Scheduling to Minimize Total Response Time

Cliff Stein
Columbia University

Why teach Scheduling to Minimize Total Response Time?

- It is the most important algorithmic problem in the world

Why teach Scheduling to Minimize Total Response Time?

- It is the most important algorithmic problem in the world

In reality

- It is a basic algorithmic problem.
- It is a special case of many practical problems.
- It will give us the opportunity to study several different algorithic techniques, and show several ways of attacking the same problem.

Non-Preemptive Min-Sum Scheduling

Consider the following basic scheduling problem

- **1 machine**
- n **jobs, job** j **has**
  - **release date** r_j
  - **processing time** p_j

- **A non-preemptive schedule assigns each job to a time interval of size** p_j **ending at time** C_j **.**
- **Flow (response) time** $F_j = C_j - r_j$
- **Objective: minimize** $\sum_j F_j$ **(min-sum)**

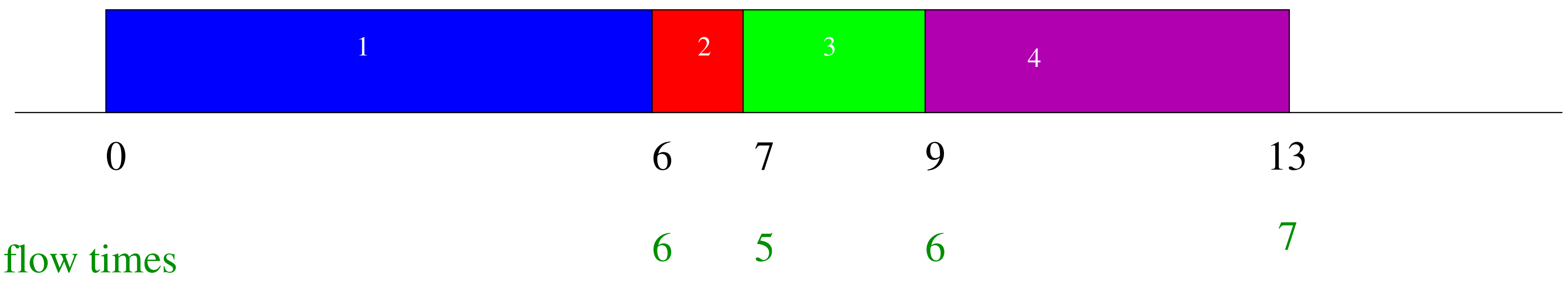
Example

j	1	2	3	4
r_j	0	2	3	6
p_j	6	1	2	4

Example

j	**1**	**2**	**3**	**4**
r_j	**0**	**2**	**3**	**6**
p_j	**6**	**1**	**2**	**4**

One schedule

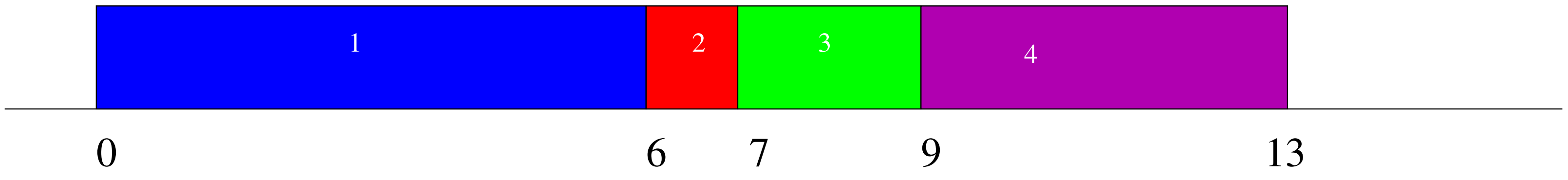


total flow time = 24

Example

j	1	2	3	4
r_j	0	2	3	6
p_j	6	1	2	4

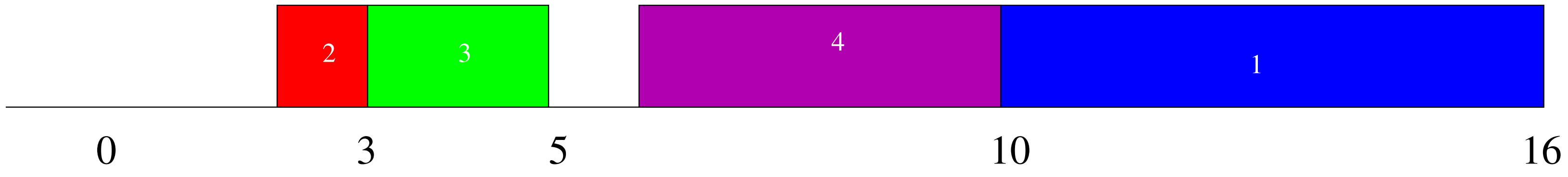
One schedule



flow times 6 5 6 7

total flow time = 24

A better schedule



flow times 1 2 4 16

total flow time = 23

What do we Know About this Problem?

- Response time is a common metric in many applications.
- Used especially in operating systems, and other real-time systems.

What about algorithms?

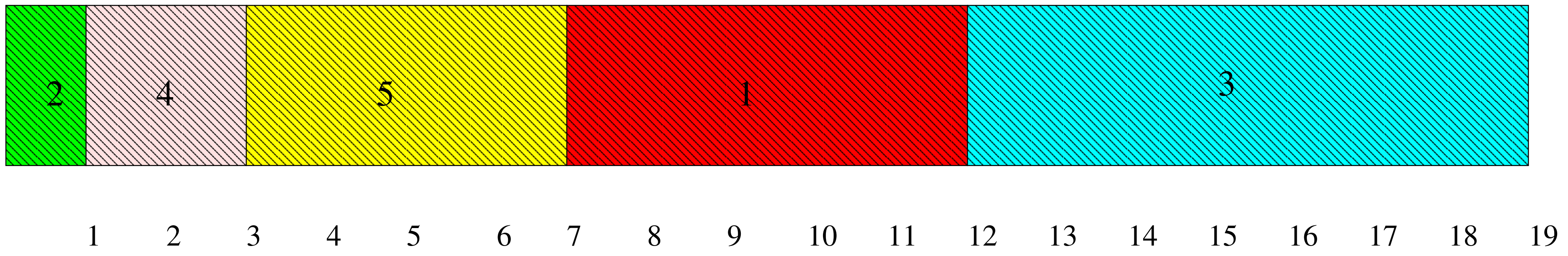
If all release dates are 0: EASY

Use **SPT, Shortest Processing Time first**

Input:

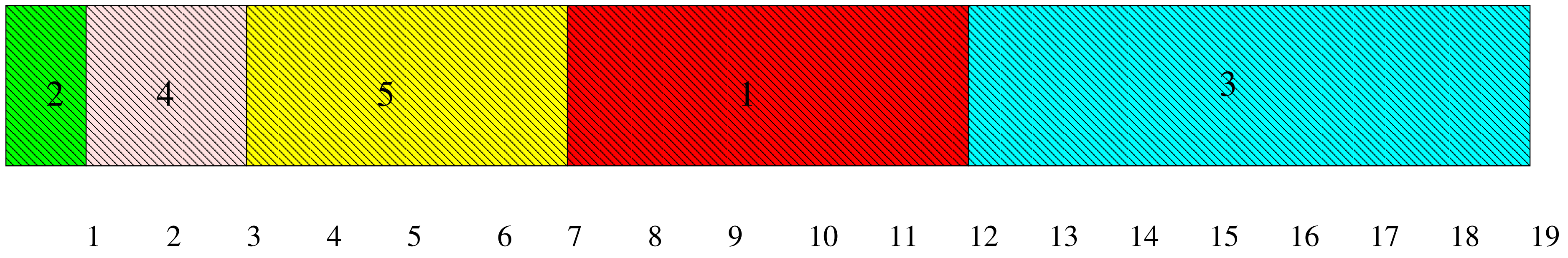
j	1	2	3	4	5
r_j	0	0	0	0	0
p_j	5	1	7	2	4

Output:



