CS261: A Second Course in Algorithms
Lecture #13: Online Scheduling and Online Steiner Tree*

Tim Roughgarden†

February 16, 2016

1 Preamble

Last week we began our study of online algorithms with the multiplicative weights algorithm for online decision-making. We also covered (non-online) applications of this algorithm to zero-sum games and the fast approximation of certain linear programs. This week covers more "traditional" results in online algorithms, with applications in scheduling, matching, and more.

Recall from Lecture #11 what we mean by an online problem.

An Online Problem

1. The input arrives "one piece at a time."
2. An algorithm makes an irrevocable decision each time it receives a new piece of the input.

2 Online Scheduling

A canonical application domain for online algorithms is scheduling, with jobs arriving online (i.e., one-by-one). There are many algorithms and results for online scheduling problems; we'll cover only what is arguably the most classic result.

2.1 The Problem

To specify an online problem, we need to define how the input arrives at what action must be taken at each step. There are m identical machines on which jobs can be scheduled;

†Department of Computer Science, Stanford University, 474 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: tim@cs.stanford.edu.

these are known up front. Jobs then arrive online, one at a time, with job j having a known processing time p_j. A job must be assigned to a machine immediately upon its arrival.

A *schedule* is an assignment of each job to one machine. The *load* of a machine in a schedule is the sum of the processing times of the jobs assigned to it. The *makespan* of a schedule is the maximum load of any machine. For example, see Figure 1.

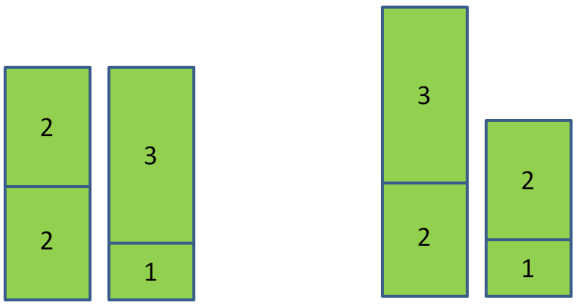

Figure 1: Example of makespan assignments. (a) has makespan 4 and (b) has makespan 5.

We consider the objective function of minimizing the makespan. This is arguably the most practically relevant scheduling objective. For example, if jobs represent pieces of a task to be processed in parallel (e.g., MapReduce/Hadoop jobs), then for many tasks the most important statistic is the time at which the last job completes. Minimizing this last completion time is equivalent to minimizing the makespan.

2.2 Graham's Algorithm

We analyze what is perhaps the most natural approach to the problem, proposed and analyzed by Ron Graham 50 years ago.

Graham's Scheduling Algorithm

when a new job arrives, assign it to the machine that currently has the smallest load (breaking ties arbitrarily)

We measure the performance of this algorithm against the strongest-possible benchmark, the minimum makespan in hindsight (or equivalently, the optimal clairvoyant solution).[1] Since the minimum makespan problem is NP-hard, this benchmark is both omniscient about the future and also has unbounded computational power. So any algorithm that does almost as well is a pretty good algorithm!

[1]Note that the "best fixed action" idea from online decision-making doesn't really make sense here.

2.3 Analysis

In the first half of CS261, we were always asking "how do we know when we're done (i.e., optimal)?" This was the appropriate question when the goal was to design an algorithm that always computes an optimal solution. In an online problem, we don't expect any online algorithm to always compute the optimal-in-hindsight solution. We expect to compromise on the guarantees provided by online algorithms with respect to this benchmark.

In the first half of CS261, we were obsessed with "optimality conditions" — necessary and sufficient conditions on a feasible solution for it to be an optimal solution. In the second half of CS261, we'll be obsessed with *bounds* on the optimal solution — quantities that are "only better than optimal." Then, if our algorithm's performance is not too far from our bound, then it is also not too far from the optimal solution.

Where do such bounds come from? For the two case studies today, simple bounds suffice for our purposes. Next lecture we'll use LP duality to obtain such bounds — this will demonstrate that the same tools that we developed to prove the optimality of an algorithm can also be useful in proving approximate optimality.

The next two lemmas give two different simple lower bounds on the minimum-possible makespan (call it OPT), given m machines and jobs with processing times $p_1, \ldots, p_n$.

Lemma 2.1 (Lower Bound #1)

$$OPT \geq \max_{j=1}^{n} p_j.$$

Lemma 2.1 should be clear enough — the biggest job has to go somewhere, and wherever it is assigned, that machine's load (and hence the makespan) will be at least as big as the size of this job.

