Recent techniques in algorithm design

by Zoltán Király

Version 3

1 Introduction

Usually class P is considered as a synonym for efficiently solvable problems. Deciding whether a graph has a complete subgraph on one thousand vertices can be solved in $O(n^{1000})$ – that is in polynomial time – however this is not considered as an effective solution. To dissolve this contradiction the concept of class FPT (Fixed-parameter tractable problems) was introduced in 1992 by Downey and Fellows. Here problems (besides the input) have a natural parameter (usually denoted by k), and in the running time the exponent of the input length *must not depend* on k. The typical running time is usually something like $O(2^k \cdot n^2)$ or $O(4^k \cdot n^2)$ or $O(k^k + n^2)$, for large inputs these are much more practical than $O(n^k)$.

Example 1. *Suppose we have algorithms A1-A4 with the following running times (here we only count $n = |V|$ in the bounds).*
A1: n^k; A2: $2^k \cdot n^2$; A3: $k^k + n^2$; A4: $4^k \cdot n^2$.
Let G be a relatively small graph with 1000 vertices, and suppose $k = 10$. When running these algorithms on your laptop the estimated running time for these algorithms is the following.
A1: $3 \cdot 10^{15}$ years; A2: 1.7 min; A3: 16.7 min; A4: 1.2 day.
(*The age of the universe is less than $1.4 \cdot 10^{10}$ years.*)

In the last twelve years 4 books and more than one thousand papers were published on this topic. Subsequently an enormous number of new algorithms were designed for fixed-parameter problems, together with developing new concepts and new techniques as well as new analyzing methods. The purpose of this talk is to give an introduction to this topic and to explain some basic methods developed.

Prerequisites: Basics of complexity theory (big O notation, classes P and NP, NP-hard problems), elementary graph theory, fundamental algorithms (sorting, BFS, DFS, dynamic programming, Dijkstra's algorithm for shortest paths, maximum matching in bipartite graphs)

2 Definitions

In a parametric problem the input consists of a pair (x, k), where $x \in \{0,1\}^N$ and k is a positive integer. A (decision) problem $\mathcal{P}$ is a subset of all possible inputs, we call call this subset the set of yes-instances. An algorithm solves $\mathcal{P}$ if it stops on every input, and answers YES if and only if the input is a yes-instance.

If x is intended to describe a graph $G = (V, E)$, then is more convenient to use $n = |V|$ and $m = |E|$ instead of the length in bits. We assume the graph is given by edge lists, so $N = O((n + m) \log n)$ is the true bit complexity, however, we consider $n+m$ as the input length and we assume we can handle (e.g., read) the name of a vertex in one step.

Example 2. *Problem* Vertex-Cover*. The input consists of an undirected simple graph $G = (V, E)$ and the parameter k. It is a yes-instance if and only if G has a vertex cover T with $|T| \leq k$. ($T \subseteq V$ is a* vertex cover *if it contains at least one end-vertex of every edge.)*

We say that problem $\mathcal{P}$ is in the class XP (or that $\mathcal{P}$ is slice-wise polynomial) if there is an algorithm solving it for every input (G, k) in time $O(f(k) \cdot N^{g(k)})$ for some (computable) functions f, g. This is equivalent to requiring a solver that runs in polynomial time for every fixed constant k.

We say that problem $\mathcal{P}$ is in the class FPT (or that $\mathcal{P}$ is fixed parameter tractable) if there is an algorithm solving it for every input (G, k) in time $O(f(k) \cdot N^c)$ for some fixed constant c and (computable) function f. (Now the exponent of N must not depend on k. For example $O(k! \cdot N^3)$ is an acceptable running time.) If such an algorithm exists, we call it an FPT-algorithm.

A *reduction rule* is a function Φ mapping from the instances to the instances, such that if $\Phi(G, k) = (G', k')$, then either both (G, k) and (G', k') are yes-instances or both are no-instances (we will call such a pair *equivalent instances.*) A reduction rule is polynomial if it is computable in time $O((n + m + k)^c)$ for an absolute constant c.