C1+C2+C3+C4+C5 = 42

Proof that SPT is optimal



C1+C2+C3+C4+C5 = 42

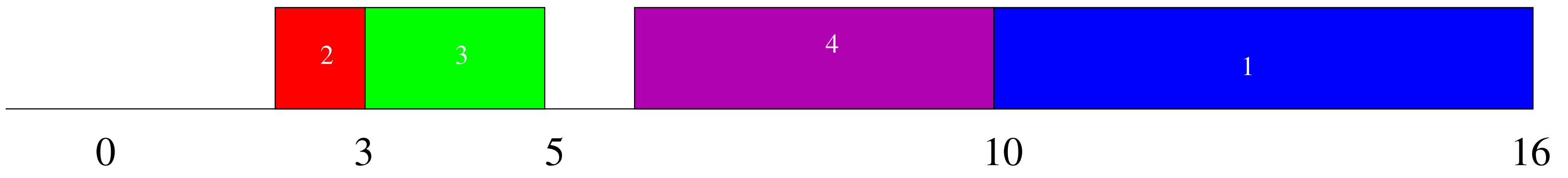
$$\begin{aligned}\sum F_j &= \sum C_j \\ &= p_2 + (p_2 + p_4) + (p_2 + p_4 + p_5) \\ &\quad +(p_2 + p_4 + p_5 + p_1) + (p_2 + p_4 + p_5 + p_1 + p_3) \\ &= 5p_2 + 4p_4 + 3p_5 + 2p_1 + p_3\end{aligned}$$

If we allow preemption: EASY

Use **SRPT, Shortest Remaining Processing Time First**

j	**1**	**2**	**3**	**4**
r_j	**0**	**2**	**3**	**6**
p_j	**6**	**1**	**2**	**4**

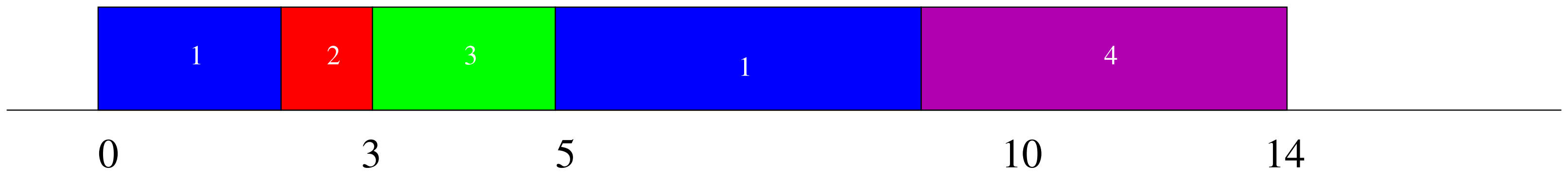
Non preemptive schedule



flow times 1 2 4 16

total flow time = 23

Preemptive schedule



flow times 1 2 10 8

total flow time = 21

Back to the original problem

- 1 machine
- n jobs, job j has
  - release date r_j
  - processing time p_j

Approximation Algorithms

A ρ -approximation algorithm (for a minimization problem) is an algorithm which, in polynomial time, finds a solution whose value is no more than ρ times the value of the optimal solution.

A polynomial time approximation scheme (PTAS) is an algorithm that, for any fixed $\epsilon > 0$, is a $1 + \epsilon$ -approximation algorithm.

BAD NEWS:

No $o(\sqrt{n})$ approximation for non-preemptively scheduling to minimize total flow time unless P=NP [Kellerer, Tautenhahn & Woeginger 96].

Two approaches to this problem:

1. **Change the objective**
2. **Change the comparison rules**

First attack:

Change the objective to average completion time

- Well-studied, basic measure
  - in theory and in practice
  - approximation algorithms and exact solutions
- Measures, in some sense, average response and “fairness”

Disadvantages:

- It is not flow time , $F_j = C_j - r_j$.
- It is not stretch , F_j/p_j .

Note: Exact optimization of $\sum F_j$ and $\sum C_j$ are equivalent, approximation is not.

3 algorithms for $\sum C_j$

- 2 -approximation
- Randomized $e/(e-1)$ -approximation
- Polynomial time approximation scheme (($1+\epsilon$)-approximation)

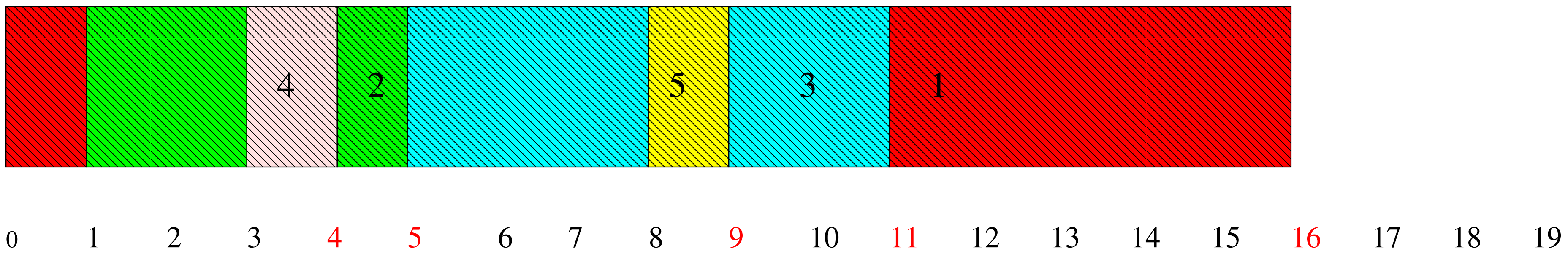
$1|r_j|\sum C_j$

[Phillips, Stein, Wein '95]

Sample input:

j	1	2	3	4	5
r_j	0	1	1	3	8
p_j	6	3	5	1	1

Step 1: Find the optimal **preemptive schedule** via **Shortest Remaining Processing Time** algorithm. [Baker '74]



$$\sum C_j^P = 4 + 5 + 9 + 11 + 16 = 45$$

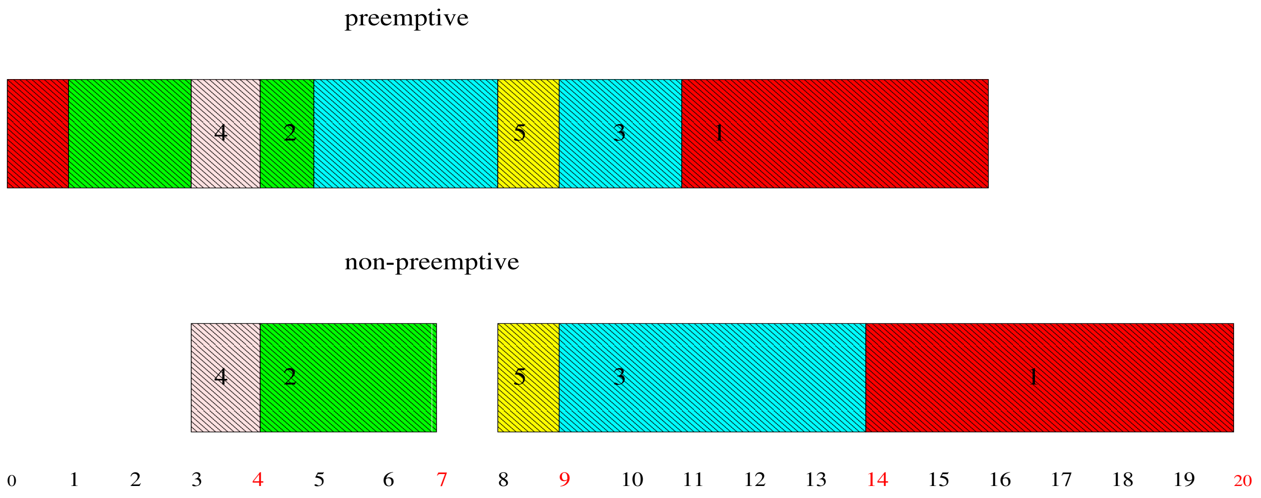
(Clearly $\sum C_j^P \le OPT$)