The second lower bound is almost as simple.

Lemma 2.2 (Lower Bound #2)

$$OPT \geq \frac{1}{m} \sum_{j=1}^{n} p_j.$$

Proof: In every schedule, we have

$$\begin{aligned} \text{maximum load of a machine} &\geq \text{average load of a machine} \\ &= \frac{1}{m} \sum_{j=1}^{n} p_j. \end{aligned}$$

■

These two lemmas imply the following guarantee for Graham's algorithm.

Theorem 2.3 *The makespan of the schedule output by Graham's algorithm is always at most twice the minimum-possible makespan (in hindsight).*

In online algorithms jargon, Theorem 2.3 asserts that Graham's algorithm is *2-competitive*, or equivalently has a *competitive ratio* of at most 2.

Theorem 2.3 is tight in the worst case (as $m \to \infty$), though better bounds are possible in the (often realistic) special case where all jobs are relatively small (see Exercise Set #7).

Proof of Theorem 2.3: Consider the final schedule produced by Graham's algorithm, and suppose machine i determines the makespan (i.e., has the largest load). Let j denote the last job assigned to i. Why was j assigned to i at that point? It must have been that, at that time, machine i had the smallest load (by the definition of the algorithm). Thus prior to j's assignment, we had

$$\begin{aligned} \text{load of } i &= \text{minimum load of a machine (at that time)} \\ &\leq \text{average load of a machine (at that time)} \\ &= \frac{1}{m}\sum_{k=1}^{j-1} p_k. \end{aligned}$$

Thus,

$$\begin{aligned} \text{final load of machine } i &\leq \underbrace{\frac{1}{m}\sum_{k=1}^{j-1} p_k}_{\leq OPT} + \underbrace{p_j}_{\leq OPT} \\ &\leq 2OPT, \end{aligned}$$

with the last inequality following from our two lower bounds on OPT (Lemma 2.1 and 2.2). ■

Theorem 2.3 should be taken as a representative result in a very large literature. Many good guarantees are known for different online scheduling algorithms and different scheduling problems.

3 Online Steiner Tree

We have two more case studies in online algorithms: the online Steiner tree problem (this lecture) and the online bipartite matching problem (next lecture).[2]

3.1 Problem Definition

In the online Steiner tree problem:

[2]Because the benchmark of the best-possible solution in hindsight is so strong, for many important problems, all online algorithm have terrible competitive ratios. In these cases, it is important to change the setup so that theory can still give useful advice about which algorithm to use. See the instructor's CS264 course ("beyond worst-case analysis") for much more on this. In CS261, we'll cherrypick a few problems where there *are* natural online algorithms with good competitive ratios.

- an algorithm is given in advance a connected undirected graph $G = (V, E)$ with a nonnegative cost $c_e \geq 0$ for each edge $e \in E$;
- "terminals" $t_1, \ldots, t_k \in V$ arrive online (i.e., one-by-one).

The requirement for an online algorithm is to maintain at all times a subgraph of G that spans all of the terminals that have arrived thus far. Thus when a new terminal arrives, the algorithm must connect it to the subgraph-so-far. Think, for example, of a cable company as it builds new infrastructure to reach emerging markets. The gold standard is to compute the minimum-cost subgraph that spans all of the terminals (the "Steiner tree").[3] The goal of an online algorithm is to get as close as possible to this gold standard.

3.2 Metric Case vs. General Case

A seemingly special case of the online Steiner tree problem is the *metric* case. Here, we assume that:

1. The graph G is the complete graph.[4]
2. The edges satisfy the *triangle inequality*: for every triple $u, v, w \in V$ of vertices,

$$c_{uw} \leq c_{uv} + c_{vw}.$$

The triangle inequality asserts that the shortest path between any two vertices is the direct edge between the vertices (which exists, since G is complete) — that is, adding intermediate destinations can't help. The condition states that one-hop paths are always at least as good as two-hop paths; by induction, one-hop paths are as good as arbitrary paths between the two endpoints.

For example, distances between points in a normed space (like Euclidean space) satisfy the triangle inequality. Fares for airline tickets are a non-example: often it's possible to get a cheaper price by adding intermediate stops.

It turns out that the metric case of the online Steiner tree problem is no less general than the general case.

Lemma 3.1 *Every α-competitive online algorithm for the metric case of the online Steiner tree problem can be transformed into an α-competitive online algorithm for the general online Steiner tree problem.*

Exercise Set #7 asks you to supply the proof.

