Fall 2016 IEOR 8100

Scheduling Algorithms
by Professor Clifford Stein

Lecture 2 Notes

Review and Introduction

Last time we discussed scheduling to identical machines to minimize the makespan ($P||C_{max}$). For this problem, we discussed two solutions. First, we discussed List Scheduling, which greedily assigns the next available job from an arbitrary ordering of jobs to a machine whenever it becomes idle. List Scheduling is a 2-approximation. Second, we discussed the Longest Processing Time (LPT) order greedy algorithm, which is a $\frac{4}{3}$-approximation.

In the previous lecture we discussed ρ-approximations for some constant $\rho \in \mathbb{R}$, or solutions whose makespan (C_{max}) is at most a ρ multiple of the optimal solution OPT; that is, $C_{max} \leq (\rho \times OPT)$. In the above examples, $\rho = 2$ for List Scheduling and $\rho = \frac{4}{3}$ for LPT; in both cases it is a fixed constant.

Some algorithms are general enough such that $\forall \epsilon > 0$ the algorithm produces an output whose ratio with the optimal solution is at most $1+\epsilon$; in the context of minimizing makespan in scheduling, such an algorithm produces a schedule with makespan $C_{max} \leq ((1+\epsilon) \times OPT)$. In this description $1+\epsilon$ is analogous to ρ in that it is the ratio between the makespan achieved and the optimal solution. A polynomial time approximation scheme (PTAS) is an algorithm that can be described in this manner and has a running time polynomial in $n^{\mathcal{O}(1/f(\epsilon))}$ where f is some arbitrary function of ϵ. By convention, a PTAS is typically described in terms of the infinitesimal parameter ϵ rather than the ratio ρ.

Even if a PTAS exists for some problem, it may not be practical for actual application. Running time of the PTAS may be very slow for any ϵ sufficiently small to provide a useful approximation.

A fully polynomial time approximation scheme (FPTAS) is similar to a PTAS except that it is polynomial in both n and $\frac{1}{\epsilon}$.

For some problems we can prove that approximation beyond a certain constant is NP-hard, which means there does not exist a ρ-approximation for any ρ smaller than this constant, but if a PTAS exists for a problem, then there cannot be such a proof because the

approximation can be arbitrarily close depending on the choice of ϵ.

Hochbaum and Schmoys' Relaxed Decision Procedure

The algorithm we are about to discuss was published in 1986 by Hochbaum and Schmoys and it is a Relaxed Decision Procedure (RDP). The basic motivation behind using an RDP is that optimization problems can be reduced to decision problems using binary search. An RDP is an algorithm that, given some $\epsilon > 0$ and some desired maximum value T of an objective function will halt in polynomial time and report either that there is no solution with objective at most T or provide an example of a solution with objective at most $(1+\epsilon)\times T$. For example, in the context of minimizing makespan, an RDP would output one of the following:

- The string "There is no schedule with $C_{max} \leq T$"
- A representation of a schedule with $C_{max} \leq (1 + \epsilon) \times T$.

Note that if the optimal solution has a makespans greater than $(1 + \epsilon) \times T$ the algorithm will always correctly say there is no schedule with $C_{max} \leq T$, and if the optimal solution has makespan at most T the algorithm will always provide a schedule (which may or may not be optimal) with $C_{max} \leq (1 + \epsilon) \times T$, but if the optimal solution OPT is such that $T < OPT \leq (1 + \epsilon) \times T$, the algorithm may not report that "There is no schedule with $C_{max} \leq T$", even though that is true, but may instead return a schedule with $C_{max} \leq (1 + \epsilon) \times T$.

If such an algorithm exists, then we can approximate the optimal makespan OPT by doing a binary search that repeatedly calls the RDP for different values of T.

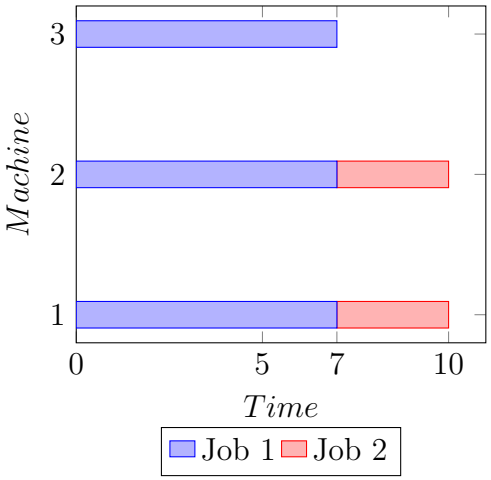
Dynamic Programming Special-Case Solution

First, let us consider the problem $P||C_{max}$ in the special case that among the n jobs, there are at most s unique processing times. Let us rephrase the optimization problem as an RDP by asking for some arbitrary "deadline" time T whether there is a schedule with $C_{max} \leq T$. Consider s to be a constant.