2-approximation algorithm

Sample input:

j	1	2	3	4	5
r_j	0	1	1	3	8
p_j	6	3	5	1	1

Step 2: **Non-preemptively schedule the jobs in the order they complete in the preemptive schedule, respecting release dates.**



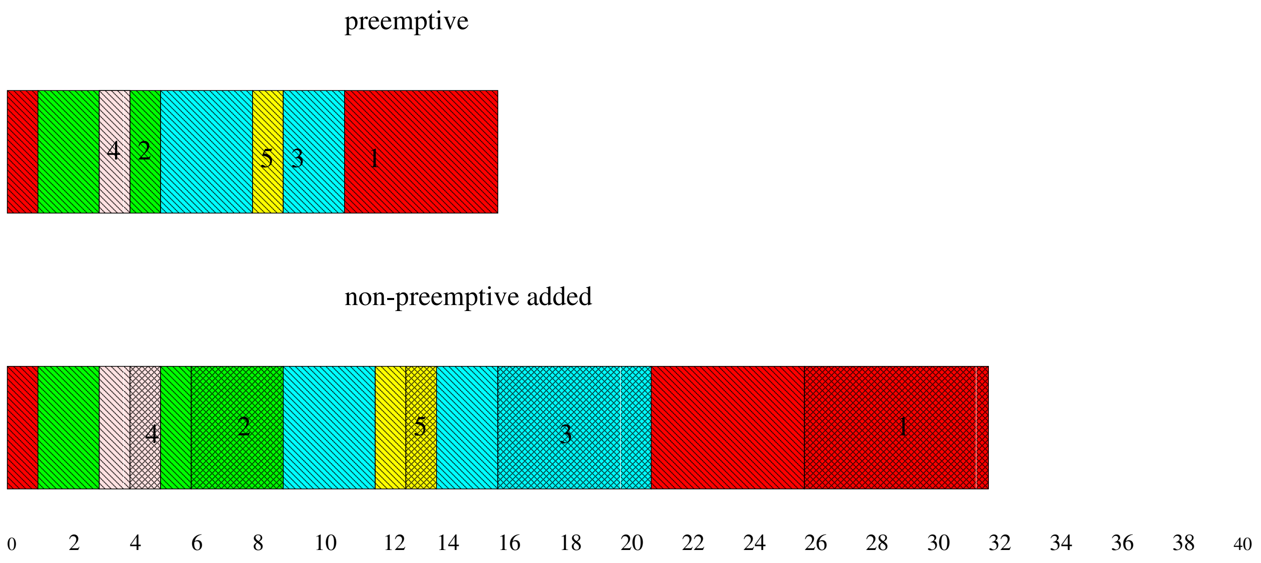
$$\sum C_j^P = 4 + 5 + 9 + 11 + 16 = 45$$

$$\sum C_j^N = 4 + 7 + 9 + 14 + 20 = 54$$

Proof of 2-approximation

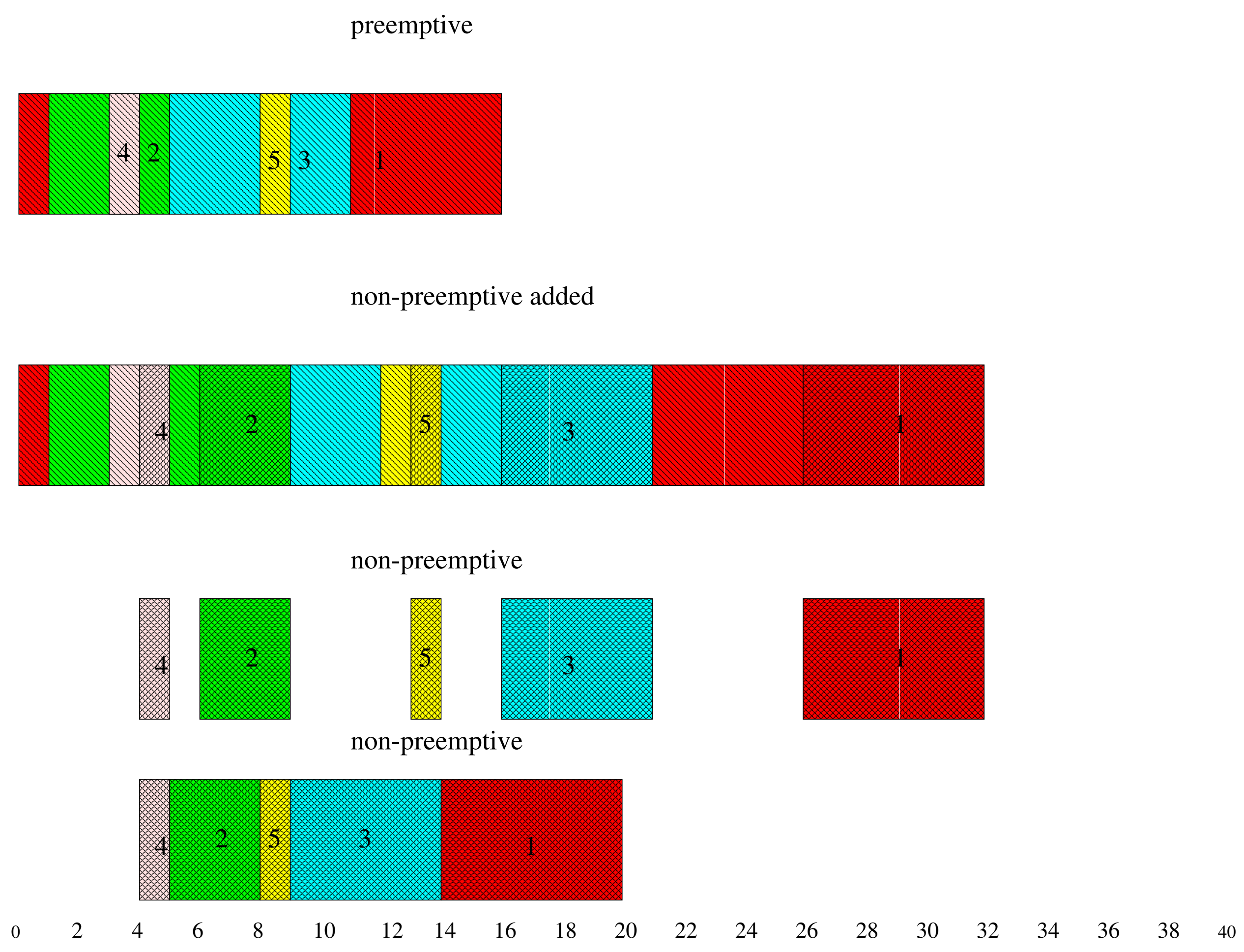
Lemma: **For each job,** $C_j^N \leq 2C_j^P$ **.**

Proof: **Take the preemptive schedule, and when job j completes, insert the job non-preemptively.**



In this schedule, the completion time of a job at most doubles!

