Lecture 1

Allan Borodin
(Recorded by Alexandra Goultiaeva)

Sept. 15, 2010

1 Introduction

This is a graduate level course on Algorithm Design, Analysis and Theory. We used to have many specialized courses, such as advanced graph theory, approximation algorithms, linear programming etc. This is the first graduate "foundational" course in algorithms, attempting to give a broad overview of the material.

1.1 Focus of the course

"Algorithms" is a very general topic, which runs throughout computer science. Obviously, not all of it can be covered in a single course.

In this course, we will not emphasize specific problems (such as algorithms for a particular type of graph), but will rather try to focus on general methods: meta algorithms, algorithmic paradigms.

The focus of this course is in:

- Discrete algorithms (as opposed to, for eg, numerical methods)
- Finite inputs, finite outputs, terminating computation (as opposed to, for eg, operating systems with continuous input and output stream)
- Sequential RAM model; centralized computation (not distributed, parallel, quantum, biological, etc)

1.2 Material

An undergraduate courses in algorithms, such as CSC373 and its analogues, usually focus on search and optimization. The following are the topics usually covered in such a course:

- greedy algorithms
- divide and conquer
- dynamic programming
- local search

- flow algorithms
- linear programming

Othogonal to these paradigms are general ideas such as reduction, randomization, scaling, embedding, etc.

Note that all of the paradigms, except perhaps the divide-and-conquer paradigm, are mostly concerned with search and optimization. We will do a lot of search and optimization as well, but might also touch upon other problem domains.

The course will start with an overview of the above undergraduate content (using some less standard examples) and then will go into the more advanced material, which might include: a deeper view of linear programming; primal/dual algorithms; streaming algorithms; social networks; algorithmic game theory.

Usually, the undergraduate course introduces a paradigm, and then provides examples to demonstrate it. Here, we will initially go a different route: we will start with a problem, and then look at how different paradigms can be used to solve it. Our first problem is the Makespan Scheduling Problem.

2 Makespan Scheduling Problem

Problem. *Given a set of jobs $\mathcal{I} = \{J_1, J_2, \cdots, J_n\}$, where each job J_i is represented by a "load" or processing time p_i, and the number of available machines m, assign n jobs to the m machines so as to minimize the maximum load on any machine. The output is a mapping $\tau : \{1, 2, \cdots, n\} \to \{1, 2, \cdots, m\}$.*

The optimal solution will have the cost:

$$\min_{\tau} \max_{i} \sum_{j:\tau(j)=i} p_j$$

Note: Usually the load p_i is considered to be the processing time for the job. In that case, the objective is to simply minimize the latest finishing time. However, sometimes people make the distinction between the load and the processing time, so that additional timing constraints can be introduced.

Problem variants The above problem definition is for the Makespan Scheduling problem with a *uniform machine model*. Other models include the following:

> **Related machines model.** Each machine i is operated as speed s_i. In this case, the optimal cost is
>
> $$\min_{\tau} \max_{i} \sum_{j:\tau(j)=i} p_j / s_i$$
>
> **Restricted machines model.** Each job is represented by a pair $(p_j, \boldsymbol{S}_j)$, where $\boldsymbol{S}_j \subseteq \{1, 2, \cdots, m\}$ and represents the set of machines on which job i can execute. We restrict the solution to only include mappings τ such that for all relevant i, $\tau(i) \in \boldsymbol{s}_i$.

Unrelated machines model. Here each job might have a different load depending upon which machine it is executed. Let p_{ji} be the cost of the job j executing on machine i. Then the optimal cost is

$$\min_{\tau} \max_{i} \sum_{j:\tau(j)=i} p_{ji}$$

The problem was originally introduced in [2] and [1], before results on NP-completeness. This problem is obviously NP-complete: it can be solved by brute force in exponential time, and for the case of two machines it reduces to a well-known NP-complete problem SUBSET-SUM.