A polynomial reduction rule is a *kernel* if $|V(G')| \leq f'(k)$ for a computable function f' and moreover $k' \leq k$.

The idea behind this is the following: after polynomial preprocessing we make a small equivalent instance. For a graph with $f'(k)$ vertices every usual decision problem can be decided in time $O(f(k))$ for another function f, usually $f(k) = c^{f'(k)}$ is sufficient.

In the first part we describe some basic techniques worked out for the problem Vertex-Cover.

3 Main tool: Kernelization

In order to make a kernel for VERTEX-COVER we give a polynomial algorithm which makes an equivalent instance (G', k') with $k' \leq k$ and $n' := |V(G')| \leq k(k+1)$. The algorithm is given via reduction rules; always the first rule must be used whose condition is met.

Algorithm 1. *Initialize $T = \emptyset$, $G' = G$ and $k' = k$. Use one of the following rule (in order) until they applies.*

i) If $k' = 0$ and G' is empty, then return YES and T as a cover. If $k' = 0$ and G' has some edges, then return NO.

ii) If G' has an isolated vertex, then simply delete it.

iii) If G' has a vertex v with degree strictly more than k', then $T := T + v$, delete v and $k' := k' - 1$.

iv) If none of the rules above applies, and either $n' > k'(k'+1)$ or $m' > k'^2$, then return NO; otherwise return (G', k', T).

Theorem 3.1. *If the algorithm stops with an answer, then this answer is correct. Otherwise, when it finishes, the current graph G' and the current k' describe an equivalent instance, and if T' is a vertex cover of G', then $T \cup T'$ is a vertex cover of G. Moreover Algorithm 1 runs in polynomial time, $k' \leq k$ and when it returns (G', k'), then $n' \leq k^2 + k$ and $m' \leq k^2$.*

Proof. By definition, either v or its neighborhood $N(v)$ must be in the cover. If $d(v) > k$ and there exists a cover of size at most k, then v must be in the cover.

If $k' = 0$, then $|T| = k$ and it consist of vertices that must be in the cover, so either T is a cover (and the answer is YES) or there are edges remaining not covered by T, in this case the right answer is NO.

When neither of the first 3 rules can be applied, every vertex has degree $1 \leq d(v) \leq k'$. If $n' > k'(k'+1)$, then no cover in G' of size k' can exist because such a cover would have at most k'^2 neighbors and every vertex is either in the cover or has a neighbor in the cover. Similarly if $m' > k'^2$, then no cover in G' of size k' can exist because such a cover can be incident with at most $k' \cdot k'$ edges, and as it is a cover, it must be incident with every edge. As we never increase k', clearly $k' \leq k$ (and so $k'^2 \leq k^2$). □

Exercise 1. *Show that with appropriate data structures Algorithm 1 runs in time $O(n + m)$.*

As for G' we can check every possible set of size k' whether it is a cover in time $O(2^{k'(k'+1)} \cdot n'^2) = O(2^{k(k+1)} \cdot k^4)$, we get the following.

Corollary 3.2. VERTEX-COVER *is in FPT.*

4 Second tool: bounded search tree

Algorithm 2. *Initialize* $T = \emptyset$, $G' = G$ *and* $k' = k$.
Call recursive algorithm $\mathrm{VC}(G', k', T)$.

$\mathrm{VC}(G', k', T)$: /* *if there is vertex cover* T' *of size* k' *in* G' *we want to return YES and* $T \cup T'$

Let v *be a vertex of* G' *with* maximum degree. *If* $d'(v) = 0$ *and* $k' = 0$, *then return YES and* T; *if* $k' = 0$ *and* $d'(v) > 0$, *then return NO.*

If $d'(v) = 1$, *then let* X *consists of one end-vertex of each edge in* E', *and if* $|X| \leq k'$, *then return YES and* $T \cup X$, *otherwise return NO.*

From now on $d'(v) \geq 2$. *Call* $\mathrm{VC}(G'-v,\ k'-1,\ T+v)$, *if it returned YES we also return YES and the set we got as a cover. If it returned NO, then we also call* $\mathrm{VC}(G'-v-N'(v),\ k'-d'(v),\ T \cup N'(v))$ *if* $k' \geq d'(v)$; *otherwise, if* $k' < d'(v)$, *then we return NO.*

If this second call returned NO, then we also return NO, otherwise we return YES and the set we got as a cover from this second call.

