1,X_2,X_3 \subseteq V}\text{partition}(X_1,X_2,X_3) \\ &\quad \wedge \text{indp}(X_1) \wedge \text{indp}(X_2) \wedge \text{indp}(X_3) \end{aligned}$$

Here, *partition* verifies that X_1, X_2, X_3 partition V and *indp* verifies that X_i is an independent set.

$$\begin{aligned} \text{partition}(X_1,X_2,X_3) &= \forall_{v \in V}\Big\{(v \in X_1 \wedge v \notin X_2 \wedge v \notin X_3) \\ &\quad \vee (v \notin X_1 \wedge v \in X_2 \wedge v \notin X_3) \\ &\quad \vee (v \notin X_1 \wedge v \notin X_2 \wedge v \in X_3)\Big\} \\ \text{indp}(X) &= \forall_{x,y \in X}\neg\text{adj}(x,y) \end{aligned}$$

Theorem 5.6 ***(Courcelle's theorem [12])*** *Assume that ϕ is a formula of MSO_2 and G is an n-vertex graph with an evaluation of all the free variables of ϕ. Suppose, moreover, that a tree decomposition of G of width t is provided. Then there exists an algorithm that verifies whether ϕ is satisfied in G in time $f(||\phi||, t) \cdot n$, for some computable function f. Here, $||\phi||$ denotes the length of the encoding of ϕ in a string.*

With this theorem, we can easily prove that 3-colorability is in FPT, since we have encoded it in constant size in example 5.5. Courcelle's theorem gives an $f(c, t) \cdot n$ time complexity algorithm, which is exactly an FPT algorithm with parameter t. The natural encoding of the vertex cover problem quantifies each vertex of the cover, which is an $O(k)$ encoding, where k is the size of the vertex cover. This results only in an $f(k, t) \cdot n$ algorithm, which is by far weaker, than what we proved earlier. Therefore, we present a more general version of Theorem 5.6.

Theorem 5.7 *[2] Let ϕ be an MSO_2 formula with p free monadic variables $X_1, X_2, ..., X_p$, and let $\alpha(x_1, x_2, ..., x_p)$ be an affine function. Assume that we are given an n-vertex graph G together with its tree decomposition of width t, and suppose G is equipped with evaluation of all the free variables of ϕ apart from $X_1, X_2, ..., X_p$. Then there exists an algorithm that in $f(||\phi||, t) \cdot n$ finds the minimum and maximum value of $\alpha(|X_1|, |X_2|, ..., |X_p|)$ for which $\phi(X_1, X_2, ..., X_p)$ is true, where f is some computable function.*

Now, using this theorem, we can prove again that vertex cover admits an FPT algorithm.

$$\text{vcover}(X) = \forall_{e \in E} \exists_{x \in X} \text{inc}(x, e).$$

Let $\alpha(|X|) = |X|$ be our affine function. By Theorem 5.7, we can find the minimum cardinality vertex cover in time $f(t) \cdot n$, since $||\phi||$ is constant.

Chapter 6

Lower bound for kernelization

The so far best known technique to prove that under certain assumptions there is no polynomial kernel for a problem is via compositionality. Here, we will see all the definitions and also the sketch of the proof of the framework, Bodlaender, Jansen, and Kratsch developped in [5]. At the end of this chapter we see some application. For an introduction into complexity theory, we refer to [3].

6.1 Distillation

We first aim to formally capture the intuition that some information has to be lost when packing too many instances into a too small space in the kernel.

Definition 6.1 *Let $L, R \subseteq \sum^*$ be two languages. An **OR-distillation** of L into R is an algorithm that, given a sequence of strings $x_1, x_2, ..., x_t \subseteq \sum^*$, runs in time polynomial in $\sum_{i=1}^{t} |x_i|$ and outputs one string $y \subseteq \sum^*$ such that*

(i) $|y| = p(\max_{i=1}^{t} |x_i|)$ *for some polynomial p.*

(ii) $y \in R$ *if and only if there exists at least one index i such that $x_i \in L$.*

The answer to the output instance of R is equivalent to the logical OR of the answers to the input instances of L. We give the definition of coNP/poly:

Definition 6.2 *A language L belongs to the complexity class **coNP/poly** if there is a Turing machine M and a sequence of strings α_n, $n = 0, 1, 2, ...$ called the advice, such that if M is given the input x of length n, it decides whether $x \in L$ in co-nondeterministic polynomial time while using α_n. We also require $|\alpha_n| \leq p(n)$ for some polynomial p.*

This is an important theorem for which we omit the proof.