[3] Since costs are nonnegative, this is a tree, without loss of generality.

[4] By itself, this is not a substantial assumption — one could always complete an arbitrary graph with super-high-cost edges.

3.3 The Greedy Algorithm

We'll study arguably the most natural online Steiner tree algorithm, which greedily connects a new vertex to the subgraph-so-far in the cheapest-possible way.[5]

Greedy Online Steiner Tree

initialize $T \subseteq E$ to the empty set
for each terminal arrival t_i, $i = 2, \ldots, k$ **do**
    add to T the cheapest edge of the form (t_i, t_j) with $j < i$

For example, in the 11th iteration of the algorithm, the algorithm looks at the 10 edges between the new terminal and the terminals that have already arrived, and connects the new terminal via the cheapest of these edges.[6]

3.4 Two Examples

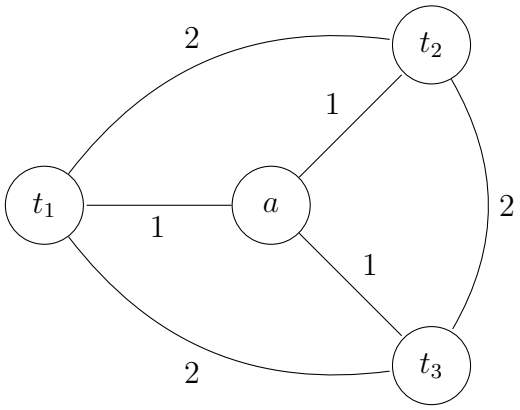

Figure 2: First example.

For example, consider the graph in Figure 2, with edge costs as shown. (Note that the triangle inequality holds.) When the first terminal t_1 arrives, the online algorithm doesn't have to do anything. When the second terminal t_2 arrives, the algorithm adds the edge (t_1, t_2), which has cost 2. When terminal t_3 arrives, the algorithm is free to connect it to either t_1 or t_2 (both edges have cost 2). In any case, the greedy algorithm constructs a

[5] What else could you do? An alternative would be to build some extra infrastructure, hedging against the possibility of future terminals that would otherwise require redundant infrastructure. This idea actually beats the greedy algorithm in non-worst-case models (see CS264).

[6] This is somewhat reminiscent of Prim's minimum-spanning tree algorithm. The difference is that Prim's algorithm processes the vertices in a greedy order (the next vertex to connect is the closest one), while the greedy algorithm here is online, and has to process the terminals in the order provided.

subgraph with total cost 4. Note that the optimal Steiner tree in hindsight has cost 3 (the spokes).

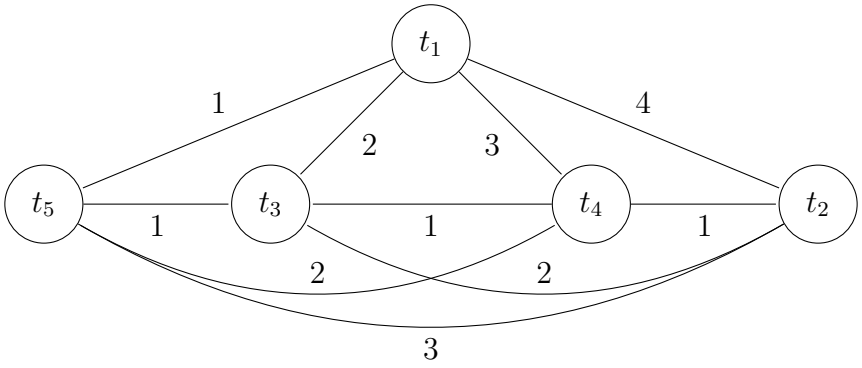

Figure 3: Second example.

For a second example, consider the graph in Figure 3. Again, the edge costs obey the triangle inequality. When t_1 arrives, the algorithm does nothing. When t_2 arrives, the algorithm adds the edge (t_1, t_2), which has cost 4. When t_3 arrives, there is a tie between the edges (t_1, t_3) and (t_2, t_3), which both have cost 2. Let's say that the algorithm picks the latter. When terminals t_4 and t_5 arrive, in each case there are two unit-cost options, and it doesn't matter which one the algorithm picks. At the end of the day, the total cost of the greedy solution is $4 + 2 + 1 + 1 = 8$. The optimal solution in hindsight is the path graph t_1-t_5-t_3-t_4-t_2, which has cost 4.