Turn into a valid schedule



So, we have a valid schedule and the completion time of each job at most doubled.

Proof with symbols

- Index the jobs by C_j^P
- Let $r_j^{'} = \max_{1 \le k \le j}\{r_k\}$

Two lower bounds:

- $C_j^P \ge r_j^{'}$
- $C_j^P \ge \sum_{k=1}^{j} p_k$

After time $r_j^{'}$, all of jobs 1 through j are available, so:

$$\begin{aligned} C_j^N &\le r_j^{'} + \sum_{k=1}^{j} p_k \\ &\le C_j^P + C_j^P \\ &= 2C_j^P \end{aligned}$$

For whole schedule:

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

Generalization

Recap of algorithm for 1 machine, release dates

1. Solve the preemptive schedule to obtain preemptive completion times C_j^P .
2. Schedule the jobs in the order given by C_j^P, respecting release dates.

General Framework

1. Solve a relaxation of the given problem in order to obtain an ordering on the jobs.
2. Schedule the jobs according to the ordering, respecting constraints.

We will see other examples of this framework shortly.

Can we get a better bound

No deterministic on-line algorithm can do better than a factor of 2 [Vestgens '95]!

Need randomization or off-line algorithms.

Alpha points

([Hall et. al. '96, Phillips, Stein, Wein '95, Goemans '97, Chekuri et. al. '97])

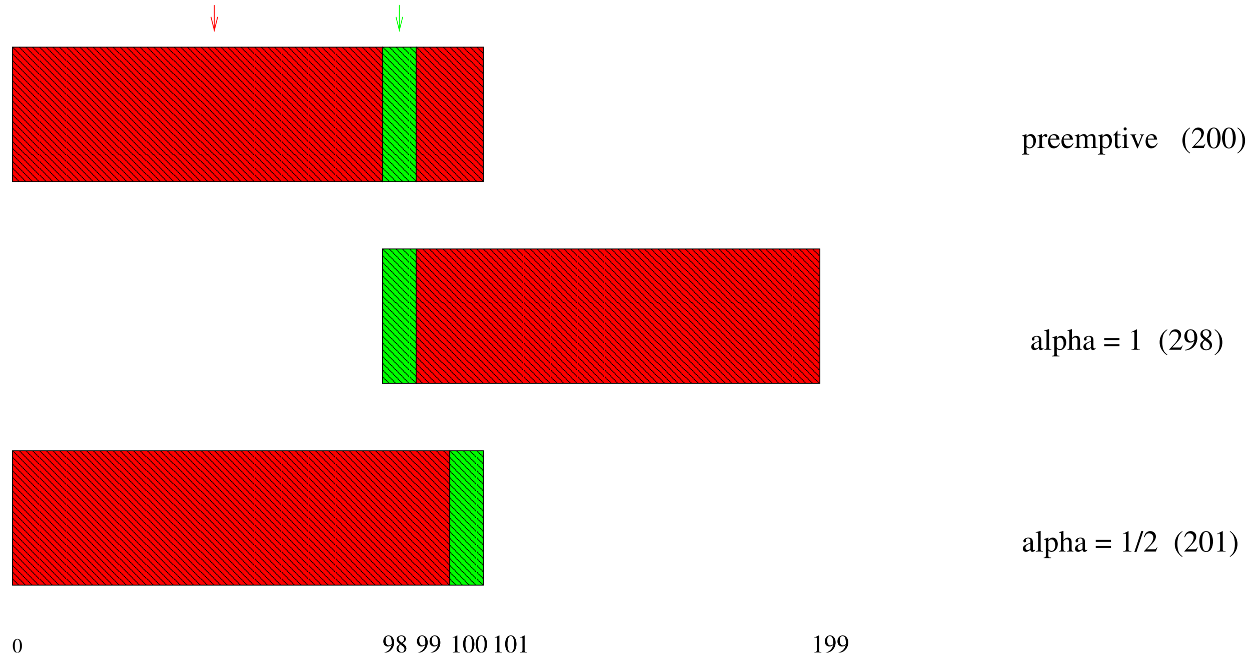
Alpha point: Let C_j^α be the earliest time at which αp_j of job j has completed.

Idea: Schedule in order of α points.

Sample input:

j	r_j	p_j
1	0	100
2	98	1

Schedules:



Intuition: α**-points can avoid "bad" case.**

Scheduling by α-points

Schedule-by-α

1. **Choose** $\alpha \in [0,1]$
2. **Solve the preemptive schedule to obtain preemptive completion times** C_j^α **.**
3. **Schedule the jobs in the order given by** C_j^α, **respecting release dates.**

Theorem: **Schedule-by-α is a $\left(1+\frac{1}{\alpha}\right)$-approximation algorithm.**

Analysis of Schedule by α

Theorem: **Schedule-by-α is a $\left(1+\frac{1}{\alpha}\right)$-approximation algorithm.**

Proof: **(Similar to 2-approximation)**

- **Index the jobs by** C_j^α
- **Let** $r_j' = \max_{1 \le k \le j}\{r_k\}$

Two lower bounds:

- $C_j^P \ge r_j'$
- $C_j^P \ge \alpha \sum_{k=1}^{j} p_k$

After time r_j' **, all of jobs** 1 **through** j **are available, so:**

$$\begin{aligned} C_j^N &\le r_j' + \sum_{k=1}^{j} p_k \\ &\le C_j^P + \frac{1}{\alpha} C_j^P \\ &= \left(1+\frac{1}{\alpha}\right) C_j^P \end{aligned}$$

Scheduling by α-points

Schedule-by-α

1. Choose $\alpha \in [0,1]$
2. Solve the preemptive schedule to obtain preemptive completion times C_j^α .
3. Schedule the jobs in the order given by C_j^α , respecting release dates.

