IEOR 8100 Scheduling Algorithms

Instructor: Cliff Stein – Scribe: Nouri Sakr

Friday 23rd September, 2016

Introduction

By now we know the following results for scheduling problems where we want to minimize the maximum lateness of any job in the presence of release dates:

- $1 \mid r_j \mid L_{max}$ min max $L_j = C_j + q_j$
- $1 \mid\mid L_{max}$ EDD: largest delivery time first
- $1 \mid r_j \mid L_{max}$ LS: 2-approx

$$\mathbf{L^*_{max}} \geq \mathbf{r_{min}}(\mathbf{S}) + \mathbf{p}(\mathbf{S}) + \mathbf{q_{min}}(\mathbf{S}) \qquad (1)$$

Starting from EDD* (which we define as the greedy algorithm that chooses the largest delivery time among available jobs), we will make observations in order to reach PTAS.

Derivation for PTAS

Consider the following sequence of jobs in a schedule after some idle time:

$$\ldots \mid \text{(idle time)} \mid a \mid z \mid b \mid y \mid c \mid \ldots$$

We refer to c as the **critical job**, i.e. $L_c = L_{max}$. If job a is the first job to run after some idle time and before c, then set S contains all jobs that run between a and c, which we call the **critical section**.

We observe that $\mathbf{L_{max}} = \mathbf{r_a} + \mathbf{p(s)} + \mathbf{q_c}$ and since there is idle time before a, we can deduce that all jobs in S are released after job a, i.e. $\forall \mathbf{j} \in \mathbf{S}, \mathbf{r_j} \geq \mathbf{r_a} = \mathbf{r_{min}}(\mathbf{S})$ (We know that EDD does not allow idle time, unless necessary).

Now suppose that all jobs in S have delivery times larger than that of c: $\forall j \in S$, $\mathbf{q_c} \leq \mathbf{q_j}$ implying that $\mathbf{q_c} = \mathbf{q_{min}}(\mathbf{S})$.

Since c achieves $\mathbf{L_{max}} = \mathbf{r_a} + \mathbf{p}(\mathbf{S}) + \mathbf{q_c}$ and given that $\mathbf{r_a} = \mathbf{r_{min}}(\mathbf{S})$, we get $\mathbf{L_{max}} = \mathbf{r_{min}}(\mathbf{S}) + \mathbf{p}(\mathbf{S}) + \mathbf{q_{min}}(\mathbf{S}) \leq \mathbf{L^*_{max}}$ (by applying the bound in (1)). Hence, if c has the smallest delivery time, then EDD* is optimal.

If EDD* is not optimal, then some job j in S has delivery time smaller than that of the critical job. Let b be the last job in S satisfying $\mathbf{q_b} < \mathbf{q_c}$. (We call this job an **interference job**) Furthermore, define s_j as the starting time of job j.

Claim: $\mathbf{L_{max}} < \mathbf{L^*_{max}} + \mathbf{p_b}$

Proof. Let S' be the jobs processed between end of b through c (including c). Then $\forall j \in S', \mathbf{q_j} \geq \mathbf{q_c} > \mathbf{q_b}$. Note that when EDD* scheduled b at time s_b, no jobs in S' had been released, hence, $\forall j \in S', \mathbf{r_j} > \mathbf{s_b}$.

By equation (1), we know that
$L^*_{max} \geq r_{min}(S') + p(S') + q_{min}(S') > \mathbf{s_b} + \mathbf{p(S')} + \mathbf{q_c}$.
By our construction, we have that $\mathbf{L_{max}} = \mathbf{C_c} + \mathbf{q_c} = \mathbf{s_b} + \mathbf{p_b} + \mathbf{p(S')} + \mathbf{q_c}$.
Plugging the equation derived for L^*_{max} gives us the following inequality:
$\mathbf{L_{max}} < \mathbf{L^*_{max}} + \mathbf{p_b}$. (Bound on p_b gives PTAS) □

This indicates that if interference jobs have very small processing times, EDD* still performs very well.

We now want to reason about a modified optimal schedule to allow us to design PTAS.
The main idea is to handle large jobs separately, then use EDD* for the smaller ones.

Assume we know the optimal schedule on instance I along with s^*_j, the starting time of job j in the optimal schedule, $L^*_{max}(I)$ and $\delta > 0$ (δ will be chosen later).