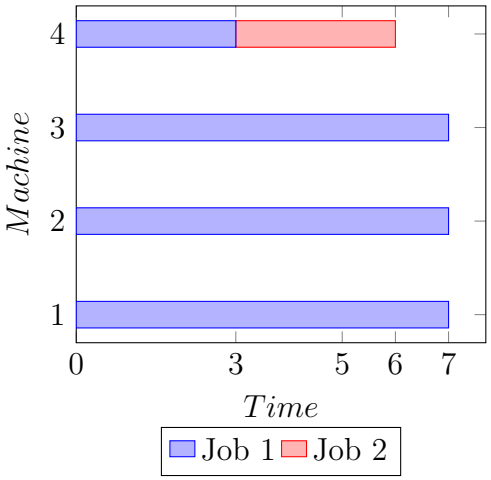
In general with n jobs the jobs are $p_1, p_2, \ldots p_n$. When we describe a schedule that assigns these jobs to m machines, each machine can be uniquely characterized by which jobs it completes, so each machine has at most 2^n states. But when we consider that all n jobs are of some standard size from among only s different sizes, then let us instead describe a machine by a vector $(z_1, z_2, ...z_s)$ where the machine completes z_1 jobs of size 1, etc. Each of these z_i can be at most n, and there are s of them, so the total number of states the machine can have is n^s. This is overcounting somewhat, but for fixed s, it is polynomial in n instead of exponential. Unfortunately this is only the number of states for one machine; the number of possible schedules for m machines is $(n^s)^m$, which quickly becomes intractable for a large

number of machines because it is exponential in m.

Let's consider instead the minimum number of machines needed to process n jobs expressed in batches as (x_i jobs of size s) by time T. Let us call this $M(x_1, \ldots x_s, T)$. For example, if $s_i = (3, 7)$, $x_i = (2, 3)$, and $T = 10$, then $M = 3$:



Thus, for $s = (3, 7)$, $M(2, 3, 10) = 3$. However, $M(2, 3, 9) = 4$:



In both of these examples we can see that the schedules shown are optimal because if we remove the last machine, the remaining machines are at full capacity until the desired deadline, so they could not possibly have completed all of the jobs without the help of the last machine. In general, we can calculate these optimal schedules in polynomial time using a dynamic programming approach.

Let $\mathbb{V}$ be the set of all feasible schedules of 1 machine (n^s such schedules are possible).

Whatever the optimal schedule is overall, it must assign some number of jobs to the last machine. For each possible set of jobs that could be completed by a single machine, the optimal schedule that would assign those jobs to the last machine is the schedule optimal for the remaining jobs on the remaining machines. If we let $v = (z_1, \ldots z_s)$ represent the jobs run on this last machine, then $M(x_1, \ldots x_s, T) = \min_{v \in \mathbb{V}} M(x_1 - z_1, x_2 - z_2, \ldots x_s - z_s, T) + 1$. Since by overcounting slightly we know there are at most n^s possible choices for v and at most n^s possible optimal subschedules containing a given v, solving this by dynamic programming is of complexity $n^{O(s)}$.

This problem may sound familiar to anyone who has studied algorithms in general without a focus on scheduling because it is equivalent to the bin packing problem. Given a finite number of jobs, each of which requires a specified amount of processing time, assigning these to a finite number of machines with some deadline is like packing a finite number of bins of fixed capacity with a collection of objects with finite volume.

Approximating the general case

We can use this dynamic programming approach to approximate the optimal solution to the more general problem of minimizing makespan even in the general case where the arbitrarily many n jobs do not belong to a fixed number s of job sizes. The high-level approach is:

1. Round the job sizes to a fixed number s of different sizes
2. Solve the rounded problem using the dynamic programming algorithm
3. Deal with the caveats of:
    - Unrounding
    - Small jobs

Intuitively, when we round a large job by ϵ, ϵ is small relative to the job size so the error is small, but when we round a small job, the rounding error is a larger fraction of the job size, which is why we need to worry a little bit more about small jobs. However, because small jobs are small, they are also easier to cram into the schedule at the end, and we'll see that for this reason we can get away with rounding them even despite the larger ratio of rounding error to job size.

