Advanced Algorithms Course. Lecture Notes. Part 1

These notes are based on several materials, first and foremost: Kleinberg, Tardos, *Algorithm Design*.

Reading advice: Instead of just reading through the text, it is more rewarding to stop after each paragraph, rest a moment, amd think ahead: "Could I answer the current question myself? What is the next logical step? What am I wondering about?" Then continue reading ...

Approximation Algorithms

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

Actually this could be the slogan of the first parts of the course. Most likely, NP-complete problems cannot be solved exactly in polynomial time. But they are abundant, and we do need their solutions in practice. What can we do about that? One option is to *approximate* optimal solutions in polynomial time, and give up on exact optimal solutions.

We start right away with an example of the approach, and we use it to explain the general issues, step by step.

Load Balancing

Suppose that n jobs with processing times t_j have to be done, where $j = 1, \ldots, n$. Every job must be executed by one of m machines. Let T_i be the load, i.e., the total processing time, of machine i, where $i = 1, \ldots, m$. The goal is to compute an assignment that minimizes $T := \max_i T_i$.

Some clarifications: Only the t_j are given. The T_i values are part of the solution (not of the problem instance): T_i is the sum of processing times of all jobs assigned to machine i. Furthermore, every job must be done

completely on the chosen machine; it cannot be split and assigned to several machines.

Imagine a schedule where all jobs assigned to a machine are executed there, one after another. Then the objective T we want to minimize is the time when all jobs on all machines are finished. Therefore we call T the *makespan*. Clearly, we would like to finish a pile of jobs as early as possible. Moreover, if the "agents" doing the jobs are human workers rather than machines, then good load balancing is also a matter of fairness.

Bad news first: This problem is NP-complete, already for $m = 2$ machines. This can be shown by a simple polynomial-time reduction from Subset Sum. (We assume that you already know that Subset Sum is NP-complete, and that you even see how to do the reduction. If not: Read about these prerequisites again, think hard, seek help, etc. This is the level where we start from.)

Therefore we look now for algorithms that give good *approximate* solutions. But still they should need only polynomial time. We use T^* to denote the optimal makespan for an instance. (This star notation for optimal values is quite common in optimization.) In general we will not achieve $T = T^*$, but hopefully a makespan T close to T^*.

A natural greedy algorithm comes to mind: Pass through the jobs and assign every job to some machine with currently smallest load.

Due to NP-completeness, this cannnot always be optimal, and in fact, there are strikingly small counterexamples: Consider $m = 2$ machines and processing times $3, 3, 2, 2, 2$. Then obviously $T^* = 6$, whereas the greedy solution yields $T = 7$. (Do the steps of the algorithm, then you will see.) Still this might be acceptable in practice.

Did the algorithm incidentally work quite well for this example, or is it reasonable in general? Intuitively, this greedy rule should not be too bad ...

But first we have to make this vague question more precise. Perhaps like this: How far is the greedy solution away from an optimal solution in the worst case (for all instances)? Such worst-case results give reliability. In the same way as worst-case time bounds are guarantees on the running times, worst-case bounds on the value of a solution guarantee a certain quality of the output.

But still the question is too imprecise in this form. What measure of the distance of solutioos shall we adopt?

What about $T - T^*$? A moment of thinking reveals that analyzing the difference is pointless: If we scale an instance by multiplying all processing times by a factor c, then also the makespan of any algorithm is multiplied by c, but the problem is not changed at all. Most importantly, the difference $T - T^*$ is unbounded, i.e., the maximum over all instances would be infinite.

What about T/T^*? The ratio is not changed by scaling (more formally, it is an invariant), and there is a chance that the maximum over all instances is finite. These properties make the *approximation ratio* T/T^* a suitable measure of quality of an algoriithm.

But how can we actually analyze the approximation ratio of a given algorithm?

The exact ratio T/T^*, as a function of the instance, is too complicated and depends on many irrelevant small details of the instances. A more practical approach to getting guarantees is to try and prove some good upper *bound* on T/T^*.

For an upper bound on T/T^* it suffices to find some lower bound on T^* and some upper bound on T. Then, clearly, T/T^* is at most the ratio of these bounds.

But beware: The situation is not symmetric. A lower bound on T^* deals with optimal *solutions*. This is a property of the *problem* and has nothing to do with any algorithm. As opposed to this, an upper bound on T deals with the output of the *algorithm*.

