Lecture 5: Introduction to Approximation Algorithms

Many important computational problems are difficult to solve optimally. In fact, many of those problems are *NP-hard*[1], which means that no polynomial-time algorithm exists that solves the problem optimally unless P=NP. A well-known example is the *Euclidean traveling salesman problem (Euclidean TSP)*: given a set of points in the plane, find a shortest tour that visits all the points. Another famous NP-hard problem is *independent set*: given a graph $G = (V, E)$, find a maximum-size independent set $V^* \subset V$. (A subset is independent if no two vertices in the subset are connected by an edge.)

What can we do when faced with such difficult problems, for which we cannot expect to find polynomial-time algorithms? Unless the input size is really small, an algorithm with exponential running time is not useful. We therefore have to give up on the requirement that we always solve the problem optimally, and settle for a solution close to optimal. Ideally, we would like to have a guarantee on how close to optimal the solution is. For example, we would like to have an algorithm for Euclidean TSP that always produces a tour whose length is at most a factor ρ times the minimum length of a tour, for a (hopefully small) value of ρ. We call an algorithm producing a solution that is guaranteed to be within some factor of the optimum an *approximation algorithm*. This is in contrast to *heuristics*, which may produce good solutions but do not come with a guarantee on the quality of their solution.

Basic terminology. From now on we will use $\textsc{opt}(I)$ to denote the value of an optimal solution to the problem under consideration for input I. For instance, when we study TSP then $\textsc{opt}(P)$ will denote the length of a shortest tour on a point set P, and when we study the independent-set problem then $\textsc{opt}(G)$ will denote the maximum size of any independent set of the input graph G. When no confusion can arise we will sometimes simply write $\textsc{opt}$ instead of $\textsc{opt}(I)$.

A *minimization problem* is a problem where we want to find a solution with minimum value; TSP is an example of a minimization problem. An algorithm for a minimization problem is called a *ρ-approximation algorithm*, for some $\rho > 1$, if the algorithm produces for any input I a solution whose value is at most $\rho \cdot \textsc{opt}(I)$. A *maximization problem* is a problem where we want to find a solution with maximum value; independent set is an example of a maximization problem. An algorithm for a maximization problem is called a *ρ-approximation algorithm*, for some $\rho < 1$, if the algorithm produces for any input I a solution whose value is at least $\rho \cdot \textsc{opt}(I)$. The factor ρ is called the *approximation factor* (or: *approximation ratio*) of the algorithm.[2]

The importance of lower bounds. It may seem strange that it is possible to prove that an algorithm is a ρ-approximation algorithm: how can we prove that an algorithm always produces a solution that is within a factor ρ of $\textsc{opt}$ when we do not know $\textsc{opt}$? The crucial observation is that, even though we do not know $\textsc{opt}$, we can often derive a *lower bound* (or, in the case of maximization problems: an upper bound) on $\textsc{opt}$. If we can then show that our algorithm always produces a solution whose value is at most a factor ρ from the lower bound, then the algorithm is also within a factor ρ from $\textsc{opt}$. Thus finding good lower bounds on

[1] Chapter 36 of [CLRS] gives an introduction to the theory of NP-hardness.

[2] In some texts the approximation factor ρ is required to be always greater than 1. For a maximization factor the solution should then have a value at least $(1/\rho) \cdot \textsc{opt}$.

OPT is an important step in the analysis of an approximation algorithm. In fact, the search for a good lower bound often leads to ideas on how to design a good approximation algorithm. This is something that we will see many times in the coming lectures.

5.1 Load balancing

Suppose we are given a collection of n jobs that must be executed. To execute the jobs we have m identical machines, $M_1, \ldots, M_m$, available. Executing job j on any of the machines takes time t_j, where $t_j > 0$. Our goal is to assign the jobs to the machines in such a way that the so-called *makespan*, the time until all jobs are finished, is as small as possible. Thus we want to spread the jobs over the machines as evenly as possible. Hence, we call this problem LOAD BALANCING.

