St. Francis Xavier University
Department of Computer Science

CSCI 550: Approximation Algorithms
Lecture 5: Deterministic Rounding of Linear Programs
Fall 2021

1 Getting "A-Round"

In this lecture, we will continue our discussion on using rounding to obtain approximate solutions to problems, but this time we will shift our focus to rounding of linear programs. Recall that we were first introduced to linear programs in our first lecture. A linear program is a formulation of an optimization problem in terms of decision variables that represent the decisions being made to solve the problem. The decision variables are constrained, generally by linear inequalities, and an assignment of values to each decision variable that satisfies all of the constraints is called a feasible solution.

How do we apply rounding to a linear program? Remember that the reason why we care about linear programs is because they act as a "generalization" of sorts of some integer program, and integer programs are very hard to solve. By relaxing an integer program to a linear program, we can obtain a solution in a more reasonable amount of time, but the solution to the linear program is no longer guaranteed to be integral. Using rounding, then, we can convert the fractional solution to a linear program to an approximate integer solution.

Typically, our rounding process will have us round up "large" values to 1 and round down "small" values to 0. Indeed, that's what we did when we applied the rounding technique to the weighted set cover problem in our first lecture. However, we are not limited to always using rounding in this way. As we will see, we could alternatively solve the relaxed linear program to gain information about how the solution should look for the corresponding integer program, and then use that relaxed linear program solution to construct an approximate solution for the integer program.

Here, we will focus on the familiar scheduling problem that we have seen in past lectures, and we will consider both the standard scheduling problem as well as a weighted variant. We will then introduce an efficient general method of solving very large linear programs known as the *ellipsoid method*.

1.1 Sequential Scheduling on a Single Machine

To motivate our topic in the next section, we will begin by studying a variant of our familiar single-machine scheduling problem. Recall that, when we first introduced this problem, our goal was to minimize the maximum lateness of any job to be completed. Here, we will consider a variant of the single-machine scheduling problem where each job must be completed *nonpreemptively*—that is, one job must be finished before another job can be processed— and where our goal is to minimize the overall job completion time.

Scheduling-Single-Machine-Nonpreemptive
Given: a set of n jobs S and, for each job j, a release time r_j and a processing time p_j
Determine: a nonpreemptive schedule that minimizes the sum of completion times $\sum_{j=1}^{n} C_j$ of all jobs j

Because jobs in this variant problem must be completed nonpreemptively, we can't stop processing one job i in favour of processing another, shorter job j before returning to the original job i. Once we begin processing job i, we are required to finish processing it before we can move on to any other job. This requirement imposes an additional constraint on whatever schedule we create. Fortunately, however, we don't need to attempt to construct a nonpreemptive schedule from scratch: we can simply create a *preemptive* schedule that allows us to interrupt jobs, and then convert that preemptive schedule to a nonpreemptive schedule with just a small penalty incurred on the total completion time.

We will use this observation about the preemptive single-machine scheduling problem to obtain an approximate solution to the nonpreemptive single-machine scheduling problem. To start, we construct an optimal solution for the preemptive single-machine scheduling problem in polynomial time using the following algorithm:

Algorithm 1: Single-machine scheduling, preemptive—shortest remaining processing time

```
for each $t \geq 0$ do
    $j \leftarrow$ released and incomplete job with shortest processing time $p_j$
    run job $j$ on the machine
```

As we can see, Algorithm 1 simply looks for the job j that is both incomplete and has the shortest processing time out of all currently-released jobs at time t. For that unit of time t, if job j is different from the job the single machine is currently processing, then the machine drops its current job and starts working on job j. The machine then repeats this process for each unit of time t until all jobs are complete.

Let C_j^P denote the completion time of job j in an optimal preemptive schedule, and let OPT denote the sum of completion times in an optimal nonpreemptive schedule. Since the optimal nonpreemptive schedule is itself a feasible solution for the preemptive single-machine scheduling problem, we immediately get the following bound.

Lemma 1.

$$\sum_{j=1}^{n} C_j^P \leq \text{OPT}.$$

Now, we can use our algorithm for the preemptive single-machine scheduling problem to obtain a nonpreemptive schedule.