Theorem 6.3 *[23] Let $L, R \in \sum^*$ be two languages. If there exists an OR-distillation of L into R, then $L \in$ coNP/poly.*

This has an immediate corollary:

Theorem 6.4 *If an NP-hard language $L \subseteq \sum^*$ admits an OR-distillation into some language $R \in \sum^*$, then NP $\in$ coNP/poly.*

Proof. This is an immediate consequence of L being NP-hard and $L \in$ coNP/poly. ■

The assumption that NP $\not\subseteq$ coNP/poly may be viewed as a stronger variant of the statement that NP $\neq$ co-NP. Also, we know that from NP $\subseteq$ coNP/poly, $\sum_3^P =$ PH follows, that means the polynomial hierarchy collapses to its third level as described in [37].

6.2 Composition

Now, we give the essential definitions of composing instances. The cross-composition technique was developed in [5].

Definition 6.5 *An equivalence relation $\mathcal{R}$ on the set $\sum^*$ is called a **polynomial equivalence relation**, if one can check in polynomial time, whether two elements are equivalent. Also $\mathcal{R}$ must have at most $p(n)$ equivalence classes restricted to $\sum^{\leq n}$ for a polynomial p.*

Definition 6.6 *Let $L \subseteq \sum^*$ be a language and $Q \subseteq \sum^* \times N$ be a parameterized language. We say that L **cross-composes** into Q if there exists a polynomial equivalence relation $\mathcal{R}$ and an algorithm $\mathcal{A}$, called the cross-composition, satisfying the following conditions. The algorithm $\mathcal{A}$ takes as input a sequence of strings $x_1, x_2, ..., x_t \subseteq \sum^*$ that are equivalent with respect to $\mathcal{R}$, runs in time polynomial in $\sum_{i=1}^t |x_i|$, and outputs one instance $(y, k) \subseteq \sum^* \times N$ such that:*

(i) for some polynomial p: $k \leq p(\max_{i=1}^t |x_i| + \log t)$.

(ii) $(y, k) \in Q$ if and only if there exists at least one index i such that $x_i \in L$.

Note that here, contrary to the OR-distillation, only the parameter has to be small, while y can be large (but $\leq \sum |x_i|$). Let's see an example of this.

Definition 6.7 *The hamiltonian path problem asks whether there exists a simple path P in the input graph G such that $V(P) = V(G)$.*

Example 6.8 The hamiltonian path problem (which is NP-hard) cross-composes into the exact path problem parameterized by the path length. We call the instances that do not describe any graph **malformed** instances. $\mathcal{R}$ can be defined as follows: put all malformed instances into one class and all well-formed instances should be partitioned with respect to the number of vertices in the graph. If $\mathcal{A}$ is given some malformed instances, it returns "NO", otherwise, given a sequence of well-formed n-vertex graphs, it returns their disjoint union with parameter $k = n$. If the disjoint union has an n-vertex path, then at least one of the input graphs has to contain a hamiltonian path.

Definition 6.9 *A* ***polynomial compression*** *of a parameterized language $Q \subseteq \sum^* \times N$ into a language $R \subseteq \sum^*$ is an algorithm that takes as input an instance $(x, k) \in \sum^* \times N$, works in time polynomial in $|x| + k$, and returns a string y such that:*

(i) for some polynomial p: $|y| \leq p(k)$.

(ii) $y \in R$ if and only if $(x, k) \in Q$.

Obviously, a polynomial kernel is also a polynomial compression by treating the output kernel as an instance of the unparameterized version of Q. We get the unparameterized version of a parameterized language by writing the parameter at the end in unary format. The main difference between polynomial compression and polynomial kernelization is that the polynomial compression is allowed to output an instance of any language R, even an undecidable one. If R is reducible in polynomial time back to Q, then by first applying the compression and then the reduction, we get a polynomial kernel for Q. But generally R can have much higher complexity than Q. Now we will see the main theorem, which we are going to apply on some examples at the end of the section.

