Advanced Algorithms Course. Lecture Notes. Part 1

These notes are based on Kleinberg, Tardos, *Algorithm Design* and also influenced by other material.

- The notes are an additional service. They should be considered only as concise summaries of the lectures and a mnemonic aid. It was not the intention to write another textbook!
- Many details are omitted (we suppose that you have already a good basic understanding of algorithms), and no diagrams or calculation examples are included.
- The contents follow the lectures, but they may differ from what was exactly said in class.

Approximation Algorithms

or: "My problem is NP-complete – what now?"

Load Balancing

Suppose that n jobs with processing times t_j have to be done, and every job must be assigned to one of m machines. Let T_i be the load, i.e., the total processing time of machine i. The goal is to compute an assignment that minimizes $T := \max_i T_i$.

In the context of scheduling problems where the jobs are executed one by one on the assigned machines, we call T the makespan. It is the time when all jobs are ready, and the problems asks to finish the given pile of jobs as early as possible. In the same problem with human workers rather than machines, good load balancing is simply a matter of fairness.

This problem is NP-complete already for $m = 2$ machines, as can be shown by a simple polynomial-time reduction from Subset Sum. (We assume that you already know that Subset Sum is NP-complete.) Therefore we look for algorithms that give good *approximate* solutions in polynomial time.

A natural greedy algorithm passes through the jobs and assigns every job to a machine with currently smallest load. Due to NP-completeness, this is not always optimal, and in fact, there are strikingly small explicit counterexamples: Consider $m = 2$ machines and processing times $3, 3, 2, 2, 2$. Here the optimal makespan is 6, whereas the greedy solution yields 7. Still this might be acceptable in practice.

The question arises how much the greedy solution is away from an optimal solution in the worst case. Such worst-case results give reliability: In the same way as worst-case time bounds are guarantees on the runtimes, worst-case bounds on the value of a solution guarantee a certain quality.

In order to analyze the quality of approximation algorithms, we focus on the ratio of algorithmic and optimal solution, T/T^* in our case. (Think why this is meaningful, whereas it would be rather pointless to analyze $T - T^*$.) The exact ratio T/T^* is too complicated, however a practical approach to get guarantees is to prove a lower bound on the optimal solution T^* and an upper bound on the algorithmic solution T. Then, clearly, T/T^* is at most the ratio of these bounds. (For maximization problems we can proceed similarly.)

For our Load Balancing problem, trivial lower bounds on T^* are any t_j, and $\sum t_j/m$. Now we use them to prove $T \leq 2T^*$. The idea is to

consider a machine i that finally has the maximum load $T_i = T$, and the job j assigned last to this machine i. Before job j was assigned, all machines had a load at least $T_i - t_j$ (because job j has been assigned to machine i with the currently smallest load). It follows $\sum_k T_k \geq m(T_i - t_j)$, hence $T - t_j = T_i - t_j \leq \sum_k T_k/m = \sum_l t_l/m \leq T^*$. Together this finally yields $T \leq (T - t_j) + t_j \leq T^* + T^*$.

This analysis is as good as it could be: There exist instances where T is actually nearly $2T^*$. A nasty case is many short jobs (that the greedy algorithm assigns in a balanced way) followed by one long job (that must be assigned to one machine, thereby destroying the balance). The obvious weakness of the greedy algorithm is that it considers the jobs in the given order. We can easily improve that. Intuitively, the shortest jobs should be the last in the sequence, such that they cannot stick out too much. We first sort the jobs such that $t_1 \geq \ldots \geq t_n$, then we apply the greedy algorithm. In fact, we can now prove a better approximation guarantee: $T \leq 1.5T^*$.

Again we try and find lower and upper bounds on T^* and T, respectively. As the algorithm became more clever, the (better) bounds are a little more tricky as well. First we can suppose $m < n$, otherwise the Load Balancing problem is trivial. Since at least two of the $m + 1$ longest jobs must be put on the same machine, we have $T^* \geq 2t_{m+1}$. From now on we reuse notations from the previous proof. If machine i with the maximum final load T does job j only, then the solution is optimal, since $T = t_j \leq T^*$. It remains the case that our algorithm has assigned two or more jobs to machine i. Since job j was assigned last to machine i, we have $j \geq m + 1$. (Think a little.) Hence $t_j \leq t_{m+1} \leq 0.5T^*$. And the previous analysis, which is still true, gave $T - t_j \leq T^*$. Together this yields $T \leq 1.5T^*$.

> *"I'm so proud: I have finished my puzzle in only 2 years. On the package it says 3-4 years."*

Center Selection

Let S be a set of n sites (points) in a metric space equipped with a distance function $dist$, and k a given number. The goal is to select a set C of k centers (which are also points in the same metric space) so as to minimize the maximum distance $r(C)$ of a site to the nearest center. Think of placing shops, fire stations, radio stations, etc., in a region. We call $r(C)$ the covering radius and define $r = \min_C \max_{s \in S} \min_{c \in C} dist(s, c)$, where C varies over all sets of k points.

In contrast to Load Balancing, it is already not so obvious to get an idea for a good greedy approximation algorithm. But, surprisingly, a little extra information would be extremely helpful: Assume for the moment that a little bird comes and tell us the optimal value r (but not the solution C). As we will see, this would make the problem much easier. (Of course, later we will have to drop this crazy assumption.) We say that a center c "covers" a site s, with covering radius r, if $dist(c, s) \leq r$.

Now we devise a simple greedy algorithm and analyze it at the same time. In the beginning all sites are uncovered. Consider any uncovered site s. We know that, in an optimal solution, some center c covers s. Instead of this unknown c we choose s itself as a center! By the triangle inequality, all sites covered by c are also covered by s if we enlarge the covering radius to $2r$. We repeat this step until all sites are covered (now with radius $2r$). Since the unknown optimal solution needed k centers, our greedy solution uses at most k centers as well. If some sites remain uncovered after k steps, this only indicates that our assumed r was too small. Thus we have an algorithm with approximation ratio 2, under the preliminary assumption of a known optimal covering radius r.

But in reality we do not have this extra information. So how do we determine r? A tempting idea is binary search, but since the search space consists of real numbers, it is not clear how many search steps we would need. The trick is much simpler: Instead of doing binary search we revise the above algorithm a little. Note that our preliminary algorithm may choose, in each step, an arbitrary site s whose distance to all centers already selected is larger than $2r$. And this gives the idea:

In each step, we consider the site s having the *largest* minimum distance to all centers already selected. Then the above analysis is still correct, but we do not need prior knowledge of r, since the modified algorithm does not use the value r in any way. Now we have a real algorithm for Center Selection, with approximation ratio 2.