Algorithm 2: Single-machine scheduling, nonpreemptive

```
run Algorithm 1 on the given set of jobs
for each job $j = \{1, \ldots, n\}$ ordered such that $C_1^P \leq \cdots \leq C_n^P$ do
    if $j = 1$ then
        run job 1 from time $r_1$ to time $r_1 + p_1$
    else
        run job $j$ from time $\max\{r_{j-1} + p_{j-1}, r_j\}$ to time $\max\{r_{j-1} + p_{j-1}, r_j\} + p_j$
```

In Algorithm 2, we use the preemptive schedule constructed by Algorithm 1 as a template to order the jobs by completion time. We can then schedule each of the jobs in this same order in a nonpreemptive way; if C_{j-1}^N denotes the completion time of job $j-1$ in this nonpreemptive schedule, then the machine processes job j from time $\max\{C_{j-1}^N, r_j\}$ to time $\max\{C_{j-1}^N, r_j\} + p_j$.

As we mentioned earlier, performing this conversion from a preemptive schedule to a nonpreemptive schedule does not incur a big penalty on the total completion time. Indeed, we only suffer a factor-of-two difference.

Lemma 2. *For each job j, where $1 \leq j \leq n$,*

$$C_j^N \leq 2 \cdot C_j^P.$$

Proof. In the optimal preemptive schedule, we know that job j is processed after each of jobs 1 through $j-1$. Therefore, we have that

$$C_j^P \geq \max_{k=\{1,\ldots,j\}} r_k \tag{1}$$

and

$$C_j^P \geq \sum_{k=1}^{j} p_k. \tag{2}$$

The first inequality tells us that the completion time of job j is lower-bounded by the latest release time of any job from 1 to j, while the second inequality tells us that the completion time of job j is also lower-bounded by the sum of processing times of each job from 1 to j.

By our construction of the nonpreemptive schedule, and by a similar argument, we also have that

$$C_j^N \geq \max_{k=\{1,\ldots,j\}} r_k.$$

Consider the nonpreemptive schedule constructed by Algorithm 2. In this schedule, the machine is idle only when the next job to be processed has not been released. Therefore, in the time interval from $\max_{k=\{1,\ldots,j\}} r_k$ to C_j^N, the machine cannot be idle. Moreover, this interval must have length at most $\sum_{k=1}^{j} p_k$, or else the machine would run out of jobs to process. Therefore, we have that

$$\begin{aligned} C_j^N &\leq \max_{k=\{1,\ldots,j\}} r_k + \sum_{k=1}^{j} p_k \\ &\leq 2 \cdot C_j^P, \end{aligned}$$

where the second inequality follows from Equations 1 and 2. □

As a consequence of Lemmas 1 and 2, we get the main result.

Theorem 3. *Algorithm 2 gives a 2-approximation algorithm for the nonpreemptive single-machine scheduling problem.*

Proof. We have that

$$\sum_{j=1}^{n} C_j^N \leq 2 \cdot \sum_{j=1}^{n} C_j^P \leq 2 \cdot \mathrm{OPT}.$$

□

1.2 Weighted Sequential Scheduling on a Single Machine

Having reviewed the single-machine scheduling problem, let's focus on a slight variant of the problem. Just as we considered the *weighted* set cover problem as a variant of the set cover problem, so too can we consider the *weighted* scheduling problem as a variant of the scheduling problem. Like before, our goal is to minimize the overall job completion time, but now we must minimize the *weighted* sum.

Scheduling-Single-Machine-Nonpreemptive-Weighted
Given: a set of n jobs S and, for each job j, a release time r_j, a processing time p_j, and a weight w_j
Determine: a nonpreemptive schedule that minimizes the weighted sum of completion times $\sum_{j=1}^{n} w_j C_j$ of all jobs j

Given the similarities between this problem and the scheduling problem of the previous section, it's reasonable to think that we can just reuse the algorithms we developed earlier to find a schedule for a set of weighted jobs. Unfortunately, however, things aren't so straightforward. Unlike our unweighted preemptive scheduling problem in the previous section, finding an optimal schedule for the weighted preemptive single-machine scheduling problem is NP-hard. However, this doesn't mean all is lost; we can still use the same ideas from the previous section to obtain an approximation algorithm for our current problem.

The main idea of this section is to use relaxation to obtain a linear program for the problem, where the variables of the linear program are the job completion times C_j. If we can show that the inequalities of the linear program hold within a constant factor, then we can obtain a constant-factor approximation algorithm.