2.1 The "Natural" Online Greedy Algorithm

The following is a greedy algorithm to solve this problem.

```
1 Take the inputs p_1, p_2, ··· , p_n "in the order given".
2 for i = 1, ··· , n do
3     schedule τ(J_i) in the currently least loaded machine
4 end
```

2.1.1 Approximation ratio

How good is the algorithm? For a minimization problem, we say that an algorithm ALG is a *c-approximation* if for all inputs I,

$$\text{Cost}(\text{ALG}[I]) \leq c \times \text{Cost}(\text{OPT}[I])$$

Where OPT is the optimal algorithm for solving the problem, and Cost(ALG[I]) is the cost of the solution provided by algorithm ALG on input I. Note: since it's impossible to do better than the optimal, c will be ≥ 1.

For maximization problems, there are unfortunately two ways of defining a c-approximation. We say that for all inputs I:

$$\text{Profit}(\text{ALG}[I]) \geq \tilde{c} \times \text{Profit}(\text{OPT}[I])$$

or

$$\text{Profit}(\text{OPT}[I]) \leq c \times \text{Profit}(\text{ALG}[I])$$

Note that in the first case $\tilde{c} \leq 1$, and in the second $c \geq 1$.

There is no standard notation, so both definitions might be used, and should be distinguished by context.

Note 1: There is another type of approximation ratio, called *asymptotic approximation*, which allows a "slow growing" overhead:

$$\forall I \text{Cost}(\text{ALG}[I]) \leq c \times \text{Cost}(\text{OPT}[I]) + o(\text{OPT}[I])$$

For online algorithms the analogue of the asymptotic approximation ratio is called the competitive ratio. Unless it is specifically stated, we will not use asymptotic approximation.

Note 2: This is the *worst-case* approximation. Other varieties exist, probably most common one being *average-case* approximation: the approximation for the average case, assuming some distribution on the input. However, usually the real distribution is unknown, and average case analysis assumptions are often pretty naive. For example, in the makespan scheduling problem, if p_j is uniformly distributed in $[0, 1]$, it is possible to get a much better approximation ratio.

A more involved variety is the *worst average case (WAC)* approximation. This considers the case where an adversary chooses the input (selecting the worst for the algorithm), but the order in which inputs from this set appear is chosen randomly. The WAC approximation for the above algorithm is obviously worse than the average-case, and is again close to 2.

2.1.2 Proving inapproximation bounds

Theorem 1. *For the uniform machine model the natural online greedy algorithm always produces a solution within a factor of* $(2 - \frac{1}{m})$ *of optimal solution for all possible sequences [1]. Also, this bound is tight.*

In general, the methods for proving inapproximation bounds for an algorithm include:

Charging argument. Argument based on reassigning solution costs/profits.

"Promising" solutions. Show that at any point in time the partial solution can be extended to a "good" solution. Note: This method is essentially for greedy algorithms.

Observing properties of the optimal solution. Say something about the properties that OPT has to observe and reason within those parameters.

On the following proof we will use the third method.

Proof. Assume some general input I. We note that the optimal latest finishing time (OPT$[I]$) must be at least as large as the duration of the longest process:

$$\mathrm{OPT}[I] \geq \max_j p_j$$

Another lower bound on the optimal solution can be obtained by taking the combined load and dividing it evenly across the machines:

$$\mathrm{OPT}[I] \geq (\sum_j p_j)/m$$

Now, let us consider the solution given by the natural online greedy algorithm ALG. Let C_{max} be the cost of that solution. That means that on some machine the largest finishing time is C_{max}, and let p_k be (the cost of) the last process on that machine (break ties arbitrarily). We can say that p_k *determines* the makespan.

At the time that p_k is scheduled, its machine was the one with the smallest load. Let $C = C_{max} - p_k$ be the load on that machine when the k^{th} job was scheduled. By the definition of the algorithm, the k^{th} job was scheduled on the least loaded machine and hence we have $C \leq (\sum_{j \neq k} p_j)/m$.

Then the maximum cost of the solution is:

$$C_{max} = C + p_k \tag{1}$$
$$\leq (\sum_{j \neq k} p_j)/m + p_k \tag{2}$$
$$\tag{3}$$

Recall that $\text{OPT}[I] \geq (\sum_j p_j)/m = (\sum_{j \neq k} p_j)/m + p_k/m$.