Theorem 6.10 *Let us assume that* NP $\not\subseteq$ coNP/poly. If an NP-hard language L cross-composes into a parameterized language Q, then Q does not admit a polynomial compression.

Proof. Let $\mathcal{A}$ be a cross-composition of L into Q and let $\mathcal{R}$ be the polynomial equivalence relation used by $\mathcal{A}$. Let us assume in contrary with the theorem that Q admits a polynomial compression $\mathcal{C}$ into some language R. Define $\mathrm{OR}(R)$ as the language of concatenating strings d_i with some special separator character. We require that for at least one $d_i \in R$. We are going to construct an OR-distillation of L into $\mathrm{OR}(R)$, which, by Theorem 6.4 implies that $\mathrm{NP} \subseteq \mathrm{coNP/poly}$.

This construction is as follows. As input, we get strings x_i. Let $n = \max|x_i|$. First we throw out the duplicates, so we result in at most $|\sum|^{n+1}$ strings of size at most n. That means $t = O(n)$, where t is the number of strings. Then by using the polynomial equivalence relation $\mathcal{R}$, we partition the strings into equivalence classes C_i. The number of these classes is at most $p_1(n)$ for a polynomial p_1. We cross-compose these classes separately with the cross-composition algorithm $\mathcal{A}$, so we obtain instances (c_i, k_i). If any of them is a yes-instance, then at least one $x_i \in L$. Also, we can check easily that $k_j \leq p_0(\max_{x \in C_j}|x| + \log|C_j|)$ for a polynomial p_0. With our previous observation $t = O(n)$, we get that $k_j \leq p_1(n)$ for some polynomial p_1. Now, we apply the assumed polynomial compression $\mathcal{C}$ to each instance. So we get some strings d_i for which $d_i \in R$ if and only if $(c_i, k_i) \in Q$. We also get that $|d_i| \leq p_2(n)$, where p_2 is a polynomial. We only need to concatenate our d_is with some special separator character to obtain a d string. It follows that $d \in \mathrm{OR}(R)$ if and only if for at least one $x_i \in L$, which by Theorem 6.4 is a contradiction to the $\mathrm{NP} \not\subseteq \mathrm{coNP/poly}$ assumption. ■

6.3 Examples

We have already constructed a cross-composition algorithm of the hamiltonian path problem into the exact path problem at 6.8. Together with the NP-hardness of the hamiltonian path problem, we can use Theorem 6.10 to prove the following theorem.

Theorem 6.11 *Exact path does not admit a polynomial kernel unless* $\mathrm{NP} \subseteq \mathrm{coNP/poly}$.

To be precise, exact path does not admit a polynomial compression, which is in fact stronger. It is interesting that this method, which is by far the most famous, cannot tell the difference between polynomial kernelization and polynomial compression.

So far, we have been using the OR function in our theorems. One could ask, if we can replace it by AND.

Theorem 6.12 *[19], [16] Let $L, R \subseteq \sum^*$ be two languages. If there exists an AND-distillation of L into R, then $L \subseteq$ coNP/poly.*

The proof of this theorem is essentially the same as Theorem 6.4. Also our main theorem will be true.

Theorem 6.13 *Let us assume that* NP $\nsubseteq$ coNP/poly. If an NP-hard language L AND-cross-composes into a parameterized language Q, then Q does not admit a polynomial compression.

However the proof is completely different from the proof of Theorem 6.10. We will see an example of the AND-cross-composition.

Example 6.14 Give a graph G and a parameter k, we would like to verify whether $\mathrm{tw(G)} \leq \mathrm{k}$. This is called the **tree-width** problem. Observe that the tree-width of disjoint graphs is the maximum tree-widths of the graphs, we can say that $\mathrm{tw(\cup G_i)} \leq \mathrm{k}$ if and only if $\mathrm{tw(G_i)} \leq \mathrm{k}$ for all i. This is an AND-cross composition from tree-width into parameterized tree-width. It is known that computing the tree-width is NP-hard, so putting this together by Theorem 6.13 we can conclude that:

Theorem 6.15 *Tree-width does not admit a polynomial kernel unless* NP $\subseteq$ coNP/poly.