3.5 Lower Bounds

The second example above shows that the greedy algorithm cannot be better than 2-competitive. In fact, it is not constant-competitive for any constant.

Proposition 3.2 *The (worst-case) competitive ratio of the greedy online Steiner tree algorithm is $\Omega(\log k)$, where k is the number of terminals.*

Exercise Set #7 asks you to supply the proof, by extending the second example above.

The following result is harder to prove, but true.

Proposition 3.3 *The (worst-case) competitive ratio of every online Steiner tree algorithm, deterministic or randomized, is $\Omega(\log k)$.*

3.6 Analysis of the Greedy Algorithm

We conclude the lecture with the following result.

Theorem 3.4 *The greedy online Steiner tree algorithm is* $2 \ln k$*-competitive, where* k *is the number of terminals.*

In light of Proposition 3.3, we conclude that the greedy algorithm is an optimal online algorithm (in the worst case, up to a small constant factor).

The theorem follows easily from the following key lemma, which relates the costs incurred by the greedy algorithm to that of the optimal solution in hindsight.

Lemma 3.5 *For every* $i = 1, 2, \ldots, k-1$*, the* i*th most expensive edge in the greedy solution* T *has cost at most* $2OPT/i$*, where* OPT *is the cost of the optimal Steiner tree in hindsight.*

Thus, the most expensive edge in the greedy solution has cost at most $2OPT$, the second-most expensive edge costs at most OPT, the third-most at most $2OPT/3$, and so on. Recall that the greedy algorithm adds exactly one edge in each of the $k-1$ iterations after the first, so Lemma 3.5 applies (with a suitable choice of i) to each edge in the greedy solution.

To apply the key lemma, imagine sorting the edges in the final greedy solution from most to least expensive, and then applying Lemma 3.5 to each (for successive values of $i = 1, 2, \ldots, k-1$). This gives

$$\text{greedy cost} \leq \sum_{i=1}^{k-1} \frac{2OPT}{i} = 2OPT \sum_{i=1}^{k-1} \frac{1}{i} \leq (2 \ln k) \cdot OPT,$$

where the last inequality follows by estimating the sum by an integral.

It remains to prove the key lemma.

Proof of Lemma 3.5: The proof uses two nice tricks, "tree-doubling" and "shortcutting," both of which we'll reuse later when we discuss the Traveling Salesman Problem.

We first recall an easy fact from graph theory. Suppose H is a connected multi-graph (i.e., parallel copies of an edge are OK) in which every vertex has even degree (a.k.a. an "Eulerian graph"). Then H has an *Euler tour*, meaning a closed walk (i.e., a not-necessarily-simple cycle) that uses every edge exactly once. See Figure 4. The all-even-degrees condition is clearly necessary, since if the tour visits a vertex k times then it must have degree $2k$. You've probably seen the proof of sufficiency in a discrete math course; we leave it to Exercise Set #7.[7]

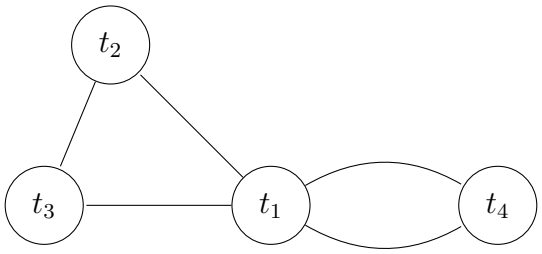

Figure 4: Example graph with Euler tour t_1-t_2-t_3-t_1-t_4-t_1.

[7]Basically, you just peel off cycles one-by-one until you reach the empty graph.

Next, let T^* be the optimal Steiner tree (in hindsight) spanning all of the terminals $t_1, \ldots, t_k$. Let $OPT = \sum_{e \in T^*} c_e$ denote its cost. Obtain H from T^* by adding a second copy of every edge (Figure 5). Obviously, H is Eulerian (every vertex degree got doubled) and $\sum_{e \in H} c_e = 2OPT$. Let C denote an Euler tour of H. C visits each of the terminals at least one, perhaps multiple times, and perhaps visits some other vertices as well. Since C uses every edge of H once, $\sum_{e \in C} c_e = 2OPT$.

