

Chapter 1

What is this course about?

This course is about problem solving. In particular, given some problem, what is the “best” algorithm that we can design for it? What does “best” mean here; what are ways in which we measure how good an algorithm is?

We will give one specific sample problem below and show steps to be done for it without giving too many details. Along the way, it should also become clear what concepts should be known already and which ones we will see later in the course in more detail.

1.1 The vertex cover problem

1.1.1 The problem

Imagine you are looking at a map of a city. You want to distribute policemen in the city (e.g. during a riot) such that every street is guarded, i.e., for every street there is a policeman placed on it. Resources are tight (as always), and therefore you want to use as few policemen as possible. Where should you place them?

1.1.2 Modeling the problem

The first step in solving a problem is in modeling it, i.e., phrasing it in precise mathematical language, and clarifying what exactly is desired.

Modeling problems is not the main focus for this course (it belongs into the field of Operations Research, or is sometimes also done as “Requirements” in Software Engineering.) In fact, most problems will be given to you as precise mathematical formulation already, and the applications of it will only be hinted at. But for this problem, let’s formulate it precisely as follows.

A map of a city can nicely be modeled using a graph.¹ Namely, let the set V of vertices

¹You are expected to know the concept of graphs, how to store them, and basic algorithms such as depth-first search, breadth-first search, minimum spanning trees and shortest paths. In this course, graphs will always be stored with adjacency lists or variants thereof, so that a vertex v can explore its incident edges in $O(\deg(v))$ time, we can add/delete an edge in $O(1)$ time and a vertex in $O(\deg(v))$ time, etc.

be those places where 2 or more different streets meet. The set E of edges is exactly the streets. One can easily convince oneself that it makes no sense to put a policeman halfway down a street (it could be moved to an endpoint instead without requiring more policemen), so the desired locations of policemen should be a set C of vertices. Moreover, the set C must satisfy that for every edge of the graph, at least one endpoint is in C . This is the so-called *vertex cover problem*:

Problem 1 (Vertex Cover) *Given: A graph $G = (V, E)$.*

Find: A set $C \subseteq V$ such that for every edge, at least one endpoint is in V .

Objective: Make set C as small as possible.

As the name implies, set C is called a *vertex cover*, and any vertex in C is said to *cover* all its incident edges.

1.1.3 Design an algorithm

The next step is to design an algorithm for this problem.² For example, we could design a greedy-algorithm. A natural greedy algorithm would be to find a vertex of maximum degree, add it to C , and iterate, always taking the vertex that covers the maximum number of edges that weren't covered before.³ Pseudo-code of this algorithm would be as follows:

0. Initialize C as an empty set.
1. While G has edges left
2. Let v be a vertex of maximum degree in G .
3. Add v to C .
4. Delete v (and all its incident edges) from G .

Sometimes more details will be required. For example, how exactly will step 2 be implemented - will you search through all vertices to find the one with minimum degree, or will you use some complicated data structure? This should not affect the result of the algorithm, but it will affect its run-time.

1.1.4 Analyze the algorithm

The next step is to analyze the algorithm that you have just designed. There are actually two parts to it: analyze the run-time and analyze the quality.

²You should be familiar with basic algorithm design techniques, such as brute-force, greedy, divide&conquer, and dynamic programming.

³Whenever you describe an algorithm, you should start by describing the general idea in a few sentences, *before* you go into the details and/or give pseudocode.

Analyze the run-time

First let's analyze the run-time. Most data structures to store a graph have an explicitly list of edges, so we can certainly check in $O(1)$ time whether G has edges left, and Step 1 takes constant time. Step 3 takes constant time. Step 4 takes $O(\deg(v))$ time, which, summed over all vertices, is $O(m)$ time.⁴ That leaves as the only tricky part Step 2.

If we search through all vertices in Step 2, then this takes $O(n)$ time and the total time is $O(n^2 + m) = O(n^2)$. If we use a smarter data structure (such as hashing with open addressing), then one can show that Step 2, for all vertices together, only takes $O(n)$ time. (This fits in the topic of asymptotic analysis, of which we will hear more later.) Details of this are left as an exercise. Therefore, the total run-time is $O(n + m)$, i.e., linear in the input size.

Quality of the output

To judge the quality of the output, we first need to ask whether the algorithm is even correct, i.e., does it return what we need? In this case this is near-trivial: we do indeed get a vertex cover since an edge is removed only if one of its endpoints was added to the cover.

The second question to ask is whether the solution is as good as possible. In other words, do we achieve the objective, i.e., the minimum vertex cover? The answer in this case is “no”: try to construct an example yourself.

If the answer is “no”, then not necessarily all is lost. We could still prove bounds that show that the solution is not too bad. For example, for this algorithm, we can show that the solution is at most a factor of H_m worse than the best possible solution.⁵ This fits into the topic of approximation algorithms, of which we will hear more later in the course.