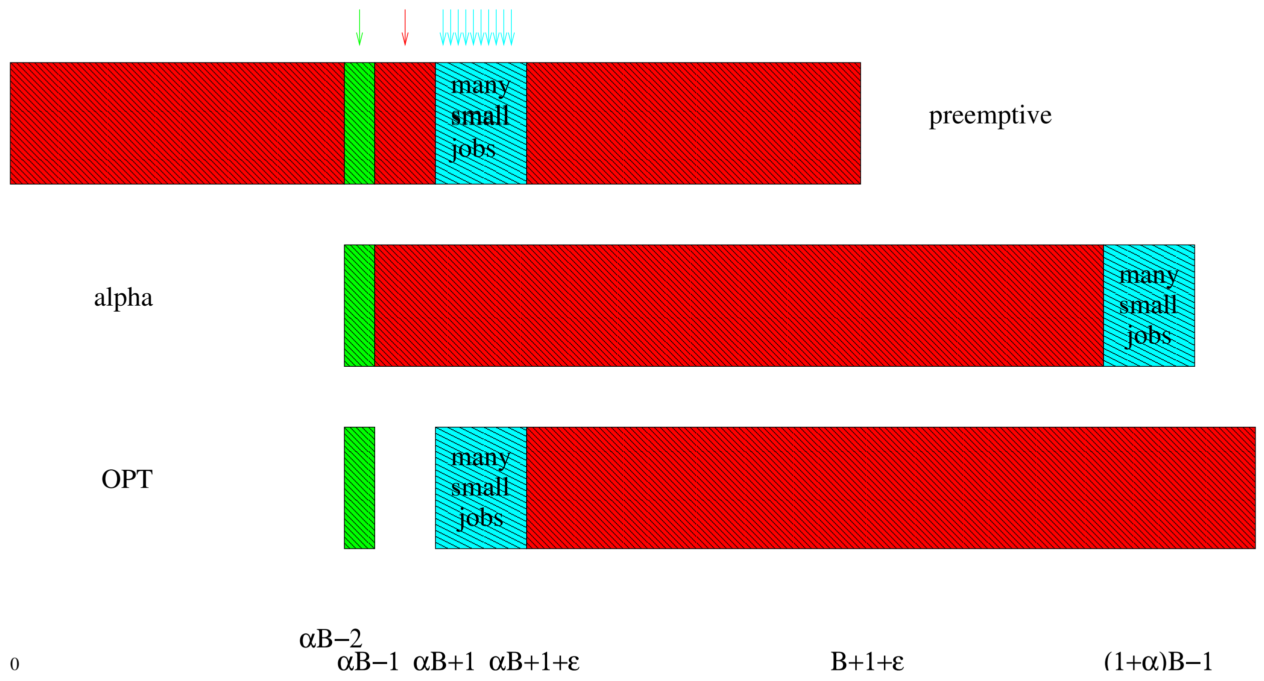
Theorem: Schedule-by-α is a $\left(1+\frac{1}{\alpha}\right)$-approximation algorithm.

Best α to choice is 1 , yielding a 2-approximation!

Analysis is tight

For any α, there is an input for which the algorithm produces a schedule that is $1+\frac{1}{\alpha}$ off from optimal. Sample input:

j	r_j	p_j
1	**0**	**B**
2	$\alpha B - 2$	**1**
$3 \dots x$	$\alpha B + 1$	ϵ



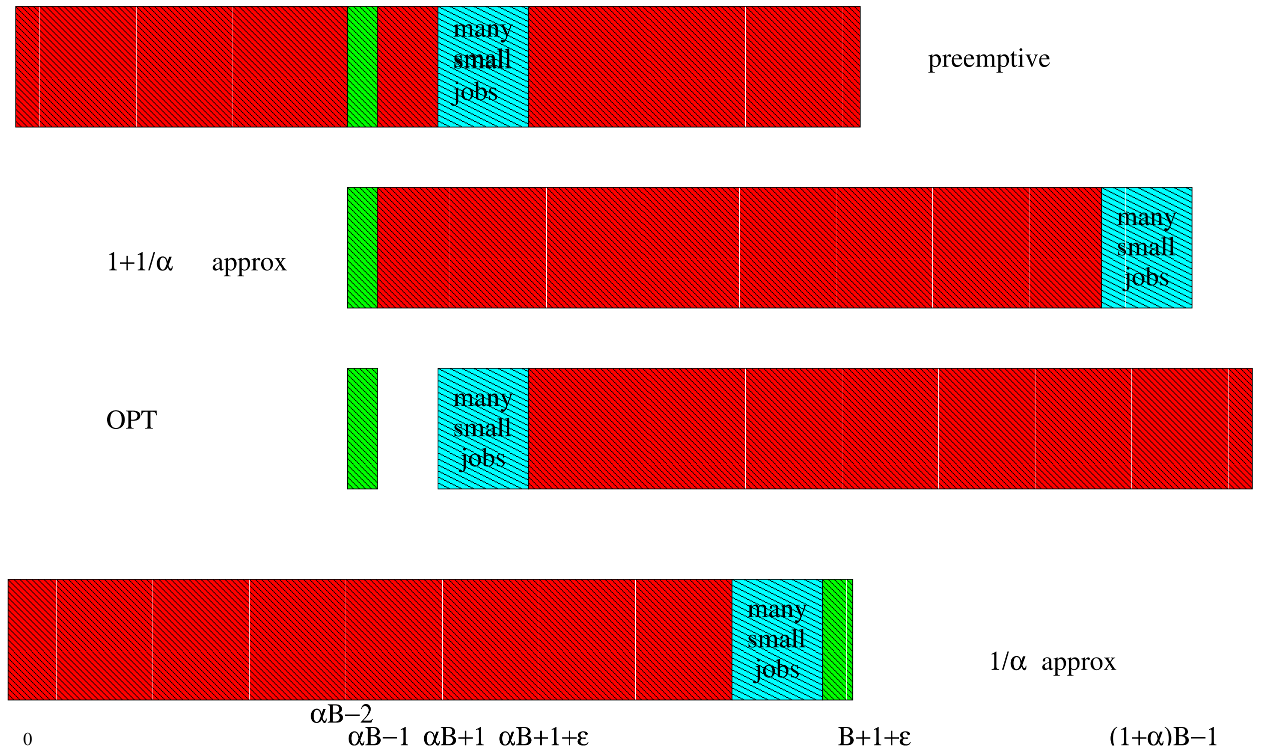
The many small jobs have complete at roughly $\frac{(1+\alpha)B}{\alpha B} = 1 + \frac{1}{\alpha}$ **times OPT.**

Insight for improvement

For any α, there is an input that can be as bad as $1+\frac{1}{\alpha}$

BUT

for any input, most α's yield significantly better schedules.



Randomize to avoid worst case

Schedule-by-random-α

1. Choose $\alpha \in [0,1]$ according to some probability distribution.
2. Solve the preemptive schedule to obtain preemptive completion times C_j^α.
3. Schedule the jobs in the order given by C_j^α, respecting release dates.

Sketch of Analysis

- Let $S_j(\beta)$ denote the set of jobs which complete exactly a β-fraction of their processing before C_j^P, in the preemptive schedule.
- Let $p(S_j(\beta)) = \sum_{j \in S_j(\beta)} p_j$.
- Let T_j be the idle time before C_j^P .

Lower bound:

$$\begin{aligned} C_j^P &= T_j + \sum_{0<\beta\le 1} \beta p(S_j(\beta)) \\ &= T_j + \sum_{0\le\beta<\alpha} \beta p(S_j(\beta)) + \sum_{\alpha\le\beta\le 1} \beta p(S_j(\beta)) \end{aligned}$$

Upper bound:

$$C_j^N \le T_j + \sum_{0\le\beta<\alpha} \beta p(S_j(\beta)) + (1+\alpha) \sum_{\alpha\le\beta\le 1} p(S_j(\beta))$$

Theorem:

If we choose α from a p.d.f $f(\alpha)$, then Schedule-by-random-α is a

$$1 + \max_{0<\beta\le 1} \int_0^\beta \frac{1+\alpha-\beta}{\beta} f(\alpha) d\alpha$$

approximation algorithm. (approximation ratio is an expected value.)

Analysis Continued

Corollary: **If we choose** $f(\alpha) = \frac{e^\alpha}{e-1}$ **, then Schedule-by-random-α is an** $\frac{e}{e-1} \approx 1.58$ **- approximation algorithm and it runs in** $O(n \log n)$ **time.**

- **Can derandomize algorithm by choosing all** $n-1$ **combinatorially distinct α-points.**
  ($O(n^2 \log n)$ **time.)**
- **Is the resulting algorithm (Best-α) any better?**

Analysis Continued

Corollary: If we choose $f(\alpha) = \frac{e^\alpha}{e-1}$, then **Schedule-by-random-α** is an $\frac{e}{e-1} \approx 1.58$ -approximation algorithm and it runs in $O(n \log n)$ time.