For analyzing our Load Balancing problem, we start with T^*. (The order is arbitrary. This is just the easier part.) Two obvious lower bounds, directly inferred from the problem, are $T^* \geq \max t_j$ and $T^* \geq \sum_{j=1}^{n} t_j/m$, that is, the longest job and the average load. (Why are they lower bounds?)

Any upper bound on T has to take into account how the algorithm actually works. We could ask what specifically makes the output bad. That is, we look at the job j that ends last in the schedule, because this job is "responsible" for the makespan. Moreover, we look at the machine i that has the highest load T_i in the end.

As a side remark, it would be a mistake to simply assume $j = n$. The job that ends last is not necessarily the last job in the given sequence, because it may be followed by some short jobs which do not further increase the makespan.

Continuing our analysis: Why has the algorithm finally assigned job j to machine i?

Because machine i had a minimum load, just before job j has been added. And this load was $T_i - t_j = T - t_j$. It follows that *every* machine k has a load $T_k \geq T - t_j$. Summation over all machines yields $\sum_{k=1}^{m} T_k \geq m(T - t_j)$. Hence $T - t_j \leq \sum_{k=1}^{m} T_k/m = \sum_{l=1}^{m} t_l/m \leq T^*$ due to the "averaging" lower bound. Since also $T^* \geq t_j$, this finally yields $T \leq T^* + t_j \leq 2T^*$. That is, we have proved an approximation ratio $T/T^* \leq 2$.

Thus we have obtained an upper bound on the approximation ratio. This is great but could also be disappointing: In this example, the solution could be twice as expensive as the optimum. Can the algorithm be improved? Or is the algorithm actually better, and only the analysis pretends too generous bounds? In hindsight, the analysis used only very simple algebra and combinatorics – maybe it missed some opportunities for tighter bounds?

It turns out that this analysis was as good as it could be: There exist instances where T is actually nearly $2T^*$. A nasty case consists of 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). Using a certain number and length of short jobs, one can enforce a ratio arbitrarily close to 2; we skip these details and only stress the idea.

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

Definitions

After this introductory example we collect some general definitions.

A minimization (maximization) problem is a computational problem that asks for a solution that minimizes (maximizes) some objective value. A c-approximation algorithm for a minimization problem is an algorithm that runs in polynomial time and outputs a solution with an objective value being at most c times as large as the minimum value. The approximation ratio $c > 1$ can be a constant, but it may also be a function of, e.g., the instance size. For maximization problems, the definitions are similar, but the objective value must be at least c times as large as the maximum value, where $c < 1$.

And a linguistic remark: Please avoid the phrase "more optimal" in mathematical texts. Here the word "optimal" means already "the best" - so what should be better?

Load Balancing 1.5

Instances that need an approximation ratio nearly 2 point out a weakness of the greedy algorithm above: it considers the jobs in the given order. We can easily improve that. Intuitively, the shortest jobs should be assigned last, such that they cannot stick out too much. We first sort and re-index the jobs such that $t_1 \geq \ldots \geq t_n$, then we apply the original greedy algorithm.

Can we prove a better approximation ratio? As the algorithm has become a bit smarter, the (better) bounds are a little more tricky as well.

First of all, the Load Balancing problem with $m \geq n$ is trivial: put every job on a different machine and obtain $T = T^*$. Thus we can suppose $m < n$ henceforth.

Since now $n \geq m+1$, at least two of the $m+1$ longest jobs must be put on the same machine. This yields $T^* \geq 2t_{m+1}$, which is stronger than the one-job lower bound we had before. This is already a good sign.

Moreover, we can reuse notations and results from the previous analysis, since it was done for arbitrary sequences and is therefore still valid for the special case of a sorted sequence.

If the machine i with the maximum final load T does job j only, then the solution is optimal, since $T = t_j \leq T^*$, actually $T = T_j$. Hence we can suppose that machine i executes two or more jobs. Since the m longest jobs are assigned first, to m different machines, but job j was assigned last to machine i, we have $j \geq m+1$. Hence $t_j \leq t_{m+1} \leq 0.5T^*$. As the previous analysis gave $T \leq T^* + t_j$, this yields $T \leq 1.5T^*$, a significant improvement.

We conclude with some general comments.

One can try and find more lower bounds, and for every instance take the maximum of them. But how do we know where to stop? What lower bounds do we need for a good analysis? Unfortunately, there is no general answer. In the above example, the two lower bounds turned out to be sufficient for a tight analysis. But we see this only afterwards. Choosing bounds is a problem-specific trial-and-error business, like many things in this field.