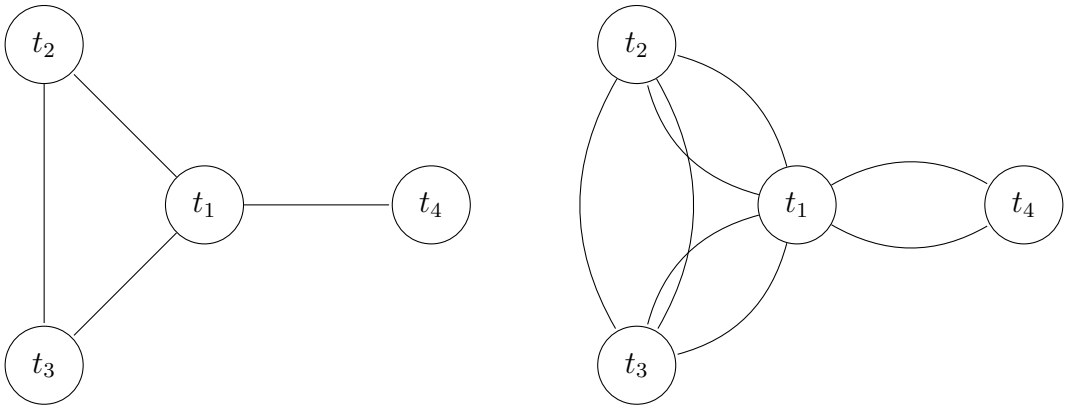

Figure 5: (a) Before doubling edges and (b) after doubling edges.

Now fix a value for the parameter $i \in \{1, 2, \ldots, k-1\}$ in the lemma statement. Define the "connection cost" of a terminal t_j with $j > 1$ as the cost of the edge that was added to the greedy solution when t_j arrived (from t_j to some previous terminal). Sort the terminals in hindsight in nonincreasing order of connection cost, and let $s_1, \ldots, s_i$ be the first (most expensive) i terminals. The lemma asserts that the cheapest of these has connection cost at most $2OPT/i$. (The ith most expensive terminal is the cheapest of the i most expensive terminals.)

The tour C visits each of $s_1, \ldots, s_i$ at least once. "Shortcut" it to obtain a simple cycle C_i on the vertex set $\{s_1, \ldots, s_i\}$ (Figure 6). For example, if the first occurrences of the terminals in C happen to be in the order $s_1, \ldots, s_i$, then C_i is just the edges $(s_1, s_2), (s_2, s_3), \ldots, (s_i, s_1)$. In any case, the order of terminals on C_i is the same as that of their first occurrences in C. Since the edge costs satisfy the triangle inequality, replacing a path by a direct edge between its endpoints can only decrease the cost. Thus $\sum_{e \in C_i} c_e \leq \sum_{e \in C} c_e = 2OPT$. Since C_i only has i edges,

$$\underbrace{\min_{e \in C_i} c_e}_{\text{cheapest edge}} \leq \underbrace{\frac{1}{i} \sum_{e \in C_i} c_e}_{\text{average edge cost}} \leq 2OPT/i.$$

Thus some edge $(s_h, s_j) \in C_i$ has cost at most $2OPT/i$.

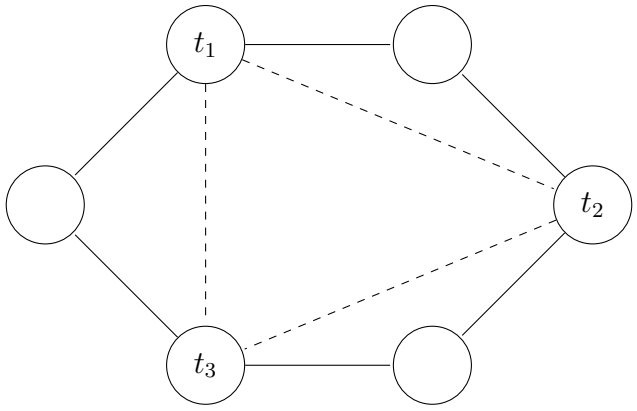

Figure 6: (a) Solid edges represent original edges, and dashed edge represent edges after shortcutting from t_1 to t_2, t_2 to t_3, t_3 to t_1 has been done.

Consider whichever of s_h, s_j arrives later in the online ordering, say s_j. Since s_h arrived earlier, the edge (s_h, s_j) is one option for connecting s_j to a previous terminal; the greedy algorithm either connects s_j via this edge or by one that is even cheaper. Thus at least one vertex of $\{s_1, \ldots, s_i\}$, namely s_j, has connection cost at most $2OPT/i$. Since these are by definition the terminals with the i largest connection costs, the proof is complete. ■