From the statement of the problem, we can immediately get the objective of our linear program: we want to minimize the value $\sum_{j=1}^{n} w_j C_j$. We also have two constraints on the program:

- for each job $j = \{1, \dots, n\}$, any job j cannot be completed before it is released and processed; and
- for some subset of jobs $S \subseteq \{1, \dots, n\}$, given the sum $\sum_{j \in S} p_j C_j$, it is the case that

$$\sum_{j \in S} p_j C_j = \sum_{\substack{j,k \in S \\ j \leq k}} p_j p_k = \frac{1}{2}\left(\sum_{j \in S} p_j\right)^2 + \frac{1}{2}\sum_{j \in S} p_j^2 \geq \frac{1}{2}\left(\sum_{j \in S} p_j\right)^2.$$

The first constraint is straightforward, so let's focus on the motivation for the second constraint. Given a subset of jobs $S \subseteq \{1, \dots, n\}$, the sum $\sum_{j \in S} p_j C_j$ is minimized when all jobs in S are released at time 0 and when all jobs in S finish first in the schedule. Otherwise, if not all jobs in S are released at time 0 or not all jobs in S finish first in the schedule, then the sum must be greater than this minimal value. In any case, the completion time C_j for a job $j \in S$ is equal to its processing time p_j plus the sum of the processing times of all jobs that were processed before job j. Therefore, the sum $\sum_{j \in S} p_j C_j$ must contain a product term $p_j p_k$ for all jobs $j, k \in S$, and from this we get the equation in the second constraint.

Altogether, we can write the linear program as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{m} w_j x_j \\ \text{subject to} \quad & C_j \geq r_j + p_j \qquad \text{for all } j \in \{1, \dots, n\} \\ & \sum_{j \in S} p_j C_j \geq \frac{1}{2}\left(\sum_{j \in S} p_j\right)^2 \qquad \text{for all } S \subseteq \{1, \dots, n\} \end{aligned} \tag{SSMW-LP}$$

Since this linear program is a relaxation of the problem, we know that an optimal solution C^* to the linear program will be subject to the upper bound $\sum_{j=1}^{n} w_j C_j^* \leq \text{OPT}$. Given the optimal solution C^*, we use the same idea as in the previous section to obtain the desired schedule for the original instance of the problem: we take the ordering of jobs by completion time given by the relaxed optimal solution, $C_1^* \leq \cdots \leq C_n^*$, schedule job 1 to run from time r_1 to time $r_1 + p_1$, and schedule all other jobs j to run from time $\max\{r_{j-1} + p_{j-1}, r_j\}$ to time $\max\{r_{j-1} + p_{j-1}, r_j\} + p_j$.

Using this approach, we obtain an approximation algorithm for the weighted problem with a slightly worse performance guarantee than we obtained for the unweighted problem in the previous section.

Theorem 4. *Modifying Algorithm 2 gives a 3-approximation algorithm for the weighted nonpreemptive single-machine scheduling problem.*

Proof. Assume that each job is ordered such that $C_1^* \leq \cdots \leq C_n^*$, and let C_j^N denote the completion time of job j in the nonpreemptive schedule constructed by the algorithm. As in the proof of Lemma 2, we know that the machine cannot be idle between time $\max_{k=\{1,\dots,j\}} r_k$ and time C_j^N. Therefore, we must have that

$$C_j^N \leq \max_{k=\{1,\dots,j\}} r_k + \sum_{k=1}^{j} p_k.$$

Let $\ell \in \{1, \dots, j\}$ denote the index of the job where $r_\ell = \max_{k=\{1,\dots,j\}} r_k$ By our assumed ordering of jobs, we know that $C_\ell^* \leq C_j^*$, and moreover we know that $r_\ell \leq C_\ell^*$ by the first constraint of the linear program SSMW-LP. Therefore, $\max_{k=\{1,\dots,j\}} r_k \leq C_j^*$.

