Advanced Algorithms Course. Lecture Notes. Part 8

3-SAT: How to Satisfy Many Conditions

The Satisfiability problem (SAT) asks to assign truth values to the variables in a Boolean formula so as to make the formula true. Specifically, the formula is given as a conjunction of clauses, where each clause is a disjunction of literals, i.e., unnegated or negated Boolean variables. SAT appears directly in many real problem settings where logical variables have to satisfy certain constraints. In 3-SAT, every clause has 3 literals. 3-SAT is a classical NP-complete problem. MAX 3-SAT is the following natural relaxation of 3-SAT: Find an assignment of truth values that satisfies as many clauses as possible. By an obvious reduction from 3-SAT we find that MAX 3-SAT is also NP-complete.

On the positive side, if any conjunction of k clauses with exactly 3 literals is given, we can easily find an assignment that satisfies most of the clauses, namely an expected number of $0.875k$ clauses. An extremely simple randomized algorithm will do: Assign truth values 0 or 1, each with probability $1/2$, to all variables independently. The analysis is very simple, too: Every clause is satisfied with probability $7/8$, hence, by linearity of expectation, an expected number of $7k/8$ clauses is satisfied.

We can conclude even more from this result: Since an *expected* number of $7k/8$ clauses is satisfied, there must always exist some truth value assignment that *actually* satisfies at least $7k/8$ clauses. This easily follows from a general argument: Consider any random variable X. Since $E[X]$ is the weighted average of X, variable X can *actually* take on some value greater than or equal to $E[X]$, with some positive probability. Now we just apply this observation to the variable X indicating the number of satisfied clauses in the result of our algorithm.

This reasoning is the famous **Probabilistic Method** that can be characterized as follows: When we look for a certain combinatorial structure (here: a truth assignment satisfying many clauses), we may apply some simple randomized algorithm and show that the desired structure is produced with some positive probability. Hence this structure must exist. Of course, this approach has some shortcomings: It does not work for any such problem (due to lack of a simple randomized algorithm), and it proves only the mere existence of the object we are looking for, but it does not say how we can find it efficiently. These questions must be further studied for any specific problem at hand.

In the case of MAX 3-SAT, how difficult is it to actually find an assignment that satisfies at least $7k/8$ clauses? The obvious idea is to iterate the above algorithm until success. Below we analyze the expected number of iterations needed. It would be a big mistake to believe that 2 expected iterations suffice, since we either succeed or fail. Of course, we must look into the probability distribution instead.

Let p_j be the probability of satisfying exactly j of the k clauses. Since the expected value of j is $7k/8$, we know the following equation, where the sum is already split in two cases:

$$7k/8 = \sum_j jp_j = \sum_{j<7k/8} jp_j + \sum_{j\geq 7k/8} jp_j.$$

We abbreviate the success probability by $p := \sum_{j\geq 7k/8} p_j$. Let k' denote the largest integer with $k' < 7k/8$. (It will become apparent soon why this is a clever definition.) We upperbound the sum generously and obtain

$$7k/8 \leq \sum_{j<7k/8} k'p_j + \sum_{j\geq 7k/8} kp_j = k'(1-p) + kp \leq k' + kp.$$

It follows $kp \geq 7k/8 - k'$, which is at least $1/8$ due to the definition of k'. Thus, a random assignment succeeds with probability $p \geq 1/(8k)$, hence the expected number of iterations until success is at most $8k$. Note that this is a Las Vegas algorithm.

The algorithm does not solve the actual MAX 3-SAT problem, as it guarantees only $0.875k$ satisfied clauses in every input. But what if, for example, $0.95k$ clauses are satisfiable ...? It has been shown that, for arbitrarily small $\epsilon > 0$, it is already NP-complete to decide whether a MAX 3-SAT instance allows to satisfy $(0.875 + \epsilon)k$ clauses. In this sense, running the simple randomized algorithm above is, amazingly, already the best one can do.

From now on we jump between randomized algorithms and another subject. This is not a course design mistake, rather, it has scheduling reasons.

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, compared to the number n of nodes, we can still solve it exactly and efficiently.

