# Advanced Algorithms Course. Lecture Notes. Part 9

## Algorithms for Problems on Special Instances

### Small Vertex Covers – XP and FPT

The Vertex Cover problem in graphs is NP-complete, but if the graph is already known (or expected) to have some vertex cover with a "small" number $k$ of nodes, we can still solve it exactly and efficiently in practice.

Let $n$ always denote the number of nodes in the given graph. A naive way to find a small vertex cover is to test all subsets of $k$ nodes exhaustively. Elementary combinatorics tells us that this costs $O(kn^{k+1}/k!)$ time: Note that $O(kn)$ time is sufficient to test whether a given set of $k$ nodes is a vertex cover, and the other factor comes from $\binom{n}{k}$. This time bound is feasible only for very small $k$. The bad thing is that $k$ appears in the exponent of $n$. It would be much better to have a time bound of the form $O(b^k p(n))$, where $b$ is a constant base, and $p$ some fixed polynomial. (To get a feeling of the tremendous difference, try some concrete figures and compare the naive time bound for Vertex Cover with the bounds we will obtain below.)

A problem with input length $n$ and another input parameter $k$ is said to be in the **complexity class XP** if it can be solved in $O(n^{f(k)})$ time, where $f$ is any computable function. A problem with input length $n$ and another input parameter $k$ is called **fixed-parameter tractable (FPT)** if it can be solved in $O(f(k) \cdot p(n))$ time, where $f$ is any computable function (usually exponential) and $p$ is some polynomial. We may write $O^*(f(k))$ instead of $O(f(k) \cdot p(n))$ if we want to suppress the polynomial factor and stress the more important parameterized part of the complexity.

In the following we show that Vertex Cover is not only an XP problem but an FPT problem. The basic algorithm is: Take an uncovered edge $(i, j)$ and put node $i$ or node $j$ in the solution. Repeat this step recursively *in both branches*, until $k$ nodes are chosen or all edges are covered.

Upon every decision ($i$ or $j$) we create new branches, hence the whole process has the form of a recursion tree that we call a **bounded search tree**. Since at most $k$ nodes of the graph are allowed in a solution, the tree has depth at most $k$, thus at most $2^k$ leaves and $O(2^k)$ nodes. If some leaf represents a vertex cover, we have found a solution, otherwise we know that there is no solution. To bound the time complexity, it remains to check how much time we need to process any node of the search tree: In a simple implementation we may copy the whole graph, delete in one copy all edges incident to $i$, and delete in one copy all edges incident to $j$ (because these edges are already covered). The main work is copying. Here we observe that the whole graph can have at most $kn$ edges, otherwise no vertex cover of size $k$ can exist. Hence copying costs $O(kn)$ time, and the overall time is $O(2^k kn) = O^*(2^k)$.

Although this is already much better than naive exhaustive search, further improvements would still be desirable. Here, the more important part is the exponential factor $2^k$. Can we improve the base 2 and thus make the algorithm practical for somewhat larger $k$?

The weakness of the search tree algorithm above is that it considers single edges and selects only one vertex at a time. If we could select more vertices, we could generate our solutions faster. Now observe: For any node $i$, we have to take *$i$ or all its neighbors*, in order to cover all edges incident to $i$. It might be good to apply this branching rule on nodes $i$ of high degree. But what if the graph has no high-degree nodes?

If all degrees are at most 2, the graph consists of simple paths and cycles, and the problem is trivial. Thus we can assume (worst case!) that there is always a node of degree 3 or larger. In a branching step we take either 1 node or 3 nodes (or more). How large is our search tree?

This can be analyzed by recurrence equations, similar to the analysis of divide-and-conquer algorithms. Let $T(k)$ be the number of leaves of a search tree for vertex covers of size $k$. Due to our branching rule we have $T(k) = T(k-1)+T(k-3)$. To figure out what function $T$ is, we assume that it has the form $T(k) = x^k$ with an unknown constant base $x$. Our recurrence becomes $x^k = x^{k-1} + x^{k-3}$, which simplifies to $x^3 = x^2 + 1$. This equation is called the **characteristic equation** of the recurrence. Numerical evaluation shows $x \approx 1.47$, which is much better than 2. Researchers have invented more tricky branching rules for Vertex Cover and further accelerated the branching process. Meanwhile the best known base is below 1.3.

Anyway, we have shown the time bound $O(1.47^k kn) = O^*(1.47^k)$.

## Kernelization

For the problem of finding a vertex cover of size at most $k$ we have shown the time bound $O(1.47^k kn) = O^*(1.47^k)$. Can we also improve the polynomial factor?

Observe that any node $i$ of degree larger than $k$ is necessarily in the solution. (If we do not select $i$, we have to take all neighbors, but these are too many.) Thus we can put aside all nodes of degree larger than $k$. This can be done in $O(kn)$ time. There remains a graph where all nodes have degree at most $k$. Now, $k$ vertices can cover at most $k^2$ of the remaining edges. Hence, if the remaining graph has more than $k^2$ edges, we know that there is no solution. This also means: In the positive case we have found a subgraph with at most $k^2$ edges, such that it remains to solve the hard Vertex Cover problem on this small graph only. The resulting time bound is $O(1.47^k k^2 + kn)$. Note that we got rid of the product of the exponential term and $n$.

The above process is called **kernelization**, and the remaining small graph is called a problem **kernel**. We skip the exact technical definition, however we observe that the size of the kernel depends only on the parameter $k$, but not on the original number $n$ of nodes, and the kernelization needs only polynomial time.

To put these results in a much more general context: Kernelization is just a formal way of preprocessing an input, and it is widely used also outside FPT problems. The idea is to take away simple parts of an instance and give a miniaturized instance of the hard problem to the actual algorithm. Thus, the underlying algorithm has to deal only with the hard part of the instance, and if this is significantly smaller than the original instance, this saves much computation time.