Theorem 4.1. *Algorithm 2 solves* VERTEX-COVER. *The number of calls is bounded by* $2 \cdot 1.6181^k$, *so the total running time is* $O(1.6181^k \cdot (n+m))$.

Proof. The first statement is obvious. A run of the algorithm can be described by a binary tree, where the nodes are labeled by the recursive calls, especially the leaves are labeled by such calls where further recursion is unnecessary.

A binary tree having L leaves may have at most $2L$ nodes. At a non-leaf vertex labeled by (G', k'), one of its children is labeled by $(G'-v, k'-1)$, the other child is labeled by $(G'-v-N'(v), k'-d'(v))$ where $d'(v) \geq 2$. Let $\phi = \frac{\sqrt{5}+1}{2} < 1.6181$. It is easy to prove by induction that the number of leaves is bounded by F_k, the k-th Fibonacci number, which is known to be less than ϕ^k.

The running time is bounded by the number of calls multiplied by the number s of steps inside one call. It is easy to see that $s = O(n+m)$. □

Exercise 2. *a). If the maximum degree is two in a graph, then a minimum cover can be computed in $O(n)$.*
b). Use this fact for modifying Algorithm 2, and prove that the new version has running time $O(1.466^k \cdot (n+m))$.

Now we turn to another crucial point. The two method discussed so far can be successfully applied together. First we make the kernel of the previous section in time $O(n+m)$, this kernel has size $n'+m' = O(k^2)$. We apply the above mentioned bounded search tree method not to the original graph but to the kernel! Doing so the joint running time of the two parts is $O(n+m+1.466^k \cdot k^2)$ (using the bound stated in Exercise 2).

5 Smaller kernel via crown reduction

The kernel we made may have more than k^2 vertices. Can we make a smaller one? Though we cannot make the bound on the number of edges in the kernel below k^2, it is much nicer to get the smaller instance with only a linear number of vertices.

A crown decomposition is a partition of the vertex set into $V = C \cup H \cup B$ with the following properties.

i) C is an independent set.

ii) No edge connects C to B.

iii) In the bipartite graph with classes H and C, there is matching covering H.

Statement 5.1. *There exist a k-element vertex cover in G if and only if there exist a $(k-|H|)$-element vertex cover in $G[B] = G-H-C$.*

Proof. Let T be a vertex cover in $G[B]$. As C is independent, $T \cup H$ is a vertex cover in G.

Now let T be a minimum vertex cover in G. By Property iii) we have a matching M between H and C with size $|H|$. Its edges are covered by T, consequently $|T \cap (H \cup C)| \geq |H|$. The edges inside B are covered by $T \cap B$ and all the other edges can be covered by H. Consequently $T' = (T \cap B) \cup H$ is also a minimum vertex cover in G, so $|T \cap B| = |T| - |H|$. □

We make another kernel for Vertex-Cover where $n' \leq 3k$.

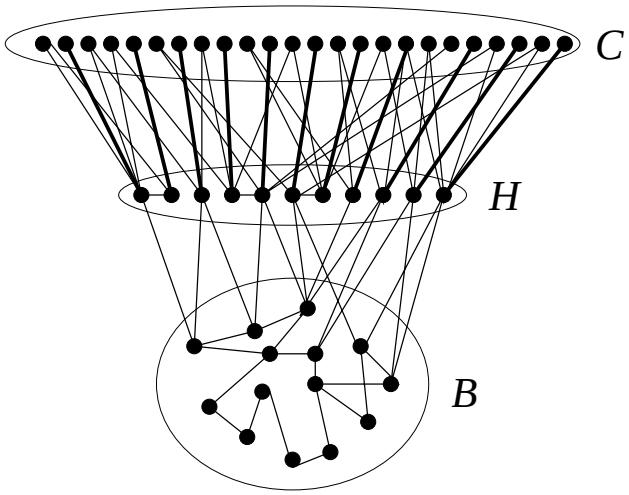

Figure 1: Crown decomposition

Algorithm 3. *Initialize* $T = \emptyset$, $G' = G$ *and* $k' = k$.