Remark 6.16 *For more examples, we refer to [25], where you can also read a similar way of describing the whole technique presented in this chapter. For a more generalized description, see [5].*

Chapter 7

Shortest paths

The disjoint shortest paths problem is defined as follows. Given a graph G and k pairs of distinct vertices $(s_i, t_i), 1 \leq i \leq k$, find whether there exist k pairwise disjoint shortest paths P_i, between s_i and t_i, for all $1 \leq i \leq k$. We may consider directed or undirected graphs and the paths may be vertex or edge disjoint. Eilam-Tzoreff in [20] showed that these four problems are NP-complete when k is part of the input even for planar graphs with unit edge-lengths. Also the problem: given a graph and two distinct pairs of vertices, find whether there exist two disjoint paths P_1, P_2 between them such that P_1 is a shortest path is shown to be NP-complete for undirected graphs with unit edge-lengths in the same article.

In the min-sum k edge-disjoint paths problem the input is an undirected graph G and ordered pairs $(s_i, t_i), 1 \leq i \leq k$ and the goal is to find a path between s_i and t_i so that these paths are edge-disjoint and the sum of their lengths is minimum. For every fixed $k \geq 2$, the question of NP-hardness for the min-sum k edge-disjoint paths problem has been open for more than two decades. Fenner, Lachish and Popa gave an PTAS for this problem for $k = 2$ in [21].

In the min-max 2 disjoint paths problem the input is a graph G and ordered pairs $(s_i, t_i), 1 \leq i \leq k$ and the goal is to find vertex-disjoint paths between s_i and t_i. The goal is to minimize the length of the longest of these k paths. This is known to be NP-hard for general graphs. It is also known that the problem is weakly NP-hard for graphs with tree-width 3. Kobayashi and Sommer presented an algorithm that solves the problem for graphs with tree-width 2 in polynomial time in [27] .

The theorems presented in this chapter are the author's own results.

Definition 7.1 *Given a graph G with a special vertex s and t, we call k paths starting from s a **k-fan** if the paths are vertex-disjoint except for s and*

t and neither of the paths contains t as a degree 2 vertex.

We will give special attention of special k-fans, for example, where each of the k paths ends in the same vertex t.

Definition 7.2 *A k-fan, where all paths end in t is called a **t-fixed k-fan**.*

Let us define a length function to k-fans. This will be used to state a couple of minimization problems.

Definition 7.3 *For a k-fan F the* maxlength(F) *is the length of the longest of the k paths in F.*

7.1 Minimizing in unweighted DAGs

We are going to present a dynamic programming approach for the following theorem.

Theorem 7.4 *An unweighted directed acyclic graph G with start vertex s, end vertex t and a parameter k is given. There is a solution for finding the minimal* maxlength *of t-fixed k-fans in time $O(n^{2k+1})$.*

Proof. Let h be a vector of lenght k of pairs of vertices and distances, h_i is a pair of a vertex and a distance associated to it: (v, dist) and $\text{vertex}(h_i) = v$, $\text{dist}(h_i) = v$. Let $\text{Fans}(h)$ be the set of k-fans with starting point s, such that $F \in \text{Fans}(h)$, if and only if for each $0 \leq i < k$ the i-th endpoint of F paired with the length of the i-th path is h_i. Let $D[h]$ be true, if and only if there exists a vertex disjoint k-fan of the set $\text{Fans}(h)$. We would like to calculate $D[h]$ for certain hs. Observe that $D[(s,0),(s,0),...,(s,0)] = \text{true}$. The solution for the problem can be calculated easily, if we know all Ds, because we look at $D[(t,\text{dist}_0),(t,\text{dist}_1),...,(t,\text{dist}_{k-1})]$ for all vectors dist, where for each $i \leq 0 < k$, $\text{dist}_i < n$ (as each path length is shorter than n). We would like to find the minimum $||\text{dist}||$ vector, for which the appropriate D is true, where $||.||$ denotes the max norm.