Some of the algorithms that we will see in the course have a random component to them, i.e., some random bit-flips. For such algorithms, one could ask the above question from the point of view of probability: What is the probability that we get a correct answer? What is the expected quality of the solution?⁶

1.1.5 Can we do better?

Whenever one algorithm has been designed and analyzed, this immediately raises the question: Can we do better? Is there an algorithm that has a better run-time? That achieves a better solution? Both?

In particular, we could do any of the following:

⁴You are expected to know asymptotic notation, such as O, Ω, Θ, o , as well as how to analyze loops and recursions, and how to evaluate basic summations and recursions.

⁵ H_m is the m th Harmonic number: $H_m = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{m} \in \Theta(\log n)$.

⁶You are expected to know basic results from probability theory. In particular, we need probabilities, expected values and variance, and also Markov's inequality and Chernoff bounds, though they will be reviewed briefly.

- Prove lower bounds on the run-time. For example, for Vertex Cover a lower bound of $\Omega(m)$ is quite obvious: We must look at every edge of the graph to know what its endpoints are, otherwise we cannot assure that all edges are covered.

For other problems, a lower bound of $\Omega(n \log n)$ can sometimes be proved using a linear reduction from Sorting.⁷ Other lower bounds will be proved using an adversary argument, which we will see later.

If the algorithm doesn't actually solve the problem, one should also try to prove the problem to be NP-hard.⁸ Vertex Cover is NP-hard (and in fact, is one of the oldest problems known to be NP-hard.) However, in this course (as opposed to what you may have seen in previous algorithms courses), NP-hardness is not the end of the problem. Quite the opposite: much of the course will be spent on what to do if a problem is NP-hard and you need to solve it anyway: What are ways to judge algorithms that necessarily can't be both polynomial and find the best solution?

We will sometimes even go a step beyond NP-hardness and prove such things as APX-hard and W[1]-hard (whatever that may be.) But this course is not a course in complexity theory, and the treatment of these topics will be short and superficial.

- Prove lower bounds on the quality. For example, for Vertex Cover one can show that no polynomial algorithm can find a solution that is better than 1.36 times the best solution (unless $P = NP$, but we will generally assume that $P \neq NP$.)

Such proofs are often fairly difficult, and we will only give some hints as to how to prove them without the full details.

- Consider special cases. This means going back to the modeling of the problem and asking whether perhaps there might be restrictions that make sense in the original problem and that could help you solve the problem.

For example, in our street-problem, the graph formed by the streets can be drawn without crossings. (Such a graph is called a *planar graph*.) Does this help for solving Vertex Cover? It turns out that the problem is still NP-hard even in planar graphs, but we can approximate it much better.

- Develop other algorithms and repeat the game. We will see many algorithm design techniques in this course, and many things that we hope to achieve. (For example, we will see randomized algorithms and how to analyze them, approximation algorithms and PTAS's, exact and fixed-parameter tractable algorithms, and others.) So for one problem, we might end up with many algorithms. And sometimes it won't even be

⁷You should know that Sorting takes $\Omega(n \log n)$ time in the *comparison-model*, i.e., if the only thing we're allowed to do with the input is to compare two numbers to see which one is bigger. You should be familiar with reductions, though we will see examples.

⁸You should be familiar with NP-hardness, as well as know some basic NP-hard problems such as 3-SAT, Vertex Cover and Independent Set.

clear which one the best one is: One algorithm may be better in one respect but worse in another.

For example, for vertex cover, we will see another (very simple) algorithm that is a 2-approximation and hence better than the greedy algorithm. For planar graphs, there is even a PTAS (whatever that may be). Vertex Cover is also fixed-parameter tractable (whatever that may be) in the output size.

1.2 A few more remarks

This is a *theory* course. In particular, you will not be asked to do any implementations. But on the flip side, you will be asked to do proofs. Lots of proofs. In essence, every statement ought to be justified. If you do not like reading and writing proofs, you will not be happy in this course.

Because this is a theory course, we will generally focus on the big idea behind the algorithm, and not so much on the details to make it happen. So while the pseudo-code should be detailed enough that an experienced coder should be able to convert it into actual code, it will definitely not be very detailed.

Because we focus on the big ideas, we don't want to be bothered with small annoying details necessitated by special cases. Therefore, we will quite frequently make simplifying assumptions. (Typical such assumptions are: “ n is a power of 2”, “all input numbers are distinct”, “no 3 input points are on a line”, etc.) It is quite important to write such assumptions down (so if someone wanted to code the algorithm, she would know where to be careful), but it is quite acceptable to make such assumptions, and on your assignments, this will lead to small if any deductions. However, it is vital that the cases you are leaving out are really not important, so you should always think through what you would do (but you don't need to write it down.) If in doubt, err on the side of too much detail for the assignments.