Approximation Algorithms and Hardness of Approximation January 21, 2013

Lecture 2

Lecturer: Ola Svensson *Scribes: Abdallah Elguindy, Ákos Lukovics*

1 Greedy Algorithms

In this lecture we study greedy approximation algorithms, algorithms finding a solution in a number of locally optimal steps. Some of their advantages are:

- easy-to-implement
- fast
- first try at tackling a problem
- optimal in some cases
- can give deeper insight on the structure of the problem in question and aid us in designing better algorithms

We will demonstrate the last point on the example of the Identical Parallel Machine Scheduling problem.

1.1 Identical Parallel Machine Scheduling Problem

In the previous lecture we saw the "List-Scheduling algorithm" (LS; see algorithm 1), a 2-approximation for this problem. We will try to improve its approximation-ratio by finding an infinite family of instances with an approximation ratio of 2 and identifying the bottlenecks.

input : n jobs with processing time $\{p_1, p_2, \ldots, p_n\}$ and m machines.
output: The assignment of jobs to machines.

1. Order the jobs arbitrarily.
2. In this order, assign each job to the machine that currently has the least work assigned to it.

Algorithm 1: List-Scheduling algorithm

Instance: we are given m machines, $m(m-1)$ jobs with processing times 1, and a single job with processing time m. Let LS schedule the 1 jobs first, and the m job last. The makespan of this schedule is clearly $2m-1$. OPT schedules the 1 jobs on $m-1$ machines, and the m job on the remaining one. The makespan of OPT is m. Thus, the approximation ratio approaches 2, as m grows towards infinity.

input : n jobs with processing time $\{p_1, p_2, \ldots, p_n\}$ and m machines.
output: The assignment of jobs to machines.

1. Order the jobs in descending order according to their processing times.
2. In this order, assign each job to the machine that currently has the least work assigned to it.

Algorithm 2: Largest Processing Time Rule algorithm

The issue with LS is with the handling of the long jobs. Using this knowledge we design a new algorithm, the "Largest Processing Time Rule algorithm" (LPT; see algorithm 2). It schedules in order from the largest to the smallest job, instead of using an arbitrary order. We claim that LPT is a $4/3$-approximation.

Theorem 1 *LPT is a 4/3-approximation.*

Proof Order the n jobs such that $p_1 \leq p_2 \leq \cdots \leq p_n$. Without loss of generality, we assume job n finishes last. If it did not finish last, removing it would not change the schedule of LPT, but only decrease OPT. Thus, if the bound holds for the case where n finishes last, it must also hold for cases where it does not, i.e., OPT is worse: $LPT \leq 4/3 \cdot OPT \wedge OPT' \geq OPT \Rightarrow LPT \leq 4/3 \cdot OPT'$. So, the makespan is

$$START_n + p_n \leq \frac{1}{m} \sum_{i \neq n} p_i + p_n \leq OPT + p_n.$$

Using the same technique as in the case of LS gets us an approximation ratio of 2. Instead of refining our bounds on OPT, we will try to bound p_n. Clearly, if $p_n \leq OPT/3$, then LPT is 4/3-OPT. We only have to show that larger p_n values don't violate our bounds. We claim if $p_n > OPT/3$, then LPT gives an optimal schedule.

Claim 2 *If $p_n > OPT/3$, LPT is optimal.*

Proof We prove this by contradiction. Assume the l-th job completes after time OPT. We consider the schedule before the l-th job. It is easy to see that in this schedule each machine must be assigned at least one job and at most two (since otherwise the length of the schedule would already be bigger than OPT). Now order the machines and select i such that the machines M_1 through M_i have a single job assigned to them and machines M_{i+1} through M_m have two. Let's call the jobs on the first i machines long jobs, the ones on the remaining ones short jobs (l is also considered a short job). LPT would schedule l so that it finishes after OPT. This means l can't be scheduled on a machine with a long job, and since l is smaller than any job that is already scheduled (by the rules of LPT) no short or long job can be scheduled with another long job. Hence, to not violate OPT any schedule needs i machines to schedule the long jobs and the short jobs must be scheduled on the remaining machines ($m - i$ many). However, as there are $2(m - i) + 1$ short jobs and each machine can execute at most two jobs, this is clearly impossible and we conclude that there is no schedule better than LPT and our claim holds. ■

We have shown that LPT returns either an optimal schedule or one with length at most 4/3OPT, thus LPT is indeed a 4/3-approximation algorithm. ■

The 4/3 bound is tight, an infinite family of instances showing this is given below.

Instance: we are given m machines, and $2m+1$ jobs. There are three jobs with processing time m, and 2 jobs with processing times $m+1, m+2, \ldots, 2m-1$ each. In case of LPT, all but one of the machines get two jobs with a total processing time of $3m - 1$, and a single machine gets three jobs with a total of $4m - 1$ processing time. Thus, the makespan is $4m - 1$. OPT schedules the three m jobs on a single machine, and the remaining jobs on the remaining $m-1$ machines, such that each of those machines get jobs with a total processing time of $3m$, thus the makespan of OPT is $3m$. As m grows towards infinity the approximation ratio approaches 4/3.

There is an even better approximation algorithm, a so-called PTAS[1], for this problem, but it is not covered by this lecture.

1.2 Set-Cover Problem

The next problem we consider is the Set-Cover Problem. We are given a universe of n elements $U = \{e_1, e_2, \cdots, e_n\}$, a collection of subsets of the universe $T = \{s_1, s_2, \cdots, s_k\}$, and a cost function

[1]Polynomial Time Approximation Scheme: For any $\epsilon > 0$ we can find a $(1 + \epsilon)$-approximate solution in time that is polynomial for any fixed ϵ (for example, the running time may be $O(n^{1/\epsilon})$).