Then

$$C_{max} \leq (\sum_{j \neq k} p_j)/m + p_k \tag{4}$$
$$= \sum_j p_j/m - p_k/m + p_k \tag{5}$$
$$\leq OPT - p_k/m + p_k \tag{6}$$
$$= OPT + (p_k)(1 - 1/m) \tag{7}$$
$$\leq OPT + OPT(1 - 1/m) \tag{8}$$
$$= (2 - 1/m)OPT \tag{9}$$

So, for all I, $\text{ALG}[I] \leq (2 - 1/m)\text{OPT}[I]$

The bound is tight. Worst case example: let $I = (1, 1, \cdots, 1, m)$ contain $m(m-1)$ unit processes (with load 1), followed by a single process with load m. The online algorithm will distribute the unit processes uniformly, and then schedule the last process on one of the machines. This obtains the cost $\frac{m(m-1)}{m} + m = 2m - 1$.

The optimal solution is to distribute the unit processes among $m - 1$ machines, and assign the last process to the remaining machine. This obtains a solution of cost m.

So, for this I, $\text{ALG}[I] = 2m - 1 = (2 - 1/m)m = (2 - 1/m)\text{OPT}[I]$.

□

3 Longest Processing Time (LPT) algorithm

The following algorithm is still greedy in the following sense: it processes the jobs one at a time. It is customary to call those kinds of algorithms greedy,

although perhaps a better word would be "myopic": the algorithm is not able to see the input beyond the current task.

Here the algorithm first sorts the jobs according to the processing time.

```
1 Sort the inputs so that p_1 ≥ p_2 ≥ ··· ≥ p_m.
2 for i = 1, ··· , n do
3     schedule τ(J_i) in the currently least loaded machine
4 end
```

This is a $\left(\frac{4}{3} - \frac{1}{3m}\right)$-approximation algorithm. This bound is tight.

For $m = 2$, the online algorithm has approximation ratio 3/2, and LPT – 7/6. As $m \to \infty$, the ratio for the online algorithm becomes 2, and for LPT it is 4/3.

Whether or not this is the best greedy algorithm for the problem is still an open question. It has been shown that an online algorithm can beat the ratio of 2, although an exact optimal ratio has not been shown yet.

4 Combining brute force and greediness

The main idea of the algorithm is to optimally solve the problem for the large processes, and then greedily assign the remaining ones. A job J_i is *large* iff:

$$p_i \geq \frac{\sum_j p_j}{sm}$$

where s is a parameter to the algorithm. So, the algorithm is as follows:

```
1 Optimally schedule all large jobs using brute force algorithm.
2 Use online greedy algorithm to fill in small jobs.
```

Let the job J_k be the one that determines makespan.

Case 1. J_k is a large job. In this case, ALG = OPT, since brute force algorithm is optimal.

Case 2. J_k is a small job. Then $p_k \leq \sum p_i/(sm)$.

Then,

$$\text{ALG} \leq (\sum_{i \neq k} \frac{p_i}{m}) + p_k \tag{10}$$

$$\leq (\sum_{i \neq k} \frac{p_i}{m}) + \frac{\sum p_i}{sm} \tag{11}$$

$$\leq (1 + 1/s)\text{OPT} \tag{12}$$

If we want an algorithm with a ratio $1 + \epsilon$, we need to set $s \geq \frac{1}{\epsilon}$.

Then the complexity, dominated by the brute force algorithm, is $m^{m/\epsilon}$. Even for a fixed ϵ, this is exponential in m.

5 Local search

Another technique for computing a $2 - \frac{1}{m}$ aproximation is the following local search algorithm.

---

1 Choose any initial solution S.
2 **while** *There exists a job J_i that defines the maekspan which can be moved so as to decrease the makespan* **do**
3 Move J_i to decrease makespan
4 **end**

Local search finds a local optimum. The largest ratio between a local optimum and the global optimum is called the *locality gap*. In this case, the locality gap is $2 - \frac{1}{m}$ and hence this algorithm is a $2 - \frac{1}{m}$ approximation, and this bound is tight. Proving this is part of the assigned exercise.

References

[1] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Tech. J.*, 45:1563–1581, 1966.

[2] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17:416–429, 1969.