Let's denote the collection of jobs assigned to machine M_i by $A(i)$. Then the *load* T_i of machine M_i—the total time for which M_i is busy—is given by

$$T_i = \sum_{j \in A(i)} t_j,$$

and the makespan of the assignment equals $\max_{1 \leqslant i \leqslant m} T_i$. The LOAD BALANCING problem is to find an assignment of jobs to machines that minimizes the makespan, where each job is assigned to a single machine. (We cannot, for instance, execute part of a job on one machine and the rest of the job on a different machine.) LOAD BALANCING is NP-hard.

Our first approximation algorithm for LOAD BALANCING is a straightforward greedy algorithm: we consider the jobs one by one and assign each job to the machine whose current load is smallest.

Algorithm *Greedy-Scheduling*$(t_1, \ldots, t_n, m)$
1. Initialize $T_i \leftarrow 0$ and $A(i) \leftarrow \emptyset$ for $1 \leqslant i \leqslant m$.
2. **for** $j \leftarrow 1$ **to** n
3. **do** $\triangleright$ Assign job j to the machine M_k of minimum load
4. Find a k such that $T_k = \min_{1 \leqslant i \leqslant m} T_i$
5. $A(k) \leftarrow A(k) \cup \{j\}$; $T_k \leftarrow T_k + t_j$

This algorithm clearly assigns each job to one of the m available machines. Moreover, it runs in polynomial time. In fact, if we maintain the loads T_i in a min-heap then we can find the machine k with minimum load in $O(1)$ time and update T_k in $O(\log m)$ time. This way the entire algorithm can be made to run in $O(n \log m)$ time. The main question is how good the assignment is. Does it give an assignment whose makespan is close to OPT? The answer is yes. To prove this we need a lower bound on OPT, and then we must argue that the makespan of the assignment produced by the algorithm is not much more than this lower bound.

There are two very simple observations that give a lower bound. First of all, the best one could hope for is that it is possible to spread the jobs perfectly over the machines so that each machine has the same load, namely $\sum_{1 \leqslant j \leqslant n} t_j / m$. In many cases this already provides a pretty good lower bound. When there is one very large job and all other jobs have processing time close to zero, however, then the upper bound is weak. In that case the trivial lower bound of $\max_{1 \leqslant j \leqslant n} t_j$ will be stronger. To summarize, we have

Lemma 5.1 $\text{OPT} \geqslant \max\left(\frac{1}{m} \sum_{1 \leqslant j \leqslant n} t_j \ , \ \max_{1 \leqslant j \leqslant n} t_j \right)$.

Let's define LB := max ($\frac{1}{m}\sum_{1\leqslant j\leqslant n} t_j$, $\max_{1\leqslant j\leqslant n} t_j$) to be the lower bound provided by Lemma 5.1. With this lower bound in hand we can prove that our simple greedy algorithm gives a 2-approximation.

Theorem 5.2 *Algorithm Greedy-Scheduling is a 2-approximation algorithm.*

Proof. We must prove that *Greedy-Scheduling* always produces an assignment of jobs to machines such that the makespan T satisfies $T \leqslant 2 \cdot \text{OPT}$. Consider an input $t_1, \ldots, t_n, m$. Let M_{i^*} be a machine determining the makespan of the assignment produced by the algorithm, that is, a machine such that at the end of the algorithm we have $T_{i^*} = \max_{1\leqslant i\leqslant m} T_i$. Let j^* be the last job assigned to M_{i^*}. The crucial property of our greedy algorithm is that at the time job j^* is assigned to M_{i^*}, machine M_{i^*} is a machine with the smallest load among all the machines. So if T'_i denotes the load of machine M_i just before job j^* is assigned, then $T'_{i^*} \leqslant T'_i$ for all $1 \leqslant i \leqslant m$. It follows that

$$m \cdot T'_{i^*} \;\leqslant\; \sum_{1\leqslant i\leqslant m} T'_i \;=\; \sum_{1\leqslant j<j^*} t_j \;<\; \sum_{1\leqslant j\leqslant n} t_j \;\leqslant\; m \cdot \text{LB}.$$

Hence, $T'_{i^*} < \text{LB}$ and we can derive

$$\begin{aligned} T_{i^*} &= t_{j^*} + T'_{i^*} \\ &\leqslant t_{j^*} + \text{LB} \\ &\leqslant \max_{1\leqslant j\leqslant n} t_j + \text{LB} \\ &\leqslant 2 \cdot \text{LB} \\ &\leqslant 2 \cdot \text{OPT} \qquad \text{(by Lemma 5.1)} \end{aligned}$$