LEMMA
Suppose all p_j satisfy $p_j \geq (\epsilon \times T)$. Then $\exists((1+\epsilon)\text{-RDP})$ for $P||C_{max}$.
PROOF
Round p_j down to p'_j in integer multiples of $\epsilon^2 \times T$:

- $0 \leq (p_j - p'_j) \leq (\epsilon^2 \times T)$
- # of job sizes $\leq \frac{T}{\epsilon^2 \times T} = \frac{1}{\epsilon^2} = s$
- # of jobs on a single machine in a feasible schedule $\leq \frac{T}{\epsilon \times T} = \frac{1}{\epsilon}$

When we use DP on the rounded instance in $n^{O(s)}$, because the number of job sizes is defined in terms of ϵ, the DP algorithm then runs in $n^{O(1/\epsilon^2)}$, which makes this a PTAS. If the DP outputs no, we output no. If the DP outputs yes, we unround. Since the number of jobs on a single machine is at most $1/epsilon$, unrounding adds at most $(\epsilon^2 \times T \times \frac{1}{\epsilon}) = \epsilon \times T$ time, which means that our overall solution has a makespan which is still at most $(1+\epsilon) \times T$, so we output yes.

Algorithm

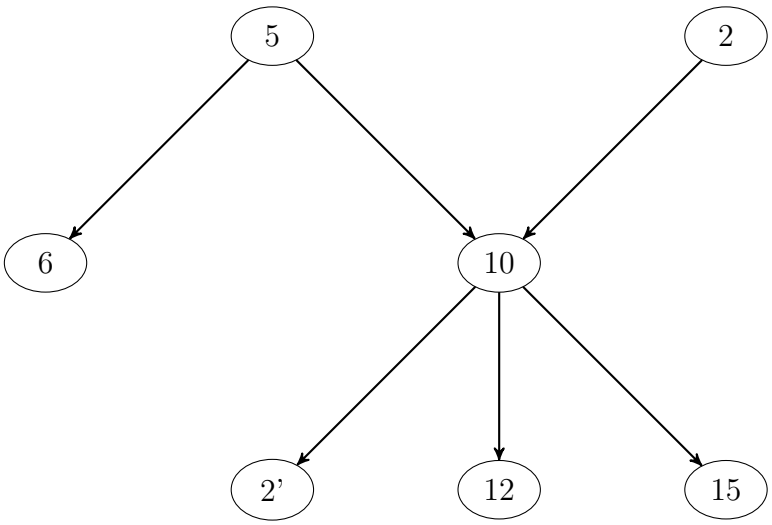
Suppose S = {jobs w/ $p_j \leq \epsilon \times T$} is the set of small jobs. Then $L = J - S$ is the set of large jobs. Then the algorithm is as follows:

- Apply the lemma (the DP algorithm) to L
- If the lemma answers no output no
- Else if the lemma answers yes; then, $\forall j \in S$:
  - If $\exists M_i$ w/ load $\leq T$, add j to M_i.
  - Else return "no schedule with makespan $\leq T$"

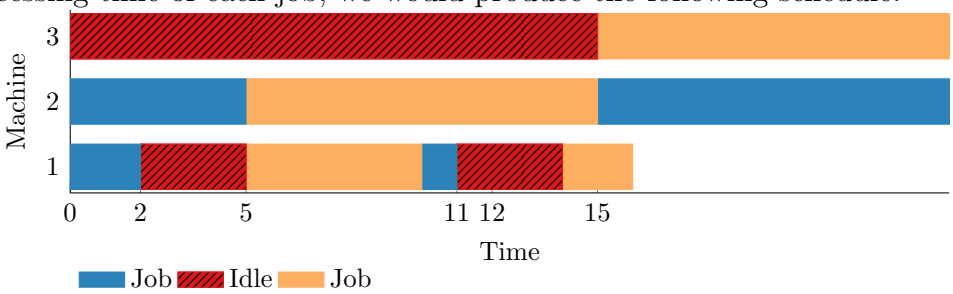
Why does this work?

