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: If the formula is not satisfiable, find an assignment of truth values that satisfies as many clauses as possible. By an obvious reduction from 3-SAT we see that MAX 3-SAT is also NP-complete.

On the other hand, 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 $0.875k$ in expectation. 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 more from this result: Since an expected number of $7/8$ of all clauses is satisfied, there must always exist some truth value assignment that actually satisfies at least $7/8$ of the clauses. This easily follows from a general argument: Consider the random variable X indicating the number of satisfied clauses. The expected value $E[X]$ is the average value of X, weighted by the probabilities of values. Hence any random variable X can take on some value greater than or equal to $E[X]$.

This reasoning is the famous **Probabilistic Method**: 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, the approach does not work for any such problem (due to lack of a simple randomized algorithm), and it proves only the existence of the object we are looking for, but it does not say how we can find it efficiently. These questions must be 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 7/8 of the clauses? The obvious idea is to iterate the above algorithm until success. We analyze the expected number of iterations needed.

Let k be the number of clauses, and let p_j be the probability of satisfying exactly j clauses. Since the expected value of j is $7k/8$, we have the following equation, where the sum is already split in two cases: $7k/8 = \sum_j jp_j = \sum_{j<7k/8} jp_j + \sum_{j\geq 7k/8} jp_j$. As an abbreviation we define $p := \sum_{j\geq 7k/8} p_j$, and we let k' be the largest integer with $k' < 7k/8$. We upperbound the sum generously and obtain

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

Thus we have $kp \geq 7k/8 - k'$, which is at least $1/8$ due to the definition of k'. Thus, a random assignment succeeds with probability $p \geq 1/8k$, and the expected waiting time for success is at most $8k$ iterations.

Note that this is a Las Vegas algorithm. Furthermore, note that it does not solve the actual MAX 3-SAT problem. It guarantees only $0.875k$ satisfied clauses in every input. But what if, for example, $0.95k$ clauses are satisfiable ...? In fact, it has been shown that, for any 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 is, amazingly, already the best one can do in general.

Median Finding and Selection

The so-called Selection problem is to find the element of rank k in a set S of n distinct numbers. The rank is the position that the element would have if S were sorted. A simpler formulation is: Find the k-th smallest element in S. Note that S is given in arbitrary order and is in general not sorted. The element with rank $\lfloor n/2 \rfloor$ is called the median. Median finding and Selection have nice applications in geometry and in the analysis of statistical data.

In addition to these motivations, recall the 1.5-approximation algorithn for Load Balancing. We had sorted the jobs by their lengths. A closer look reveals that it is enough to separate the m longest jobs from the shorter jobs, since neither the algorithm nor the analysis uses any sorting within these two sets of jobs.

To solve the Selection problem we may first sort S in $O(n \log n)$ time, which makes the problem trivial. But we can also avid sorting and solve Selection directly in $O(n)$ time. There exists a deterministic divide-and-conquer algorithm for Selection, but it is a bit complicated and, more importantly, the hidden constant in $O(n)$ is rather large. It is much more advisable to apply a simple randomized algorithm like the following.

Choose an element $s \in S$ called the splitter. Compare all elements to s, in $O(n)$ time. Now we know the rank r of s. If $r > k$ then throw out s and all elements larger than s. If $r < k$ then throw out s and all elements smaller than s, and set $k := k - r$. If $r = k$ then return s. Repeat this procedure recursively.

Correctness should be obvious. The only unspecified step is the choice of the splitter. Let us use the above scheme with a splitter chosen uniformly at random. That is, every element of S becomes the splitter with probability $1/n$.

Intuitively, this is a good algorithm because a random element will usually split the set in two reasonably well balanced subsets, hence the number of elements to consider should exponentially decrease in each iteration. For a rigorous analysis of the expected time needed by this Las Vegas algorithm, we simply introduce a "cut-off point" that defines when the split is well balanced or not. More precisely, we call an element "central" in a set, if this element is smaller and larger, respectively, than at least $1/4$ of the elements. We say that the algorithm is "in phase j" if the number of remaining elements is between $n(3/4)^{j+1}$ and $n(3/4)^j$. Clearly, our random splitter is central with probability $1/2$. It follows immediately that the expected number of splitters needed in every phase j is $2 = O(1)$. Furthermore, since $\sum_j n(3/4)^j$ is a geometric series converging to some $O(n)$, the total expected time is $O(n)$, but now with some moderate hidden constant.

A Quick but Rigorous Analysis of Quicksort

The basic version of the famous Quicksort algorithm (which we do not repeat here) works with a random splitter in every recursion step. For the sake of a simple analysis we slightly modify the algorithm, however we keep it close to the original Quicksort: We check after comparison to all other elements whether the random splitter is central, and if not, we discard it altogether and pick a new splitter. Of course, this is a certain waste of time. Hence the original Quicksort performs no worse than this modified Quicksort.

We say that a subproblem is "of type j" if the number of elements is between $n(3/4)^{j+1}$ and $n(3/4)^j$. We find a central splitter after an expected number of only 2 attempts. Thus, the expected time spent on any subproblem of type j is $O(n(3/4)^j)$. Moreover, since we accepted only central splitters, we can easily see that all subproblems of type j are pairwise disjoint, i.e., they deal with disjoint subsets of the entire set. Hence at most $(4/3)^{j+1}$ subproblems of type j can exist during the execution of the algorithm. By linearity of expectation, the expected time spent on all subproblems of size j is therefore $O(n)$. Since $O(\log n)$ types exist, the total expected time is $O(n \log n)$.