□

So this simple greedy algorithm is never more than a factor 2 from optimal. Can we do better? There are several strategies possible to arrive at a better approximation factor. One possibility could be to see if we can improve the analysis of *Greedy-Scheduling*. Perhaps we might be able to show that the approximation factor is in fact at most $c \cdot \text{LB}$ for some $c < 2$. Another way to improve the analysis might be to use a stronger lower bound than the one provided by Lemma 5.1. (Note that if there are instances where LB = OPT/2 then an analysis based on this lower bound cannot yield a better approximation ratio than 2.)

It is, indeed, possible to prove a better approximation factor for the greedy algorithm described above: a more careful analysis shows that the approximation factor is in fact $(2 - \frac{1}{m})$, where m is the number of machines. This is tight for the given algorithm: for any m there are inputs such that *Greedy-Scheduling* produces an assignment of makespan $(2 - \frac{1}{m}) \cdot \text{OPT}$. Thus the approximation ratio is fairly close to 2, especially when m is large. So if we want to get an approximation ratio better than $(2 - \frac{1}{m})$, then we have to design a better algorithm.

A weak point of our greedy algorithm is the following. Suppose we first have a large number of small jobs and then finally a single very large job. Our algorithm will first spread the small jobs evenly over all machines and then add the large job to one of these machines. It would have been better, however, to give the large job its own machine and spread the small jobs over the remaining machines. Note that our algorithm would have produced this assignment if the

large job would have been handled first. This observation suggest the following adaptation of the greedy algorithm: we first sort the jobs according to decreasing processing times, and then run *Greedy-Scheduling*. We call the new algorithm *Ordered-Scheduling*.

Does the new algorithm really have a better approximation ratio? The answer is yes. However, the lower bound provided by Lemma 5.1 is not sufficient to prove this; we also need the following lower bound.

Lemma 5.3 *Consider a set of n jobs with processing times $t_1, \ldots, t_n$ that have to be scheduled on m machines, where $t_1 \geqslant t_2 \geqslant \cdots \geqslant t_n$. If $n > m$, then* $\textsc{opt} \geqslant t_m + t_{m+1}$.

Proof. Since there are m machines, at least two of the jobs $1, \ldots, m+1$, say jobs j and j', have to be scheduled on the same machine. Hence, the load of that machine is $t_j + t_{j'}$, which is at least $t_m + t_{m+1}$ since the jobs are sorted by processing times. □

Theorem 5.4 *Algorithm Ordered-Scheduling is a (3/2)-approximation algorithm.*

Proof. The proof is very similar to the proof of Theorem 5.2. Again we consider a machine M_{i^*} that has the maximum load, and we consider the last job j^* scheduled on M_{i^*}. If $j^* \leqslant m$, then j^* is the only job scheduled on M_{i^*}—this is true because the greedy algorithm schedules the first m jobs on different machines. Hence, our algorithm is optimal in this case. Now consider the case $j^* > m$. As in the proof of Theorem 5.2 we can derive

$$T_{i^*} \quleq t_{j^*} + \tfrac{1}{m} \textstyle\sum_{1 \leqslant i \leqslant n} t_i.$$

The second term can be bounded as before using Lemma 5.1:

$$\tfrac{1}{m} \textstyle\sum_{1 \leqslant i \leqslant n} t_i \quleq \max\left(\max_{1 \leqslant j \leqslant n} t_j \;,\; \tfrac{1}{m} \textstyle\sum_{1 \leqslant i \leqslant n} t_i \right) \quleq \textsc{opt}.$$

For the first term we use that $j^* > m$. Since the jobs are ordered by processing time we have $t_{j^*} \leqslant t_{m+1} \leqslant t_m$. We can therefore use Lemma 5.3 to get

$$t_{j^*} \quleq (t_m + t_{m+1})/2 \quleq \textsc{opt}/2.$$

Hence, the total load on M_{i^*} is at most $(3/2) \cdot \textsc{opt}$. □