Now, let $S = \{1, \ldots, j\}$. Since we know that C^* is a feasible solution to the linear program SSMW-LP, we know that

$$\sum_{k \in S} p_k C_k^* \geq \frac{1}{2} \left(\sum_{k \in S} p_k \right)^2$$

by the second constraint of the linear program. However, from our assumed ordering of jobs, we have that $C_j^* \geq \cdots \geq C_1^*$, and so

$$C_j^* \sum_{k \in S} p_k \geq \sum_{k \in S} p_k C_k^*.$$

After combining these two inequalities, we get

$$C_j^* \sum_{k \in S} p_k \geq \frac{1}{2} \left(\sum_{k \in S} p_k \right)^2,$$

and, dividing both sides by $\sum_{k \in S} p_k$, we get that $C_j^* \geq \frac{1}{2} \left(\sum_{k \in S} p_k \right)$ and therefore $2 \cdot C_j^* \geq \sum_{k \in S} p_k$.

Altogether, we have

$$C_j^N \leq \max_{k=\{1,\ldots,j\}} r_k + \sum_{k \in S} p_k \leq C_j^* + 2 \cdot C_j^* = 3 \cdot C_j^*. \qquad \square$$

1.3 The Ellipsoid Method

You may have noticed in the previous section that we never discussed the details of how to solve the linear program SSMW-LP. Indeed, even as far back as our first lecture, we simply assumed that there was *some* way to solve a given linear program before jumping into the analysis of our approximation algorithm.

There exists a handful of methods to solve linear programs. One of the most well-known methods, as well as the earliest method, is known as the *simplex method*. First developed by George Dantzig during World War II and refined throughout the 1970s and 1980s by American and Soviet scientists separately, the simplex method computes an optimal solution in the following way: starting at some corner point in the "feasibility region" of the problem where each variable is zero, the method traverses each corner point in this region and improves on the objective function at each step. The method terminates once the optimal solution is found.[1]

While the simplex method remains quite popular and is rather efficient in practice, one major drawback of the method is that neither it nor any variation of it is known to run in polynomial time. Indeed, for each variant of the simplex method, researchers have found a family of linear programs for which the method requires exponential time to complete. Naturally, since the main goal of this course is to find efficient algorithms for difficult problems, we don't want to introduce a step into our algorithms that may take exponential time to run!

Instead, for the purposes of solving linear programs, we will turn to a technique known as the *ellipsoid method*. First introduced in the 1970s by the Russian mathematician Naum Shor, and applied to linear programs shortly thereafter by the Soviet-American mathematician Leonid Khachiyan, the idea behind the ellipsoid method is to use a series of ellipsoids, with each ellipsoid decreasing in volume after each iteration, to give a bound on the size of the feasible solution space for the problem. Each smaller ellipsoid is constructed with the help of a special oracle machine called a *separation oracle*. The method halts either when it finds a feasible solution in the space, in which case it establishes feasibility, or when the volume of the ellipsoid is too small, signalling that no feasible solution exists.

[1]This is a very high-level description of the simplex method that glosses over many details. Since we don't focus on the simplex method in this section, we omit many of these details.

(The following paragraphs give a technical description of how the ellipsoid method works. Feel free to skip this portion of the text on your first reading.)

To see how the ellipsoid method works, suppose we have a linear program of the following general form:

$$\begin{aligned} \text{minimize} \quad & \sum_{j=1}^{n} d_j x_j \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \geq b_i, \quad \text{for all } i \in \{1, \ldots, m\} \\ & x_j \geq 0, \quad \text{for all } j \end{aligned} \tag{LP}$$

Suppose also that we can give a bound ϕ on the number of bits needed to encode any inequality of the form $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$. The ellipsoid method gives us a way to find an optimal solution to the linear program LP in time polynomial to both the number of variables, n, and the number of bits, ϕ. Note that the method does *not* depend on the number of constraints, m, which gives us a marked advantage when the linear program has many constraints to satisfy simultaneously.

To find an optimal solution to the linear program, we use a separation oracle, which takes as input a solution x to the linear program that we assume is feasible. It then either verifies that x is, in fact, a feasible solution to the linear program, or produces a constraint that is violated if x is infeasible. More precisely, if it is not true that $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ for all $i \in \{1, \ldots, m\}$, then the separation oracle returns a constraint i such that $\sum_{j=1}^{n} a_{ij} x_j < b_i$.

Given this separation oracle, the ellipsoid method proceeds by finding an ellipsoid in $\mathbb{R}^n$ that contains all feasible solutions for the linear program. Suppose that $\bar{x}$ denotes the center of this ellipsoid. The method calls the separation oracle with this point $\bar{x}$.