We create a modified instance $\tilde{I}$ with sets:

- S (small jobs): If $p_j \leq \delta$, then $\tilde{r_j} = r_j$, $\tilde{p_j} = p_j$, and $\tilde{q_j} = q_j$ (copy job)
- B (big jobs): If $p_j > \delta$, set $\tilde{r_j} = s^*_j$, $\tilde{p_j} = p_j$, and $\tilde{q_j} = L^*_{max}(I) - p_j - s^*_j$

Note that we have not decreased any delivery times, so instance $\tilde{I}$ is not easier than I. We just constrained the big jobs to run where they would have run in the optimal schedule. If $j \in B$, then $L_j = \tilde{r_j} + \tilde{p_j} + \tilde{q_j} = L^*_{max}$. Furthermore, we note that $L^*_{max}(I) = L^*_{max}(\tilde{I})$

We run EDD* on our modified instance, $\tilde{I}$. If the critical sequence has no interference job, then the solution is optimal. Otherwise, the following condition holds: $L^*_{max}(\tilde{I}) < L^*_{max} + p_b$

Claim: $\mathbf{b} \notin \mathbf{B}$, i.e. **none of the jobs in B are interference jobs**

Proof. Assume fpoc. that $b \in B$ and b is an interference job, then
$\tilde{\mathbf{q}}_\mathbf{c} > \tilde{\mathbf{q}}_\mathbf{b}$ and $\tilde{\mathbf{r}}_\mathbf{b} = \mathbf{s^*_b}$ imply that $\tilde{\mathbf{r}}_\mathbf{c} > \tilde{\mathbf{r}}_\mathbf{b}$

Now take S'={c, b}, then by (1) we have:
$L^*_{max} \geq r_{min}(S') + p(S') + q_{min}(S') = \tilde{r_b} + p_b + p_c + \tilde{q_b} = L^*_{max} + p_c$
CONTRADICTION. □

Corollary: Let Σ be the schedule from running EDD* on $\tilde{I}$, then by the definition of δ, we get $\mathbf{L_{max}(\Sigma) < L^*_{max} + \delta}$

Goal now: Use this fact to derive PTAS!

PTAS

Let $\delta = \epsilon \sum p_j$, then $L_{max} < L^*_{max} + \epsilon \sum p_j \leq (1+\epsilon) L^*_{max}$ (Recall $\Sigma p_j \leq L^*_{max}$)

How many jobs are in B?
$\leq \frac{1}{\epsilon}$ and we can find them easily

Our only problem is that we actually do not know $\tilde{I}$
We can guess $L^*_{max}(I)$, but we don't know s^*_j for $j \in B$ (too many possibilities).

It is, in fact, sufficient to guess the "position" of a big job in the optimal schedule (k^{th} job to run for some k), rather than knowing its starting time. This may lead us to enumerating all possible ($n^{1/\epsilon}$) numberings for the big jobs. Given a numbering, we can run EDD* on the small jobs, and insert the big jobs at the appropriate places in O(n log n) time. This gives us an O($n^{1+1/\epsilon}$ log n)-algorithm which finds a schedule with $L_{max} \leq (1+\epsilon) L^*_{max}$.

$\mathbf{1 \mid d_j = D \mid \sum w_j C_j}$ (Knapsack)

In the following two cases of single machine, we obtain the optimal schedule using simple rules, namely Shortest Processing Time when minimizing makespan and Smith's Rule when minimizing the weighted sum of completion times.

$1 \mid\mid \sum C_j$

Claim: **SPT sorts by increasing p_j and is optimal**:

j	p_j	w_j
1	5	10
2	2	5
3	7	2

SPT gives 23

$\sum C_j = \sum C_{(j)} = \sum (n-i+1) P_{(i)}$

$1 \mid\mid \sum w_j C_j$

Claim: **SMITH'S RULE sorts by decreasing $\frac{w_j}{p_j}$ and is optimal**

Assume fpoc. that we have an optimal schedule not obeying Smith's Rule, then there exist two consecutive jobs with $\frac{w_j}{p_j} < \frac{w_k}{p_k}$.

SWAP them:
$\sum w_j (C_j - C'_j) = w_k p_j - w_j p_k > 0$

Parallel Machines.

In the setting of parallel machines, a job can be processed on any of the available machines, but shall be processed by one machine at a time. A machine also processes one job at a time.