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. Unfortunately, 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 some 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 some computable function (usually exponential) and p is some polynomial. Note that FPT$\subset$XP.

Recall that the O-notation suppresss constant factors. In the analysis of FPT algorithms we may want to focus on the heavy non-polynomial terms in the time complexity. Thus we also introduce the more generous O^*-notation that even suppresses factors that are polynomial in n. That is, we may write $O^*(f(k))$ instead of $O(f(k) \cdot p(n))$. Since $k < n$, the O^*-notation also suppresses factors that are polynomial in k. Since in practice it would be weird to ignore polynomial factors, this should be seen only as a "lazy" notation for a rough analysis.

In the following we show that Vertex Cover is not only an XP problem but also an FPT problem. Let G be the input graph. Our basic algorithm is simply the following:

Take any uncovered edge (i, j) in G. Create two copies of G. In one copy, put node i in the solution. In the other copy, put node j in the solution. Repeat this step recursively *in both graphs*. In every copy of G, stop the process as soon as k nodes have been chosen or all edges are covered.

Upon every decision (for choosing i or j) we create copies of the problem instance that we call **branches**. We can think of the whole process as a recursion tree that we call a **bounded search tree**. How large is this tree, for the proposed algorithm?

Since at most k nodes of G are allowed in a solution, the search tree has depth at most k, and since the tree is binary, it has at most 2^k leaves, hence $O(2^k)$ nodes. If a vertex cover is found in a copy of G in some of the leaves of the search tree, then we have a solution, otherwise we know that no solution of size k can exist at all.

To bound the time complexity, it remains to study how much time we need in every node of the search tree. The main work is copying. We observe again that G can have at most kn edges, since 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 much better than naive exhaustive search, further improvements would be desirable. The primary concern is the exponential factor 2^k. Can we improve the base 2 and thus make the algorithm practical for larger k?

The weakness of the algorithm above is that it considers single edges and selects only one node at a time. If we could select more nodes at once, we could generate our potential solutions faster.

Indeed, a faster FPT algorithm is based on the following fact: For any node i, we have to take *i or all its neighbors*, in order to cover all edges incident to i. Accordingly, we modify the branching rule in our algorithm: In one copy, put node i in the solution. In the other copy, put all neighbors of i in the solution. The rest of the algorithm is unchanged.

Clearly, it is advantageous to apply this branching rule to nodes i of high degrees. But what if the graph has no high-degree nodes?

If all degrees are at most 2, then G consists of simple paths and cycles, and the Vertex Cover problem is trivial there. In general we can stop branching in a copy of G as soon as the graph after removal of the covered edges has maximum degree 2; the remaining instance is then solvable in linear time. Thus, in every branching step we take 1 node and at least 3 nodes, respectively, in the two branches. How large is now our search tree?

This can be analyzed by recurrence equations, similarly as for divide-and-conquer algorithms. Let $T(k)$ be an upper bound on the number of leaves of a search tree for vertex covers of size k. Due to our branching rule it fulfills $T(k) = T(k-1) + T(k-3)$. (Why?) Any recurrence of this form is called a linear recurrence, and it has a solutions of the form $T(k) = x^k$

with some 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. The best known base is below 1.3. Anyway, we have shown here 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 every node i in G 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 all nodes of degree larger than k in the solution, and delete the indicent edges, as they are covered. All this can be done in $O(kn)$ time.

There remains a graph H of yet uncovered edges, where all nodes have degree at most k. Thus, k vertices of H can cover at most k^2 edges of H. Hence, if H has more than k^2 edges, we know that no solution exists. This also means: In the positive case we have found a subgraph H with at most k^2 edges, such that it remains to solve the hard Vertex Cover problem on this small graph H 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 H is called a problem **kernel**. We skip the exact technical definition of kernels, however, we observe the important features in our problem example: The size of the kernel depends only on the parameter k, but not on the original size n, and the kernelization needs only polynomial time. Informally, the kernel contains the hard part of the problem instance, whereas the easy part has been cut away.

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 pass only a miniaturized instance of the hard problem to the actual algorithm. Thus, the underlying algorithm has to deal only with the hard part, and if it is significantly smaller than the original instance, this saves much computation time.