1. If the DP returns no on L, then there is no schedule with the desired makespan even when we consider only the large jobs, so obviously when we consider the small jobs as well there is still no schedule.
2. If the DP returns yes on L, then we need to fit in the small jobs. If we can fit these in one at a time to machines with load less than T, then after adding the small job each time the load of the small job is at most $\epsilon \times T$ so the machine load with the small job is at most $(1 + \epsilon) \times T$, which is exactly within the desired approximation of the target makespan.
3. If the DP returns yes on L but we have small jobs remaining and no machine has load less than T, then that means that all machines are at full capacity through time T but not all jobs have been scheduled, so obviously no schedule can exist with makespan at most T.

Scheduling with Precedence Constraints



GREEDY We can use list scheduling to approach this problem. When a machine is available, run a job that has all its precedence constraints completed if one exists. For example, for the precedence relation above where the node labels in the graph represent the processing time of each job, we would produce the following schedule:



How closely does that approximate the optimal solution? As before, we want to consider our lower bounds. The lower bounds we discussed previously still apply, but now we also have a third lower bound due to the precedence relation:

- $p_{avg} = \frac{\sum p_i}{m}$
- p_{max}
- p_{chain} is the sum of the job times of the longest (by time) chain of jobs in the directed acyclic graph of the precedence relation

Now let B be the set of times when all machines are busy. Clearly $|B| \leq p_{avg}$ (if $|B| > p_{avg}$ then all machines are busy for more than the average job time, meaning that we've completed more work than is available; that is impossible). But $C_{max} \leq |B| + p_{chain} \leq p_{avg} + p_{chain} \leq 2 \times LB \leq 2 \times OPT$. Thus list scheduling is still a 2-approximation for this problem with

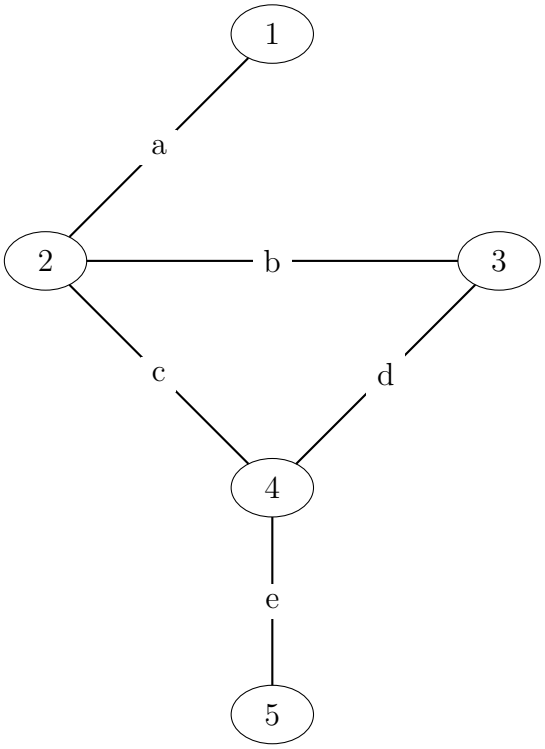
the precedence constraints.

We can also consider minimizing makespan under precedence constraints in the special case where all jobs have length one. $P|prec,\ p_j = 1|C_{max}$ is NP-complete. Also, given an instance of $P|prec,\ p_j = 1|C_{max}$, the question "Is there a schedule with $C_{max} \leq 3$ is NP-complete. As a result, unless $P = NP$, there cannot be any approximation better than a $\frac{4}{3}$-approximation. If there were such an approximation, it would have to return a schedule of lenght 3 if one existed, since returning a schedule of length 4 would be less than a $\frac{4}{3}$-approximation if a schedule of length 3 existed. But this would allow us to solve the NP-complete problem.