As G is directed and acyclic, the vertices have a topological ordering, where edges are going from left to right. Let $j < k$ be an index for which $\text{vertex}(h_i)$ is the rightmost in the topological ordering.

Let us calculate $D[h]$ now. The algorithm guesses all the last but one vertices on path j. It loops through all q vertices for which $\big(q, \text{vertex}(h_k)\big)$ is a directed edge and $q \neq \text{vertex}(h_i)$ for all i.

$$D[h] = \bigvee_q \Big\{ D[h_0, ..., h_{j-1}, \big(q, \text{dist}(h_j) - 1\big), h_{j+1}, ..., h_{k-1}] \Big\}.$$

First, we can see that if $D[h]$ is true, then there must exist some last but one vertex q, for which the appropriate D is true. Also, have to prove that the paths if for some q the appropriate D is true, then we can find a k-fan for proving that $D[h]$ is true. We take F' proving the trueness of the previous D. Adding vertex(h_j) at the end of the j-th path, we will indeed get vertex disjoint paths, since the j-th path ended in the rightmost vertex in the topological ordering. It could only intersect with previous vertices, but F' was also non-intersecting. These two observations together show the correctness of this dynamic programming solution.

Our algorithm finds the minimizing k-fan for the problem, if for each true D, we store the last but one vertex on the path ending in the rightmost vertex of the topological ordering.

We turn to calculate the overall running time of the algorithm. We have $n^k \cdot n^k$ sets for which we calculate the D value, as each h_i is a pair of a vertex and a distance less than n. We recurse at most on n vertices, that means, the running time is $O(n^{2k+1})$. ■

Proposition 7.5 *Immediately follows from definition 1.3 that this problem is in* XP.

7.2 Minimizing in unweighted graphs

Let us fix k as a global constant and let us introduce l, the length of the longest path as a new parameter. We are also given an undirected or a directed, unweighted graph G. The problem is to minimize the maxlength of t-fixed k-fans, where all the paths are inner-vertex disjoint and all the path lengths are at most l. We are going to find an FPT algorithm for this case (k is a global constant).

Theorem 7.6 *There is a randomized algorithm for minimizing the* maxlength *of t-fixed k-fans, where all the paths are inner-vertex disjoint and all the path lengths are at most l, in time $O(e^{kl} \cdot 2^{k \cdot l} \cdot n^{k+1} \cdot l^k)$. It returns the minimum* maxlength *with constant probability and returns a bigger value otherwise.*

Proof. Start by coloring all vertices of G with randomly with $k \cdot l$ colors. The probability that for an optimal solution all vertices of the k paths are distinctly colored, as proven in Lemma 4.1, is at least $e^{-k \cdot l}$, since the union of these k paths have at most $k \cdot l$ vertices. We now proceed to solving the problem only for colorful fans.

Let $C \subseteq \{1, ..., k \cdot l\}$ be the colors already used in a solution. Also let e_i be the end vertices and l_i be the lengths of the k-fan respectively. Let

$D[C, e_1, l_1, ..., e_k, l_k]$ be true, if and only if there is a k-fan ending in vertices e_i, having length l_i and using every color in C exactly once. Let us suppose that l_k is the largest amongst all l_is.

$$D[C, e_1, l_1, ..., e_k, l_k] = \bigvee \Big(\{D[C - \mathrm{col}(e_k), e_1, l_1, ...r, l_k - 1]\} \\ \wedge \{col(r) \in C - \mathrm{col}(e_k)\} \wedge \{e_k r \in E(G)\}\Big).$$

Here, we did the recursion on the last but one vertex on the kth path in the fan. Since we have $2^{k \cdot l} \cdot n^k \cdot l^k$ subproblems ($2^{k \cdot l}$ sets for C, n^k vertex-sets and l^k lengths), each of which can be calculated in time $O(n)$ by the above formula, we have a computational upper bound of $O(2^{k \cdot l} \cdot n^{k+1} \cdot l^k)$. Repeating this algorithm e^{kl} times by recoloring every vertex uniformly randomly and independently gives us a constant error probability. ■