- Can derandomize algorithm by choosing all combinatorially distinct α-points. ($O(n^2 \log n)$ time.)
- Is the resulting algorithm (**Best-α**) any better

NO, in the worst case

Theorem: No Ordering Rule that starts with the optimal preemptive (SRPT) schedule can do better than $e/(e-1)$ [Torng, Uthaisombut '99].

Experimental Results

([Savelsberg, Uma, Wein, '98])

	arrival rate = 2								
(p_j, w_j)	$schedule-by-\bar{C}_j$			$shcedule-by-fixed-\alpha$			$best-\alpha$		
	Mean	Std.Dev.	Max	Mean	Std.Dev.	Max	Mean	Std.Dev.	Max
(1,1)	1.360	0.186	1.781	1.271	0.105	1.480	1.114	0.049	1.199
(1,2)	1.351	0.190	1.842	1.250	0.111	1.455	1.101	0.044	1.179
(1,3)	1.400	0.196	1.833	1.262	0.119	1.549	1.147	0.094	1.407
(2,1)	1.307	0.119	1.530	1.210	0.076	1.359	1.086	0.021	1.140
(2,2)	1.247	0.111	1.515	1.205	0.082	1.352	1.084	0.035	1.152
(2,3)	1.262	0.120	1.518	1.187	0.068	1.289	1.100	0.070	1.276
(3,1)	1.341	0.207	1.810	1.246	0.141	1.622	1.129	0.089	1.367
(3,2)	1.305	0.154	1.645	1.238	0.120	1.548	1.095	0.057	1.258
(3,3)	1.287	0.147	1.612	1.173	0.096	1.303	1.079	0.053	1.198

Table 1: 50 jobs $\sum w_j F_j$. Performance of C_j-based heuristics, compared to C_j-based lower bound.

(n,a)	Schedule-by-$\bar{C}_j$			Schedule-by-Fixed-α			Best-α		
	Mean	Std.Dev.	Max	Mean	Std.Dev.	Max	Mean	Std.Dev.	Max
(50,2)	1.271	0.144	1.729	1.184	0.092	1.495	1.066	0.047	1.282
(50,5)	1.107	0.045	1.245	1.073	0.032	1.197	1.018	0.013	1.071
(100,2)	1.268	0.125	1.786	1.190	0.096	1.686	1.071	0.043	1.271
(100,5)	1.079	0.030	1.152	1.056	0.018	1.104	1.014	0.010	1.055

Table 2: Performance of algorithms applied to solution of C_j-relaxation for $\sum w_j F_j$. We report on ratio of algorithm performance to x_{jt}-relaxation lower bound.

General Framework on other problems

General Framework

1. Solve a relaxation of the given problem in order to obtain an ordering on the jobs.
2. Schedule the jobs according to the ordering.

Problems:

- Only polynomial time solvable problem is one machine, release dates and preemption.
- Multiple machines create more complicated orderings
- Precedence constraints

Solutions

- All can be handled via various relaxations including:
  - One machine relaxations
  - A variety of linear programs
- Judicious use of ordering rules such as α-points, idependetly chosen α's for each job, rules about assigning jobs to machines.

One Example – $1|r_j, prec|\sum C_j$

Use a linear programming relaxation[Hall et. al. '96, Dyer Wolsey '91]

Variables: $y_{jt} = 1$ **if job j completes at time t**

$$\min \sum_{j=1}^{n} \sum_{t=1}^{T} t y_{jt}$$

subject to

$$\begin{array}{llll}
\displaystyle\sum_{t=1}^{T} y_{jt} = 1 & j = 1 \ldots n & \textbf{jobs run} \\
y_{jt} = 0 & \textbf{if } t < r_j + p_j & \\
\displaystyle\sum_{j=1}^{n} \sum_{s=t}^{t+p_j-1} y_{js} \leq 1 & t = 1 \ldots T & \textbf{machine} \\
\displaystyle\sum_{s=1}^{t} y_{js} \geq \sum_{s=1}^{t+p_k} y_{ks} & \textbf{if } j \rightarrow k, t = 1 \ldots T - p_k & \textbf{prec}
\end{array}$$

LP is lower bound, but how do we get ordering?

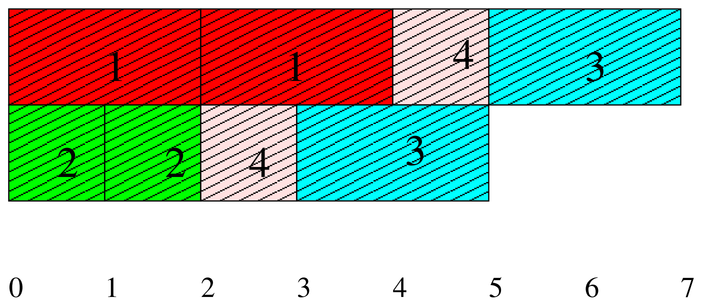
Example

Sample input:

j	r_j	p_j
1	**0**	**2**
2	**0**	**1**
3	**0**	**2**
4	**0**	**1**

$$\begin{tikzcd}
1 \arrow[d] & 2 \arrow[dl] \arrow[d] \\
3 & 4
\end{tikzcd}$$

Solution to LP:



(1/2 point) 2 1 4 3

Variables:

$$y_{21} = y_{12} = y_{22} = y_{43} = y_{14} == y_{35} = y_{45} = y_{37} = \frac{1}{2}$$

Algorithm and Analysis

General Framework

1. Solve the LP relaxation of the given problem to obtain $1/2$-points, the earliest point at which half of processing is done.
2. Schedule the jobs according to the ordering for the $1/2$-points, respecting release dates.

- release dates are obeyed
- precedence constraints are obeyed
- 5.33-approximation (better approx. is possible)
- non-polynomial size can be handled

There are many other $O(1)$ approximation algorithms for r

Problem	non-PTAS	PTAS
$1\|r_j\|\sum C_j$	1.58[1]	yes
$1\|r_j\|\sum w_jC_j$	1.69[2]	yes
$1\|r_j$, pmtn $\|\sum \mathrm{w_jC_j}$	4/3[3]	yes
$P\|r_j\|\sum w_jC_j$	2[4]	yes
$P\|r_j$, pmtn $\|\sum \mathrm{w_jC_j}$	2[4]	yes
$Rm\|r_j\|\sum w_jC_j$	2[5]	yes
$Rm\|r_j$, pmtn $\|\sum \mathrm{w_jC_j}$	3[5]	yes
$Rm\|\,\|\sum w_jC_j$	3/2[6]	yes
$R\|r_j\|\sum C_j$	2[4]	no
$1\|\prec\|\sum w_jC_j$	2[7]	no
$1\|r_j,\prec\|\sum w_jC_j$	e[3]	no
$P\|\prec\|\sum w_jC_j$	4[8]	no

1. **Chekuri et. al., 1997**
2. **Goemans et. al., 1999**
3. **Schulz, Skutella, 1999**
4. **Schulz, Skutella, 1997**
5. **Skutella, 1999**
6. **Skutella, 1999**
7. **Chekuri, Motwani, 1999**
8. **Munier Queyranne, Schulz, 1998**
9. **Hall, et. al. , 1997**
10. **Goemans, 1997**
11. **Chakrabarti et. al., 1997**

Can we do better?

Yes, there is a PTAS for the 1 machine problem, and several related problems.

One negative result: [Hoogeveen, Schuurman, Woeginger '98] Unless $P = NP$, there is no PTAS for:

- $R|r_j|C_j$
- $R||\sum w_jC_j$
- $P|prec, p_j = 1|\sum C_j$

PTAS

A PTAS for $1|r_j|\sum C_j$, with running time $O(n\log n + 2^{poly(1/\epsilon)})$.