while $n' > 3k'$
Delete the isolated vertices
Let M *be an inclusion-wise maximal matching in* G'
if $|M| > k'$ ***then return(NO)***
Let X *consists of the* M*-covered vertices and* $I := V' - X$
Calculate the maximum matching M' *in the bipartite graph* G_b
containing the edges between I *and* X*;*
and a minimum vertex cover T' *for* G_b
if $|M'| > k'$ ***then return(NO)***
if $T' \cap X \neq \emptyset$ ***then***
$H := T' \cap X$; $C := I - T'$; $B := V' - H - C$
$G' := G'[B]$; $k' := k - |H|$; $T := T \cup H$
return(G', k', T)

Theorem 5.2. *If the algorithm stops with a NO answer, then this answer is correct. Otherwise, when it finishes, the current graph* G' *and the current* k' *describe an equivalent instance, and if* T' *is a vertex cover of* G'*, then* $T \cup T'$ *is a vertex cover of* G*. Moreover Algorithm 3 runs in polynomial time,* $k' \leq k$ *and when it returns* (G', k')*, then* $n' \leq 3k$.

Proof. Clearly any vertex cover must have size at least the size of any matching. As M is maximal, I is independent. If we did not stop after calculating M, then $|X| \leq 2k'$.

We calculate M' by the augmenting-path method of Kőnig, it also gives a minimum cover T' with the property $|M'| = |T'|$.

In the case $T' \cap X \neq \emptyset$ we claim that the partition given in the algorithm is a crown decomposition. C is independent as $C \subseteq I$. As T' is a vertex cover of G_b (and I is independent in G), edges from C must go into H. Finally as $|M'| = |T'|$, matching M' covers H, and the other end-vertices cannot be in T', so they are in C.

When $T' \cap X = \emptyset$, we have $T' = I$ as we deleted all isolated vertices. Consequently $|I| = |M'| \leq k'$ and we deduced that $|X| \leq 2k'$, so $n' = |V'| = |I \cup X| \leq 3k' \leq 3k$.

With a small trick we can make Algorithm 3 to run in $O(n + k^2 m)$. For this observe first that – as k' decreases – we have at most k iterations of the **while** loop. When we run Kőnig's augmentation algorithm we can stop after $k' + 1$ augmentations because at this point we already have $|M'| > k'$. □

Exercise* **3.** *Make an even smaller kernel, namely of size $n' \leq 2k$.*

Hint. A half-integral matching is a function $\mu : E \to \{0, 0.5, 1\}$ where at every vertex the sum of the μ-values on incident edges is at most one. Its size is $\sum_{e \in E} \mu(e)$. First show that the size of any half-integral matching is a lower bound for the minimum vertex cover. Then show that we can calculate the maximum size of a half-integral matching using Kőnig's algorithm for an appropriate bipartite graph, in time $O(km)$ if we may stop when the size grew above k (and this algorithm also calculates a half-integral vertex cover with the same size). Using this cover we can make the kernel required by just one crown reduction.

The current best algorithm for Vertex-Cover runs in time $O(n + km + 1.274^k)$.

6 Another parameter for Vertex-Cover

Is the bound on the size of the vertex cover is the right parameter? Usually, given a graph G, we are looking for the smallest vertex cover whose size is denoted by $\tau(G)$. Unfortunately, for a typical graph $\tau(G) = c \cdot n$, thus it is not a small value.

A much more practical parameter is the following. Let $\nu(G)$ denote the size of the maximum matching in G, we have already seen that $\tau(G) \geq \nu(G)$. The parametric problem Vertex-Cover-above-Matching asks for a vertex cover with size at most $\nu(G) + k$.