Let us introduce another new parameter: the tree-width. We are going to show an FPT algorithm for this problem in undirected graph with parameters: k, l and w, where w denotes the tree-width. Note that the above algorithm does not give us an FPT algorithm, even if we take k and l as parameters. In Chapter 5.2 we have seen Courcelle's theorem. With the use of that, we are going to show the existence of an FPT algorithm with parameters: k, l and w. In a graph G with maximal tree-width w the value $\mathrm{MaxFan}(X_1, ..., X_k)$ is true, if and only if there exists a k-fan with paths X_i with maxlength not more than l.

$$\begin{aligned} \mathrm{MaxFan}(X_1, ..., X_k) \;&=\; \big(\wedge_{i=1}^{k} \mathrm{isPath}(X_i)\big) \wedge \big(\wedge_{i=1}^{k} |X_i| \le l\big) \\ &\quad \wedge \big(\mathrm{isDisjoint}(X_1, ..., X_k)\big). \end{aligned}$$

Here $\mathrm{isPath}(X)$ decides if X_i is a path starting from s and ending at t:

$$\begin{aligned} \mathrm{isPath}(X) \;&=\; \mathrm{connected}(X) \wedge \big(s, t \in X\big) \wedge \big(\deg(s) = 1\big) \\ &\quad \wedge \big(\deg(t) = 1\big) \wedge \big(\forall_{v \in X - \{s,t\}} \deg(v) = 2\big) \end{aligned}$$

Also, $\mathrm{isDisjoint}(X_1, ..., X_k)$ is true, if and only if all X_i are pairwise disjoint:

$$\mathrm{isDisjoint}(X_1, ..., X_k) = \forall_{1 \le i \ne j \le k} \forall_{x \in X_i} x \notin X_j.$$

Finally,

$$|X| \le l = \exists_{v_1, v_2, ..., v_l \in V} \forall_{x \in X} \big(\vee_{i=1}^{l} x = v_i\big).$$

This way, we encoded the problem as Theorem 5.6 requested. As $||\phi|| = O(k+l)$, Courcelle's theorem gives us the existence of an $O(f(k,l,w) \cdot n)$ algorithm.

Bibliography

[1] F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, M. A. Langston, W. H. Suters, and C. T. Symons. Kernelization Algorithms for the Vertex Cover Problem: Theory and Experiments. *ALENEX/ANALC*, 69, 2004.

[2] S. Arnborg, J. Lagergren, and D. Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12:308–340, 1991.

[3] S. Arora and B. Barak. *Computational complexity: a modern approach.* Cambridge University Press, 2009.

[4] H. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Computing*, 25:1305–1317, 1996.

[5] H. Bodlaender, B. Jansen, and S. Kratsch. Cross-composition: A new technique for kernelization lower bounds. *Proceedings of the 28th International Symposium on Theoretical Aspects of Computer Science (STACS), Leibniz International Proceedings in Informatics (LIPIcs)*, 9:165–176, 2011.

[6] J. Buss and J. Goldsmith. Nondeterminism within P. *SIAM J. Computing*, 22:560–572, 1993.

[7] L. Cai, J. Chen, R. G. Downey, and M. R. Fellows. Advice classes of parameterized tractability. *Annals of pure and applied logic*, 84(1):119–138, 1997.

[8] Y. Cao, J. Chen, and Y. Liu. On Feedback Vertex Set, New Measure and New Structures. *CoRR*, abs/1004.1672, 2010.

[9] J. Chen, F. Fomin, Y. Liu, S. Lu, and Y. Villanger. Improved algorithms for feedback vertex set problems. *J. Computer and System Sciences*, 74:1188–1198, 2008.

[10] J. Chen, I. A. Kanj, and G. Xia. Improved Parameterized Upper Bounds for Vertex Cover. *Theoretical Computers Science*, 411:3736–3756, 2010.

[11] B. Chor, M. Fellows, and D. Juedes. Linear kernels in linear time, or how to save k colors in $O(n^2)$ steps. In *Graph-theoretic concepts in computer science*, pages 257–269. Springer Berlin Heidelberg, 2004.