(Present algorithm of Karger and Stein other algorithms, results obtained by [Afrati, Bampis, Chekuri, Kenyon, Khanna, Milis, Queyranne, Skutella, Sviridenko])

Highlights

- Simple algorithm – rounding, enumeration, and Shortest processing time (SPT).
- Enumeration is only at end of schedule.
- ϵ term is additive not multiplicative.

Approach:

- Give a series of transformations, each increasing $\sum C_j$ by at most a $1+\epsilon$ factor. (Blowing up the schedule by a $1+\epsilon$-factor is "free".)
- Resulting instance is solvable by SPT and enumeration.

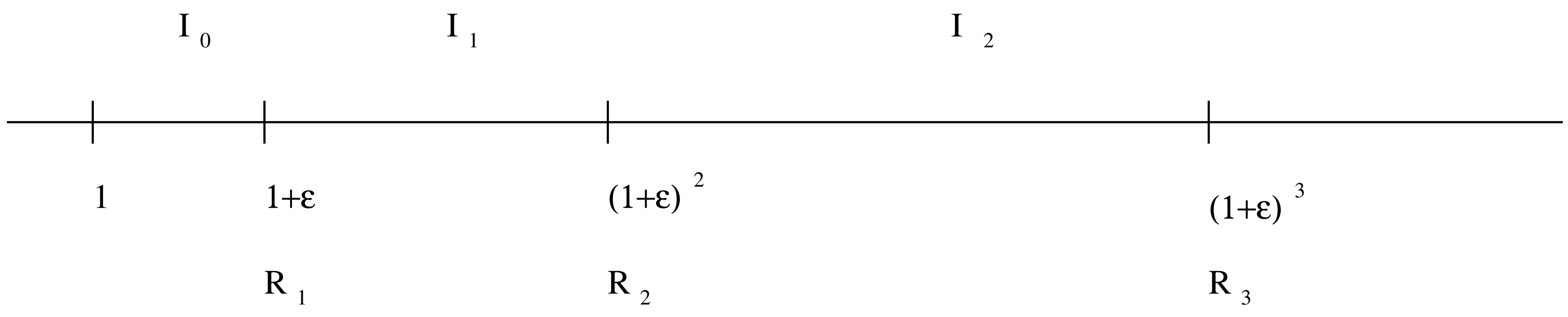
$$(1+\epsilon)^k = 1 + O(\epsilon), \text{ for constant } k$$

Transformation 1: Geometric Rounding

We can round release dates and processing times to be powers of $1+\epsilon$.

This simplifies things:

- **Let $R_x = (1+\epsilon)^x$ be the release dates.**
- **Define intervals $I_x = [R_x, R_{x+1})$.**
  (Note that $|I_x| = R_{x+1} - R_x = \epsilon R_x$.)

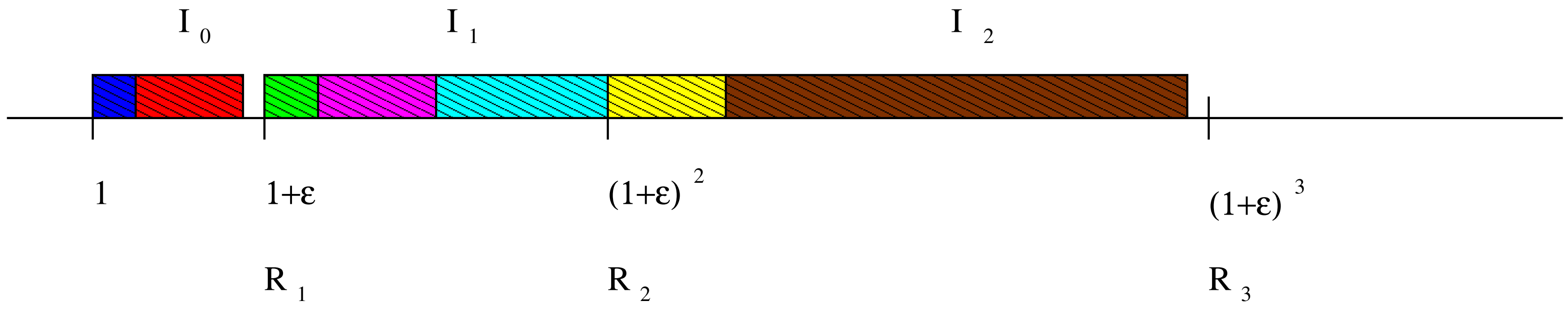


Observations:

- **Fewer decision points (release dates)**
- **Could also round up completion times C_j. This yields a problem of packing jobs to intervals.**

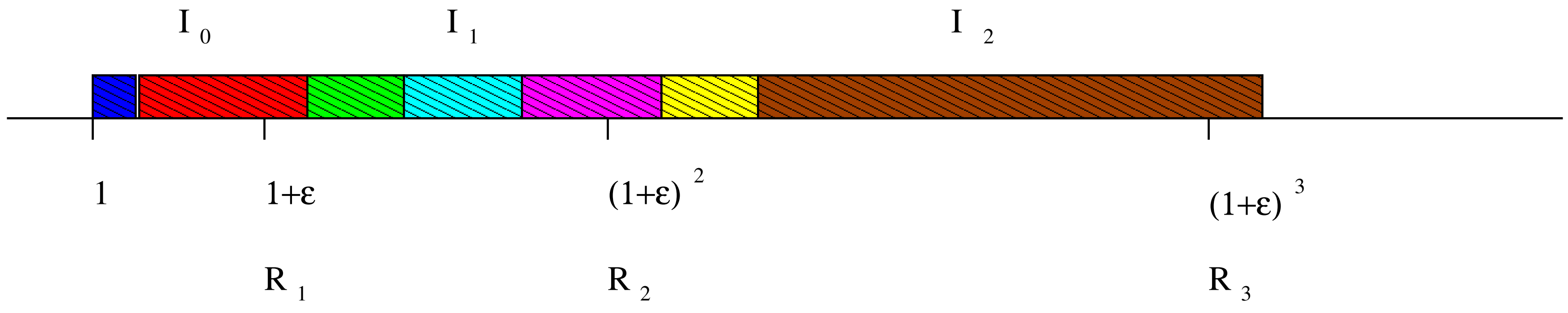
Intuition: SPT is often good

If no job happens to be running at any release date, SPT is optimal.



Proof In this case, SRPT = SPT. SRPT is the optimal preemptive schedule, and this is no more than the optimal non-preemptive schedule.

If no job has much remaining processing time at any release date, SPT is nearly optimal.