- If $\bar{x}$ is a feasible solution, then the method produces a constraint $\sum_{j=1}^{n} d_j x_j \leq \sum_{j=1}^{n} d_j \bar{x}_j$. This constraint signifies that any optimal solution to the problem must have an objective function value no greater than that of the feasible solution $\bar{x}$.
- If $\bar{x}$ is not a feasible solution, then the separation oracle returns a constraint $\sum_{j=1}^{n} a_{ij} x_j < b_i$ that is violated by $\bar{x}$.

In either case, we are able to define a hyperplane through the point $\bar{x}$ such that any optimal solution to the linear program must lie on one side of the hyperplane. This hyperplane effectively divides the ellipsoid into two parts. The method then finds a smaller ellipsoid containing only the half of the region that can include an optimal solution, and repeats the process using the center of this smaller ellipsoid. The iteration process terminates once the ellipsoid is so small that it can contain at most one feasible solution, if it exists.

(This concludes the technical description of the ellipsoid method.)

Even though the ellipsoid method is generally not efficient in practice, it remains a very important tool for developing polynomial-time algorithms involving linear programs, especially where the size of the linear program—specifically, the number of constraints—is exponential in the size of the input.

So, how do we use the ellipsoid method to our advantage? Let's apply it to the linear program SSMW-LP from the previous section. Observe that the second constraint, $\sum_{j \in S} p_j C_j \geq \frac{1}{2} \left(\sum_{j \in S} p_j \right)^2$, applies to all subsets $S \subseteq \{1, \ldots, n\}$. Since there are 2^n such subsets, this constraint of the linear program is exponential in the size of the input.

We can use the ellipsoid method to obtain a polynomial-time separation oracle for this second constraint. Suppose C is a solution to the linear program, and reorder the variables in such a way that $C_1 \leq \cdots \leq C_n$. Then, consider n specific subsets $S_1 = \{1\}$, $S_2 = \{1, 2\}$, ..., $S_n = \{1, \ldots, n\}$. We will show that we only need to check the second constraint for these n subsets, thus reducing the number of checks our algorithm must perform from an exponential amount to a linear amount in the size of the input.

Lemma 5. *Given variables C_j, if the second constraint of the linear program SSMW-LP is satisfied for each of the n subsets $S_1, \ldots, S_n$, then it is satisfied for all subsets $S \subseteq N$.*

Proof. Suppose that S is a subset that does not satisfy the second constraint of the linear program SSMW-LP. We will show that there must exist some subset S_i from our collection of subsets that also does not satisfy the constraint.

To show this, we will make some change to the original subset S that decreases the difference $\sum_{j \in S} p_j C_j - \frac{1}{2}\left(\sum_{j \in S} p_j\right)^2$ and, therefore, tightens the inequality in the constraint. In doing so, we will create a new subset S' that also does not satisfy the second constraint.

Observe that removing a job k from S will decrease the difference if $C_k > \left(\sum_{j \in S} p_j - p_k\right) + \frac{1}{2}p_k$, while adding a job k to S will decrease the difference if $C_k < \left(\sum_{j \in S} p_j - p_k\right) + \frac{1}{2}p_k$.

Let ℓ denote the highest-indexed job in S. As we observed earlier, we remove job ℓ from S if $C_\ell > \left(\sum_{j \in S} p_j - p_\ell\right) + \frac{1}{2}p_\ell$, and by our earlier reasoning we know that the resulting subset $S \setminus \ell$ also does not satisfy the second constraint.

Repeatedly remove the highest-indexed job from S in this way until we end up with a subset S' with the property that its highest-indexed job ℓ' is such that $C_{\ell'} \leq \left(\sum_{j \in S'} p_j - p_{\ell'}\right) + \frac{1}{2}p_{\ell'}$. Now, suppose $S' \neq S_{\ell'} = \{1, \ldots, \ell'\}$. Then, there exists some job $k < \ell'$ such that $k \notin S'$, and since $C_k \leq C_{\ell'}$, adding job k to S' can only decrease the difference. Thus, we add all jobs $k < \ell'$ to S', and in doing so, the resulting subset $S_{\ell'}$ also does not satisfy the second constraint.

Therefore, if the second constraint of the linear program SSMW-LP is satisfied for all n subsets $S_1, \ldots, S_n$, then there cannot exist a subset S that itself does not satisfy the second constraint. □