$P \mid\mid \sum C_j$

In the example below, we are interested in minimizing the sum of completion times. Assume we have 3 machines available to process the following jobs, what would be the optimal schedule?

j	1	2	3	4	5	6	7	8
p_j	1	2	5	6	7	10	15	16

M3	J3		J6
M2	J2	J5	J8
M1	J1	J4	J7

We mentioned earlier that SPT is an exact algorithm for $1 \mid\mid \Sigma C_j$. Given parallel identical machines, the optimal schedule applies SPT here as well. In a sense, it is like assuming we have one machine, but instead of having to schedule, say $J1$, $J2$ and $J3$ on one machine, we assign them in the same order but on separate machines (simultaneously). That is: We sort all jobs in ascending order of their processing times and schedule the jobs on any idle machine. In this example, we run $J1$ on $M1$, $J2$ on $M2$ and $J3$ on $M3$. $J4$ and each subsequent job will then run once its turn has come and one of the three machines has become available. Note that due to SPT, we can guarantee that the job assignment will be balanced among all jobs. That is: The 4^{th} job will never finish before first three, as it (1) has the fourth smallest processing time, (2) will run after one of the three (in particular the first) jobs. Since $J1$ is the job with shortest processing time, $M1$ will be the first machine to become available again and will, hence, process $J4$. The table on the right above, shows the job assignment after applying SPT.

$P \mid\mid \sum w_j C_j$

The problem of minimizing the weighted sum of completion times becomes NP-complete in a parallel machine environment, however, Smith's Rule gives a $\frac{1+\sqrt{2}}{2}$-approx.

$R \mid\mid \sum C_j$

In an unrelated machine environment, machines have different capabilities and thus their relative performance on a job is unrelated. In other words, the processing time of a job becomes both job- and machine-dependent. This polynomial-time algorithms matches each job to its position on one of the machines in order to obtain the optimal schedule.

Construct the following bipartite graph (Figure 1):
1. The left set of n nodes representing n jobs.
2. The right set of nodes represent m sets, a set for each machine.
3. Each set consists of machine copies labeled by position k from last on that machine: "last", "last-1", "last-2", etc.
3. Assign an edge from each job j to each position "last-k" of each machine i with cost (k+1)p_{ij}.
4. Match all jobs to positions s.t. the weighted bipartite matching is minimized

We claim that **a minimum-weight perfect matching in this graph corresponds to an optimal schedule.**

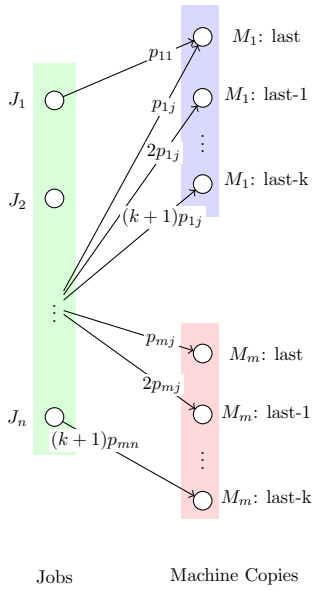

Figure 1: Graph for $R \mid\mid \Sigma C_j$: The nodes on the left (green) represent jobs and the node sets (red and blue) on the right represent machines. Each set has copies of the same machine representing different positions depending on when the job gets executed on that machine - starting from the last position and moving backwards.

The idea here –as illustrated by Figure 1– is to create copies of each machine that are labeled by the position of the job. That is: If machine 1 processes jobs 3, 4 and 7 in that order, then we will match job 7 to the node of machine 1 that represents the last position on machine 1, job 4 to position "last-1" and job 3 to position "last-2". Note that the costs of the arcs are position-dependent because each job has an effect on the entire schedule depending on its position on a certain machine. The claim here is that **each job running in position "last-k" contributes to the sum of completion times by exactly (k+1) times its processing time** and this can be easily verified.

An optimal schedule is in one-to-one correspondence with a minimum weight bipartite matching for the model in Figure 1. Note that no optimal matching will "skip" positions, i.e. cannot –for instance– assign a job to "last-1" on a machine if "last" is not already occupied simply because if "last" were not occupied, then there would exist an optimal matching of less cost where we simply switch the edge to point to the "last" position instead. The argument holds for any "skipped" position, so that we can always keep "pushing" positions with the gain of minimizing cost, until we have no more "gaps".