Example 3. *If G is a cycle on* 1001 *vertices, then* $\tau(G) = 501$*, thus* Vertex-Cover *finds a cover if we run it with parameter* 501*. On the other hand* $\nu(G) = 500$*, so an algorithm solving* Vertex-Cover-above-Matching *finds it with parameter* 1*.*

Without any details we mention the following

Theorem 6.1. Vertex-Cover-above-Matching *can be solved in time* $O(4^k \cdot n^c)$*.*

7 Longest path, randomization and color coding

Given a graph G and a vertex s, it is well-known that computing the longest path starting from s is an NP-hard problem. An essentially equivalent (see the remark at the end of this section) parametric problem is Exact-Path, when a parameter k is also given, and we ask for a path starting at s and have length exactly k.

Randomization helps a lot in many cases of algorithm design, we see now an example for Exact-Path. This method is called *color coding*.

We color vertices in $V - s$ randomly and independently by colors $C := \{1, 2, \ldots, k\}$. We call a path (starting from s) C-colorful if its vertices (except s) have distinct colors and every color in C acts as a color of a vertex in the path. Note that if we have a C-colorful path, then its length is exactly k.

Statement 7.1. *If we make* $M = 185 \cdot e^k$ *independent colorings of the graph, and neither admits a* C*-colorful path, then the probability of having a path starting at* s *with length exactly* k *is at most* e^{-185}*.*

Proof. Suppose P is a path starting at s and have length exactly k. In a random coloring the probability that P is C-colorful is $k!/k^k$. So the probability that every coloring is bad is $(1 - k!/k^k)^M < (1 - e^{-k})^M < e^{-185}$. *Remark: there are less than* e^{185} *particles in the (observable) universe.* □

Colorful paths in a colored graph can be find by dynamic programming.

Algorithm 4. *The subproblems to solve: for every vertex* $v \in V - s$ *and for every color set* $C' \subseteq C$ *we decide whether there is* C'*-colorful path from* s *to* v *(and if the answer is YES, then we remember the last vertex before* v*).*

So $\mathrm{Cf}(v, C') = \bigvee_{u \in N(v)} \mathrm{Cf}(u, C' - c(v))$ *if* $c(v)$*, the color of* v *is an element of* C'*, otherwise* $\mathrm{Cf}(v, C') =$*FALSE.*

Algorithm 4 runs in time $O(2^k m)$, so with the repetition described in Statement 7.1 we get a randomized algorithm running in time $O((2e)^k m)$ with the following property. If we have a no-instance of Exact-Path, then the algorithm surely answers NO, otherwise it returns a path of length k with probability at least $1 - e^{-185}$. (And, with probability at most e^{-185}, it answers erroneously NO).

Remark 7.2. Suppose in the input graph G the longest path starting from s has length K. We run the algorithm above successively for $k = 1, 2, \ldots$ until we get a NO answer. The total running time will be

$$c \cdot m \cdot \sum_{k=1}^{K+1} (2e)^k < c \cdot m \cdot \frac{(2e)^{K+2}}{2e-1}.$$

8 Feedback vertex set and iterative compression

The last design method we discuss is *iterative compression*.

We show an example for this method through a new problem. In a graph G, a subset X of the vertices is called a *feedback vertex set* if $G-X$ is a forest (i.e., X blocks every cycle).

The problem Feedback-Vertex-Set asks whether – given G and k – a feedback vertex set of size at most k exists in G.

The basic idea of the method "iterative compression" is the following. Let $V = \{v_1, v_2, \ldots, v_n\}$ and let V_i denote $\{v_1, v_2, \ldots, v_i\}$ and $G_i = G[V_i]$. We are going to construct feedback vertex sets $X_1, \ldots, X_n$ for $G_1, \ldots, G_n$ of size at most k consecutively. For $i \leq k$ it is easy: $X_i = V_i$ suffices. (If for any i we have no such set, then we return NO.)

