September 15, 2016

Scheduling Algorithms Class Notes

Lecture 3

We began by discussing the open problem $P_m|prec, p_j = 1|C_{max}$ for a fixed number of machines m, recalling that we don't know whether this is in P or NP-complete. This precedence constraint problem remains one of the few unsolved problems in Garey and Johnson's "A Guide to the Theory of NP-Completeness".

What is known about this problem:

- Saw 2-approximation (e.g. List Scheduling, solves even with general processing times)
- We have $\frac{4}{3}$-approx for $m = 3$
- In general for $m \geq 4$, we have a best approximation of $2 - \frac{7}{3m+1}$.
- More so, there's no $2 - \epsilon$ approximation assuming the **Unique Games Conjecture**.

Unique Games Conjecture: Given $\Sigma = \mathbb{Z}_n$ and a set of linear equations mod n, for every $0 < \epsilon < \frac{1}{2}, \exists\ \Sigma$ s.t. there is no poly-time algorithm that, given a Unique Game w/range Σ in which it is posible to satisfy $(1 - \epsilon)$ fraction of the equations, can satisfy ϵ fraction of the equations. (e.g. given a solution that satisfies 90% of the equations, 10% can be satisfied).

Solving this can give better approximation algorithms for many problems, but surprisingly, there is very little idea on whether this is true or not.

Best recent result for the scheduling problem $P_m|prec, p_j = 1|C_{max}$: gets $(1 + \epsilon)$-approximation in $n^{(lgn)^{O(lglgn)}}$ time (this complexity is in between poly and exp). This could be a good area for a research project!

Note that if the number of machines is not fixed, and m is part of the input, this precedence problem is NP-Complete (showed last time).

We then discussed the specific problem with 2 machines: $P_2|prec, p_j = 1|C_{max}$, and its relation to a matching problem. (Note: this algorithm was finalized in Lecture 5).

General concept for algorithm:

- Each job is represented as a node, with precedence constraints as directed edges between them.
- Find matching between nodes as jobs to schedule together.
- At each timestep, find two (if there exist two) jobs whose predecessors have all been scheduled. (Not that there will never be zero such jobs, but there may not always exist two.

Related Machines
$Q|C_{max}$: this problem involves m machines with speeds $v_1, v_2, \dots v_m$ and n jobs with processing times $p_1, p_2, \dots p_n$. Assume that $v_1 \geq v_2 \geq \cdots \geq v_m$ and $p_1 \geq p_2 \geq \cdots \geq p_n$ for convenience.

Running job j on machine i takes $\frac{p_j}{v_i}$ time (rate of processing). For example, given $p_1 = 10, v_1 = 4$, if we run $J1$ on $M1$ for unit of time, 6 units of work still remain.

First, note that the naive idea of List Scheduling is *bad* in this case, because there may be huge differences in the speeds of machines. The most intuitive following solution is to put the job with LPT on the fastest machine. This naive idea gives a 1.2-approx and PTAS.

The pre-emptive case $Q|pmtn|C_{max}$ can be solved in poly time.

Lower bound: $\max\{\max_{1 \leq s < m} \frac{\sum_{j=1}^{s} p_j}{\sum_{i=1}^{s} v_i}, \frac{\sum_{j=1}^{n} p_j}{\sum_{i=1}^{m} v_i}\}$
The idea is that we lower bound by the maximum of the average load, and by the biggest job on the fastest machine.

We can *always* achieve this lower bound C^*_{max} by running LRPT-FM (longest remaining processing time, fastest machine). However, if interpreting this literally, this means there could be a constant switching of jobs between machines to always be using the fastest machine optimally.

Unrelated Machines The pre-emptive $R|pmtn|C_{max}$ is polynomial, but the non-pre-emptive is still NP-Complete (this makes intuitive sense, since the unrelated case just makes the problem harder).

The input to this kind of problem is a matrix, where every cell P_{ij} represents the time it takes machine i to process job j. (∞ is used to denote a pair of machine and job that is not possible.) These values can be interpreted as rate of computation (if we run the first job on the first machine in the below example for 1 unit of time, $\frac{1}{10}$ of the job is done). This matrix represents a problem with 3 machines and 4 jobs.