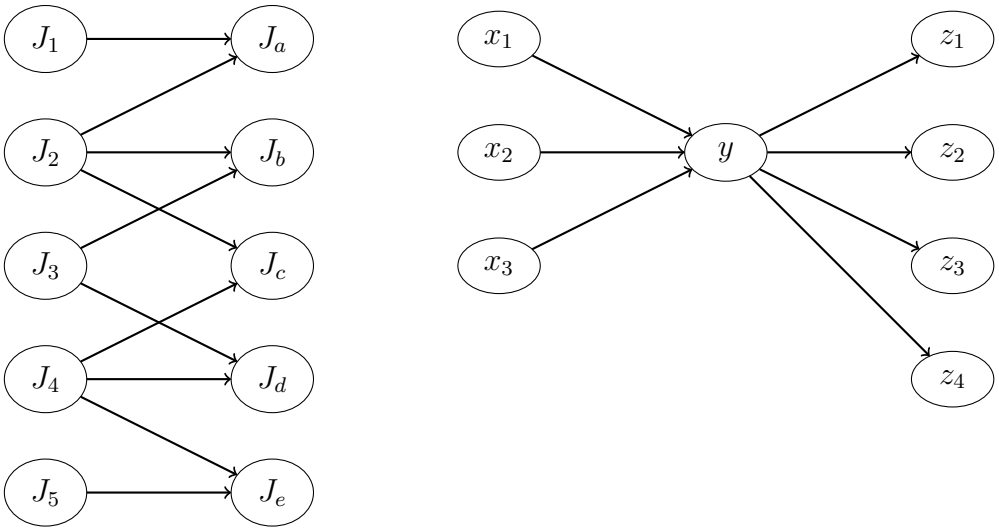
We can show that this problem is NP-complete by reducing from the problem of whether there is a k-clique in a graph, which is known to be NP-complete. Let graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. Then $l = \frac{k \times (k-1)}{2}$ is the number of clique edges; $k' = |V| - k$ is the number of non-clique vertices, and $l' = |E| - l$ is the number of non-clique edges.

Let us then construct an instance of the scheduling problem $P|prec,\ p_j = 1|C_{max}$. Let $m = max(k, l + k', l') + 1$, and let $n = 3m$. Let us create one job per vertex J_v, one job per edge J_e, $m - k$ dummy jobs x_x, $m - l - k'$ dummy jobs y_y, and $m - l'$ dummy jobs z_z. Let the precedence constraints have $J_v \to J_e$ if e is incident to v (edges must be scheduled after the vertices they connect), $(\forall x, y)\ x_x \to y_y$, and $(\forall y, z)\ y_y \to z_z$.

For example, consider this graph:



We would create jobs with precedence constraints as follows:



Proof If there is a k-clique, then we have this schedule:

	Time 1	**Time 2**	**Time 3**
Clique jobs	All k vertices of k-clique	All l edges of k-clique	
Other graph jobs		All k' non-clique vertices	All l' non-clique edges
Dummy jobs	All $m-k$ of x	All $m-l-k'$ of y	All $m-l'$ of z
Total job count	m jobs	m jobs	m jobs

If there is a k-clique, then we can create the schedule described in the table above where the jobs representing vertices of the k-clique and the dummy x jobs go at time 1, the edges of the k-clique, all other vertices of the graph, and the dummy y jobs go at time 2, and all other graph edges and the dummy z jobs go at time 3. If there is no k-clique, then all x dummy jobs have to go at time 1 in order to complete the z jobs by time 3, and some other k vertices can go at time 1 because there are exactly $m - k$ x dummy jobs in this construction. However, because there is no l-clique, we cannot schedule l edges at time 2. If we were able to schedule l edges at time 2, then all the vertices incident to those edges must have been scheduled at time 1, implying there was a k-clique. As a result, we can schedule at most $l - 1$ graph edges, and since there are only k' remaining vertices and exactly $m-l-k'$ of the y dummy jobs that can possibly be scheduled, we can schedule at most $m-1$ jobs. Because we are not using the machines at full load, there is too much load remaining to complete in time 3, so a schedule with makespan at most 3 is impossible without a k-clique.

This shows that there is a schedule for these jobs that has $C_{max} \leq 3$ if and only if there is a k-clique. Because existence of a k-clique in a graph is NP-complete, so is $P|prec,\ p_j = 1|C_{max}$. However, $P_2|prec,\ p_j = 1|C_{max}$ has several polynomial-time exact solutions. We will discuss an example in a later lecture.

It is an open question whether $P_3|prec,\ p_j = 1|C_{max}$, or really $P_n|prec,\ p_j = 1|C_{max}$ for any fixed finite $n > 2$, has a polynomial-time solution or is NP-complete. This is a tantalizing open problem because NP-completeness was first described in 1972; in 1978 Garey and Johnson published a list of commonly discussed algorithms and whether there were polynomial time solutions or they had been shown to be NP-complete or whether it was unknown. Of the ones that were unknown and listed in that list at that time, this problem is the only one that is still unknown. The best approximation right now is a $(1 + \epsilon)$-approximation in running time on the order of $n^{\log n^{O(\log \log n)}}$.