In a general step we are given X_{i-1}, a feedback vertex set of size at most k for G_{i-1}. Clearly $Z := X_{i-1} + v_i$ is a feedback vertex set of size at most $k+1$ for G_i. Starting with this as a hint, we are going to construct X_i. If $|Z| \leq k$, then we are done by defining $X_i := Z$, so we may assume that $|Z| = k+1$.

First we define a related problem Disjoint-Feedback-Vertex-Set. Here G, k and a feedback vertex set $W \subseteq V$ is given, and the objective is to find a feedback vertex set $X \subseteq V - W$ of size at most k, or correctly conclude that no such feedback vertex set exists. We give an algorithm for Disjoint-Feedback-Vertex-Set running in time $O(4^k \cdot n^c)$.

Denote $G-W$ by H; now H is a forest (as W is a feedback vertex set), and if $G[W]$ is *not* a forest, then we may stop with answer NO (as the whole $V(H)$ is not a feedback vertex set).

First apply some basic reduction rules. Execute on of the following rules until none of them applies.

i) If there exists $v \in V$ with $d(v) \leq 1$, then delete it.

ii) If there exists a vertex v of H such that $G[W+v]$ contains a cycle, then include v in the solution, and let the new instance be $(G-v,\ W,\ k-1)$.

iii) If there is a vertex v of H of degree 2 in G with neighbors u and w, and u is a vertex of H, then delete v and add a new edge uw if $w \in V(H)$, otherwise (if $w \in W$) we have two possibilities. If uw is an edge, then include u in the solution, and let the new instance be $(G-u-v,\ W,\ k-1)$. If uw is not an edge, then delete v and add a new edge uw.

Exercise 4. *We made an equivalent instance.*

From now on let (G, k, W) denote a reduced instance (none of the rules above applies). If $k = 0$, then we return YES if G is a forest, and NO otherwise.

Let x be a vertex of H with at most one neighbor in H (as H is a forest, such a vertex exists). Vertex x must have at least two neighbors in W (as rule i) an iii) were not applicable) but cannot have two neighbors in the same component of $G[W]$ because rule ii) was not applicable. Now we call recursively for $(G-x,\ W,\ k-1)$ and also for $(G,\ W+x,\ k)$. If both calls answer NO, then we also answer NO. If one of them answers YES, then we answer YES, and if this answer was given by the first call, then we also add x to the solution given by it.

As if there is a feedback vertex set required, then either x is inside it or not, thus this algorithm is correct.

Exercise* 5. *Prove the running time $O(4^k \cdot n^c)$ claimed.*

Hint. Take the value $c(G[W]) + k$ where $c(G[W])$ denotes the number of components. Prove that when we branch, this value strictly decreases for both the two new instances we create.

Now we remained to show how to use this algorithm for our original problem Feedback-Vertex-Set. We are looking for a feedback vertex set X_i of size at most k, given feedback vertex set Z of size exactly $k+1$. We branch on possible sets $Y = Z \cap X_i$ ($2^{k+1} - 1$ branches, as $Y = Z$ is not good as it has size $k+1$). Let $W := Z - Y$ and call Disjoint-Feedback-Vertex-Set with $(G-Y, W, k-|Y|)$. If on any branch it returns YES and a feedback vertex set $X \subseteq V-Y-W$ with size at most $k-|Y|$, then we found the suitable $X_i := X \cup Y$. Otherwise, if on every branch we got NO, then we also answer NO.

Exercise* **6.** *Show that the total running time is* $O(5^k n^{c+1})$.

Hint. Use Newton's binomial theorem.

9 Suggested reading

M. Cygan, F.V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk and S. Saurabh: *Parameterized Algorithms*, Springer, 2015.

R.G. Downey and M.R. Fellows: *Parameterized complexity*, Springer, 1999.

J. Flum and M. Grohe: *Parameterized Complexity Theory*, Texts in Theoretical Computer Science (EATCS Series), Springer, 2006.

R. Niedermeier: *Invitation to fixed-parameter algorithms*, Oxford Lecture Series in Mathematics and its Applications, Oxford University Press, 2006.