10	3	∞	10
3	3	5	5
20	3	5	15

We could keep track of the time units that each job runs on each machine in the following way:

$$x_{ijt} = \begin{cases} 1 & \text{if job } j \text{ is on machine } i \text{ from } [t-1, t] \\ 0 & \text{otherwise} \end{cases}$$

We discussed that an exact solution would require variables like these x_{ijt}, which tell us exactly which job runs on which machine at each time step. However, we do not pursue this

approach. Instead, we formulate an LP which only assigns jobs to machines for amounts of time, but does not compute a schedule. The overall plan take the solution to this LP and then uses a separate matching algorithm to convert it to an explict schedule.

Composing an LP:

Define $T = \Sigma p_j$ (our solution may not be poly in T). The LP has the following variables:

t_{ij} = total time that job j is run on machine i.
C_{max} (the value we are trying to minimize).

Additional constraints:

1. Every job runs ($\sum_{i=1}^{m} \frac{t_{ij}}{p_{ij}} = 1 \forall j$)
2. Each machine has $\leq C_{max}$ load assigned to it ($\sum_{j=1}^{n} \leq C_{max} \forall i$)
3. Each job runs for $\leq C_{max}$ total time ($\sum_{i=1}^{m} \leq C_{max} \forall j$).

This LP has nm variables, and $2n + m$ constraints $\rightarrow$ polynomial-time LP. Let t_{ij} be the output of the LP. Note that this output gives us a a matrix of times for corresponding jobs and machines, but does not give us a schedule – for instance, it does not require that a job not be processed on more than one machine at the same time. However, we show that given this output, we *can* always find an optimal schedule.

Theorem. Given a matrix t_{ij}, it is always possible to find a feasible schedule s.t.

$$C_{max} = \max\{\max_i(\sum_j t_{ij}), \max_j(\sum_i t_{ij})\} \tag{1}$$

In other words, if we get our output t_{ij} from the LP, then we get the desired C_{max}.

Proof: This is equivalent to the pre-emptive "open shop" problem $O|pmtn|C_{max}$. Recall that in the shop environment, each job j is made up of operations, with each operation required to run on a specific one of m given machines. Different operations may take different amounts of time (possibly 0). The open shop environment allows the operations of jobs to be processed in any order, as long as no two operations from the same job are processed on different machines at the same time. The general idea for our conversion is to interpret operations in our open shop instance as fractions of the original jobs in the unrelated machine instance. Thus, if we find an open shop schedule, then we get a pre-emptive schedule of the same length in which no two fragments of a job are running at the same time. We do this by creating an operation o_{ij} for each $t_{ij} > 0$ with size $t_{ij}p_{ij}$. We then use a matching algorithm to find an optimal schedule as follows:

We know that both the maximum machine load and job size are bounded from above by C_{max}. We call the maximum machine load Π_{max} and the maximum job size P_{max}, but in this case the values are both equal to C_{max}. At any point in time, any given machine

is processing at most one job, so we aim to find a sequence of matchings that determine which machines are running which jobs at any moment in time. If we process jobs on their corresponding matched machine for some time t, we can then reduce both Π_{max} and P_{max} by t. Then, we recompute the matching and repeat the process, eventually reducing both values to 0.

To aid in this process, we define the following notion of 'tightness', where either a job or a machine is tight if they take up all of the processing time. In other words, if we look at our grid of jobs and machines:

Row i is tight if $\sum_j t_{ij} = C_{max}$.
Column j is tight if $\sum_i t_{ij} = C_{max}$.

For any matching, we must process every tight job and every tight machine, because otherwise we will be unable to reduce Π_{max} and P_{max}. Thus, we are looking for a matching in which every tight machine and job is matched, and that, in addition, each matching edge requires some positive processing time t. This kind of matching is deemed a decrementing set, and we can show that one always exists.

To find such a set, we construct a bipartite graph G with a node representing each job and each machine, and include edges between machine i and job j only if job j requires non-zero processing time on machine i. Again, we do so in a way that matches each tight node.

Once this set has been found, we schedule machines to run on jobs matched to them until either one of the matched jobs becomes completed, or a new job or machine becomes tight. Executing an edge for time t_{ij} reduces both Π_{max} and Π_{max} by t_{ij}.

With every iteration, either a job-machine pair goes to 0 because it has been processed, or either a row or column becomes tight. Note that because once a job or machine becomes tight, it stays tight forever, the number of tight nodes is bounded by $n + m$, which means that constructing this schedule can be done in $nm + n + m$ iterations, which is a polynomial bound for this algorithm that gives us a schedule of optimal length.