$c : T \to \mathbb{R}$ mapping each subset to a real-valued cost. The task is to find a minimum cost subset of collections containing all elements.

Example: assume we have 6 elements $e_1, e_2, \cdots, e_6$ and 4 sets $S_1 = \{e_1, e_2, e_3\}, S_2 = \{e_1, e_4\}, S_3 = \{e_2, e_5\}, S_4 = \{e_3, e_6\}$, and a cost function $c(S_i) = 2$. The optimal solution for this instance is the set $\{S_2, S_3, S_4\}$ with a total cost of 6. Note: $\{S_1, S_2, S_3, S_4\}$ is also a feasible (however, not optimal) solution.
We discuss a greedy approximation algorithm for this problem. The algorithm greedily chooses the set that covers most elements per unit cost, following the slogan "most bang for the buck".

input : A collection of subsets $T = \{s_1, s_2, \cdots, s_k\}$ of the set $U = \{e_1, e_2, \cdots, e_n\}$.
output: A subcollection of subsets that covers U.

1. $C \leftarrow \emptyset$.
2. While $C \neq U$
   a. Find the set S that minimizes $\frac{c(S)}{|S-C|}$.
   b. Set $price(e) = \frac{c(S)}{|S-C|}$ for every $e \in S - C$.
   c. $C \leftarrow C \cup S$.

Algorithm 3: Greedy algorithm ("most bang for the buck")

Example: in the example above, initially, $\frac{c(S_1)}{|S_1-C|} = 2/3$ and $\frac{c(S_2)}{|S_2-C|} = \frac{c(S_3)}{|S_3-C|} = \frac{c(S_4)}{|S_4-C|} = 1$. Thus, S_1 is chosen. In the second iteration, $\frac{c(S_2)}{|S_2-C|} = \frac{c(S_3)}{|S_3-C|} = \frac{c(S_4)}{|S_4-C|} = 2$. The algorithm then chooses S_2, S_3 and S_4 in some arbitrary order.

Theorem 3 *The greedy algorithm is an H_n-approximation algorithm (where $H_n = 1+1/2+1/3+\ldots+1/n \approx \ln n + O(1)$).*

Proof Consider the elements in the order $e_1, e_2, \ldots, e_n$ they were chosen by the algorithm, breaking ties arbitrarily. Since the cost of a chosen set is distributed among the newly covered elements, we have that $c(Sol) = \sum_{k=1}^{n} price(e_k)$. We prove the bound $price(e_k) \leq \frac{OPT}{n-k+1}$ and this directly proves the theorem since $c(Sol) = \sum_{k=1}^{n} price(e_k) \leq \sum_{k=1}^{n} \frac{OPT}{n-k+1} = H_n\, OPT$.
To prove that $price(e_k)$ is at most $OPT/(n-k+1)$, it is enough to notice that before covering the kth element, at most $k-1$ elements have been covered. Therefore, at most $n-k+1$ remain uncovered. These uncovered elements can be covered at cost at most OPT (by just picking an optimal covering). Therefore, there are subsets not chosen with cost per element at most $OPT/(n-k+1)$, since the algorithm chooses the subset that minimizes the cost per element, the price is at most $OPT/(n-k+1)$. ■

Tight example: consider the set $U = \{e_1, e_2, \ldots, e_n\}$ and the collection of subsets $\{\{e_i\} | i = 1, 2, \ldots, n\} \cup \{U\}$. The cost of the subset U is $1+\epsilon$ while the cost of $\{e_i\}$ is $\frac{1}{n-i+1}$ (subset $\{e_1\}$ costs $1/n$, e_2 costs $1/(n-1)$, ...etc.). The algorithm will choose all singletons with a total cost of H_n while the optimal is choosing only the subset U with cost $1+\epsilon$.
Note: the problem cannot be approximated better than by a factor $O(\log n)$ unless $NP \subseteq DTIME(n^{\log \log n})$.

Theorem 4 *If for each subset in the collection $|S| \leq t$ then the greedy algorithm is an H_t-approximation algorithm.*

1.3 K-center Problem

The last problem we study is the problem of placing k centers to minimize the maximum distance of customers to their nearest center. The problem is defined as given a set of n points V and a metric d_{ij} on the points. We need to find a subset $S \subset V$ such that $|S| = k$ and the maximum distance from any point in V to the closest point to it in S is minimized.
The greedy algorithm "furthest away" just iteratively chooses points that are furthest away from already chosen points.

input : Set of points $V = \{p_1, p_2, \ldots, p_n\}$
output: A subset S of V of size k

1. Pick any arbitrary point p, set $S \leftarrow \{p\}$.
2. While $|S| < k$, find the point whose minimum distance to S is maximum and add it to S.

Algorithm 4: Furthest away algorithm

Theorem 5 *"Furthest away" is a 2-approximation algorithm for the k-center problem.*

Proof Consider an optimal solution $O = \{o_1, \ldots, o_k\}$. This set of points defines a Voronoi diagram. If each cell has one point from S, then a point p in any cell k is at most OPT away from o_k. Moreover, s_k (the point in S lying in cell k) is also at most OPT away from o_k. Therefore, by the triangle inequality, p is at most $2 \cdot OPT$ from s_k. Otherwise, there must be a cell k with two points from S (call them s and s', included in S in that order, without loss of generality). Since s' was chosen to be the furthest point from the previously chosen points, then the maximum distance of any point to a point in S is at most $d(o_k, s) + d(o_k, s')$ which is again at most $2 \cdot OPT$. This proves that the algorithm is indeed a 2-approximation algorithm. ■