[12] B. Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Information and computation*, 85(1):12–75, 1990.

[13] M. Cygan, F. V. Fomin, Ł. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, and S. Saurabh. *Parameterized Algorithms.* Springer, 2015.

[14] M. Cygan, J. Nederlof, M. Pilipczuk, M. Pilipczuk, J. M. van Rooij, and J. O. Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on.* IEEE, 2011.

[15] F. Dehne, M. Fellows, M. Langston, F. Rosamond, and K. Stevens. An $O(2^{O(k)}n^3)$ FPT algorithm for the undirected feedback vertex set problem. *Springer: Proceedings of the 11th Annual International Conference on Computing and Combinatorics (COCOON), Lecture Notes in Comput. Sci.*, 3595:859–869, 2005.

[16] H. Dell. A simple proof that AND-compression of NP-complete problems is hard. *Electronic Colloquium on Computational Complexity (ECCC)*, 21:75, 2014.

[17] R. Downey and M. Fellows. Fixed-parameter tractability and completeness. *Proceedings of the 21st Manitoba Conference on Numerical Mathematics and Computing*, 87:161–178, 1992.

[18] R. G. Downey and M. R. Fellows. *Parameterized complexity.* Springer Science and Business Media, 2012.

[19] A. Drucker. New limits to classical and quantum instance compression. *Proceedings of the 53rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 609–618, 2012.

[20] T. Eilam-Tzoreff. The disjoint shortest paths problem. *Discrete Applied Mathematics*, 85:113–138, 1998.

[21] T. Fenner, O. Lachish, and A. Popa. *Approximation and Online Algorithms*, chapter Min-Sum 2-Paths Problems, pages 1–11. Springer International Publishing, Cham, 2014.

[22] J. Flum and M. Grohe. *Parameterized Complexity Theory.* Berlin: Springer Verlag, 2006.

[23] L. Fortnow and R. Santhanam. Infeasibility of instance compression and succinct PCPs for NP. *J. Computer and System Sciences*, 77:91–106, 2011.

[24] J. Guo, J. Gramm, F. Hüffner, R. Niedermeier, and S. Wernicke. Compression-based fixed-parameter algorithms for feedback vertex set and edge bipartization. *J. Computer and System Sciences*, 72:1386–1396, 2006.

[25] L. Hans, G. Rodney, R. Michael, and H. Danny. On Problems Without Polynomial Kernels. *Journal of Computer and System Sciences*, 75:423–434, 2009.

[26] Z. Király. Algoritmuselmémet. Lecture notes, Eötvös Loránd University, 2015. http://www.cs.elte.hu/∼kiraly/Algoritmusok.pdf.

[27] Y. Kobayashi and C. Sommer. On shortest disjoint paths in planar graphs. *Discrete Optimization*, 7(4):234–245, 2010.

[28] T. Kociumaka and M. Pilipczuk. Faster deterministic feedback vertex set. *Information Processing Letters*, 114(10):556–560, 2014.

[29] H. Lenstra, Jr. Integer programming with a fixed number of variables. *Mathematics of operations research*, 8:538–548, 1983.

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

[31] G. Nemhauser and L. Trotter Jr. Properties of vertex packing and independence system. *Math. Programming*, 6:48–61, 1974.

[32] R. Niedermeier. *Invitation to fixed-parameter algorithms, volume 31 of Oxford Lecture Series in Mathematics and its Applications*, volume 70. Oxford University Press, Oxford, 2006.

[33] V. Raman, S. Saurabh, and C. Subramanian. Faster fixed parameter tractable algorithms for finding feedback vertex sets. *ACM Transactions on Algorithms*, 2:403–415, 2006.

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

[35] N. Robertson and P. Seymour. Graph minors. XIII. The disjoint paths problem. *Combinatorial Theory Ser. B*, 63:65–110, 1995.

[36] S. Saurabh. *Exact algorithms for optimization and parameterized versions of some graph-theoretic problems.* PhD thesis, Homi Bhaba National Institute, Mumbai, 2007.

[37] C. Yap. Some consequences of non-uniform conditions on uniform classes. *Theoretical Computer Science*, 26:287–300, 1983.