Also, the example should not give the impression that approximation

ratios are always obtained as ratios of some global lower and upper bounds. There are other methods, like pairing up the elements of the two solutions and comparing their costs.

Finally, you may have missed a statement on the time complexity. The above algorithms clearly have low polynomial complexity. We have skipped the time analysis, but not because it is unimportant. It is simply not the focus here. We assume that you have already seen plenty of time estimates previously.

Center Selection

Next we discuss an approximation algorithm for a problem of completely different nature, to get an impression of the diversity of the topic.

A metric space is any set of points equipped with a distance function d that is symmetric and satisfies the triangle inequality. Let S be a set of n sites (points) from a metric space, 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 of a site to the nearest center. To help imagination, one may think of points in the plane with the usual Euclidean distance, but the problem is defined for arbitrary metric spaces. We can assume $|S| > k$, since otherwise the problem is trivial.

As a motivation: We may want to place facilities (shops or fire stations or radio stations, etc.) in a region, such that no inhabitant is far away from the next facility.

The objective can be written as

$$\min_{|C|=k} \max_{s \in S} \min_{c \in C} d(s, c).$$

This looks a bit bulky. But by defining the disk with center c and radius r to be the set of all points x with $d(c, x) \leq r$, the Center Selection problem can be rephrased: Find a set of k disks of minimum (equal) radius r that together cover S.

The following greedy algorithm selects, in k steps, some set $C \subset S$. Start with $C := \{s\}$, where $s \in S$ is an arbitrary site. Then do the following $k-1$ times: Take a site $s \in S$ being farthest from its nearest center in C, and put it in C. More formally, pick some $s \in S$ with maximum $\min_{c \in C} d(s, c)$ and do $C := C \cup \{s\}$.

It obviously runs in polynomial time, and it is a 2-approximation algorithm. More precisely, we can show: Let C be the set of centers produced by the algorithm. If a solution with radius r exists, then the k disks with centers in C and radius $2r$ cover S as well.

For the proof, let $\mathcal{D}$ denote a set of disks of radius r that together c over S, and let U denote the union of all disks with centers in C and radius $2r$. For the situation after any number of steps of the algorithm we claim the following: Either $S \subseteq U$, or there exists some $D \in \mathcal{D}$ with currently $D \not\subseteq U$, but with $D \subseteq U$ after the next step of the algorithm.

Since $|\mathcal{D}| = k$, this claim implies that we reach $S \subseteq U$ after at most k steps. It remains to prove the claim. So assume that $S \subseteq U$ is false. That is, some site has a distance larger than $2r$ from its nearest center in C. Since the algorithm chooses some site s with maximum distance to its nearest center, this distance is also larger than $2r$. It follows $s \notin U$. But there exists some $D \in \mathcal{D}$ with $s \in D$. This yields $D \not\subseteq U$. Now the algorithm adds the disk D' with center s and radius $2r$ to U. Furthermore, the triangle inequality yields $D \subseteq D'$. Hence, after this step we have $D \subseteq U$, as claimed.

The proof we presented here is shorter and goes more straight to the goal than the one in the textbook.

Simple Vertex Cover Approximation

A vertex cover in an undirected graph $G = (V, E)$ is a node set $C \subseteq V$ being incident to all edges of E. That is, for every edge $e \in E$, at least one node of e must be in C. Finding a vertex cover of minimum size is a classsic NP-complete problem. But we can very easily solve it within an approximation ratio 2.

The this end we use the concept of a matching. A matching in a graph is an edge set $M \subseteq E$ whose edges are pairwise disjoint, i.e., no two edges in M share a node. First we determine a matching M that is maximal in the sense that no further edges of E can be added to M. (This does not necessarily mean that M is a matching of maximum size.)

A maximum-size matching can also be computed in polynomial time, but this is complicated. Here we only need a maximal matching in the above weaker sense, and we can obtain it in a greedy way: Start with $M := \emptyset$ and successively add an edge to M that is disjoint to all others, as long as possible. Hence the final M is a maximal matching.

Let C simply be the set of all nodes in the edges of M. Then C is a vertex cover: If not, then some edge e exists where both nodes are not in M, which contradicts the maximality of M.

Furthermore, every vertex cover contains at least one node from every edge in M, by the definition of vertex covers. It follows $|C^*| \geq |M|$ even for the smallest vertex cover C^*. Hence we have $|C| = 2|M| \leq 2|C^*|$. Simple, isn't it?