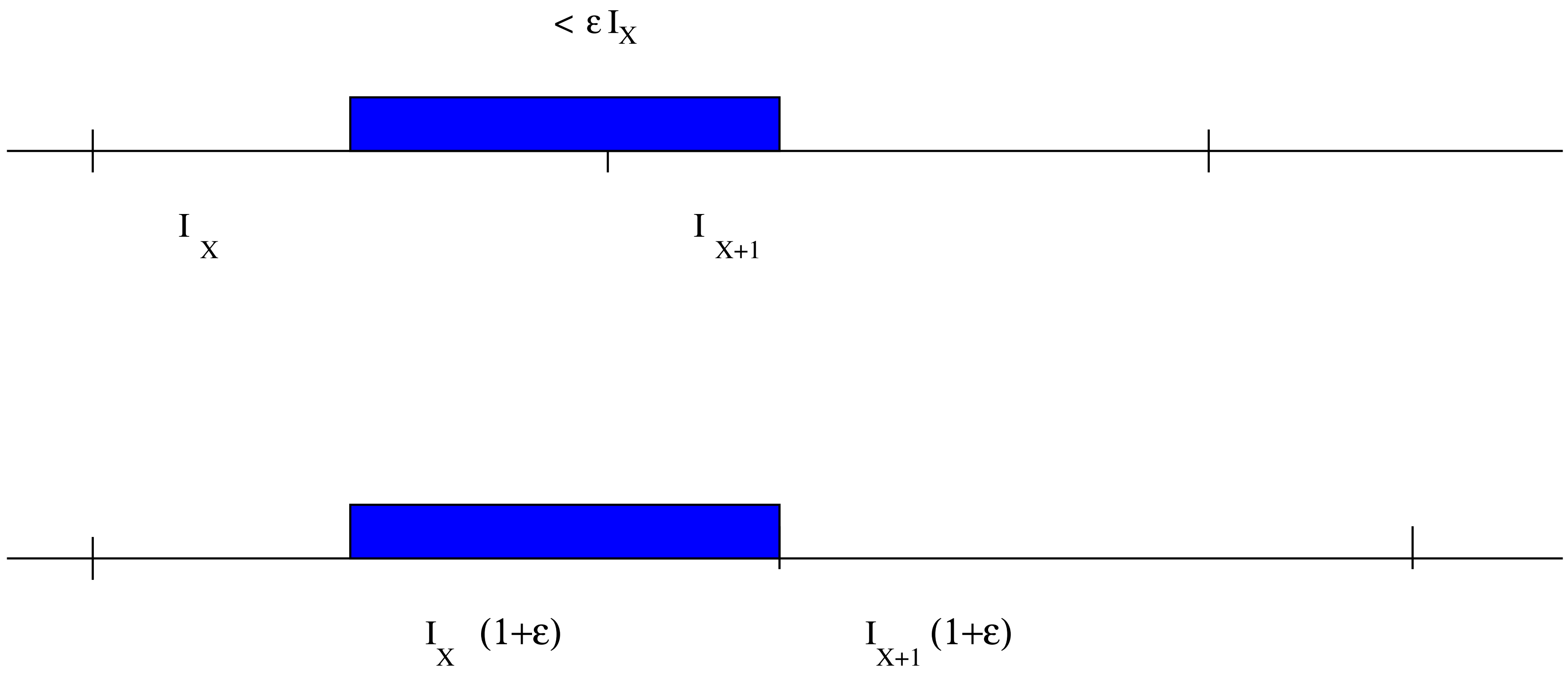
Goal: We want jobs to be small relative to the interval in which they run.

What if all jobs were really small

Small job definition. **A job j is small in interval x if $p_j \leq \epsilon |I_x|$.**

Lemma **If in the optimal schedule, all jobs are small when they run, then SPT is $(1+\epsilon)$ -optimal.**

Proof **Run SRPT, the optimal preemptive schedule. Schedule jobs where they first run, this can be accomplished by blowing up intervals by a $1+\epsilon$ -factor.**



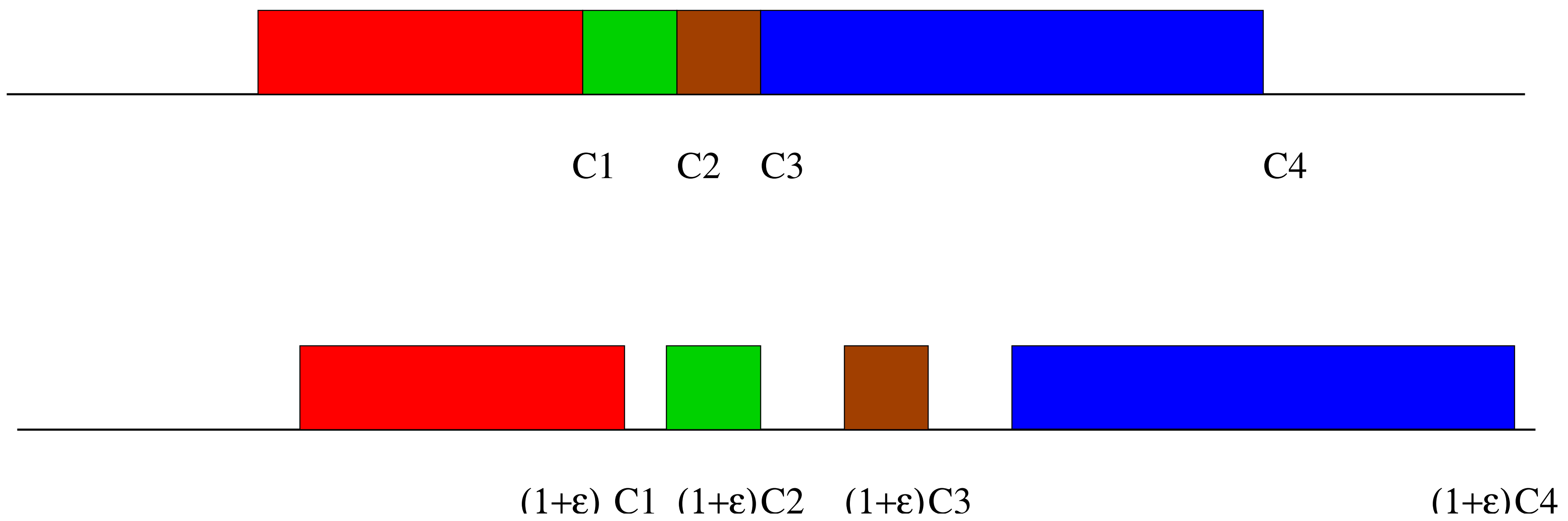
Transformation 2: Making jobs smaller

Lemma We can set r_j to be $\max\{r_j, \epsilon p_j\}$.

Proof Multiply all C_j by $1+\epsilon$. (This just blows up time by a $1+\epsilon$ factor.) Now, job j starts no earlier than

$$C_j(1+\epsilon) - p_j \geq p_j(1+\epsilon) - p_j \geq \epsilon p_j.$$

So we can increase its release date. $\square$



Note This implies that if job j runs in interval x , then

$$p_j \leq \frac{1}{\epsilon} R_x \leq \frac{1}{\epsilon^2} |I_x|.$$

Smaller jobs don't cross many intervals

Lemma Jobs can't cross more than $s = \log_{1+\epsilon}\left(1+\frac{1}{\epsilon}\right)$ intervals.

Proof If job j runs in interval x, then

$$p_j \leq \frac{1}{\epsilon^2}|I_x|.$$

Since interval sizes are geometrically increasing, after going out s intervals, there is enough total size to hold the entire job, i.e.

$$\sum_{k=0}^{s} |I_{x+k}| \geq \frac{1}{\epsilon^2}|I_x|.$$

Status

We have

$$p_j \leq \epsilon^{-2}|I_x|.$$

Can we get

$$p_j \leq \epsilon|I_x|?$$

We can make (most) jobs really small

We define a threshhold $t = \epsilon^7 OPT$.
No more than ϵ^{-7} jobs have $C_j > t$.

Lemma There is a $1 + O(\epsilon)$ -optimal schedule in which for each job j, either

- j is small when it runs, or
- j runs after t.

Proof idea: Let $k = \log_{1+\epsilon}\left(\frac{1}{\epsilon^4}\right)$. We move each large job forward k intervals. Since

$$p_j \leq \frac{1}{\epsilon^2}|I_x|,$$

and

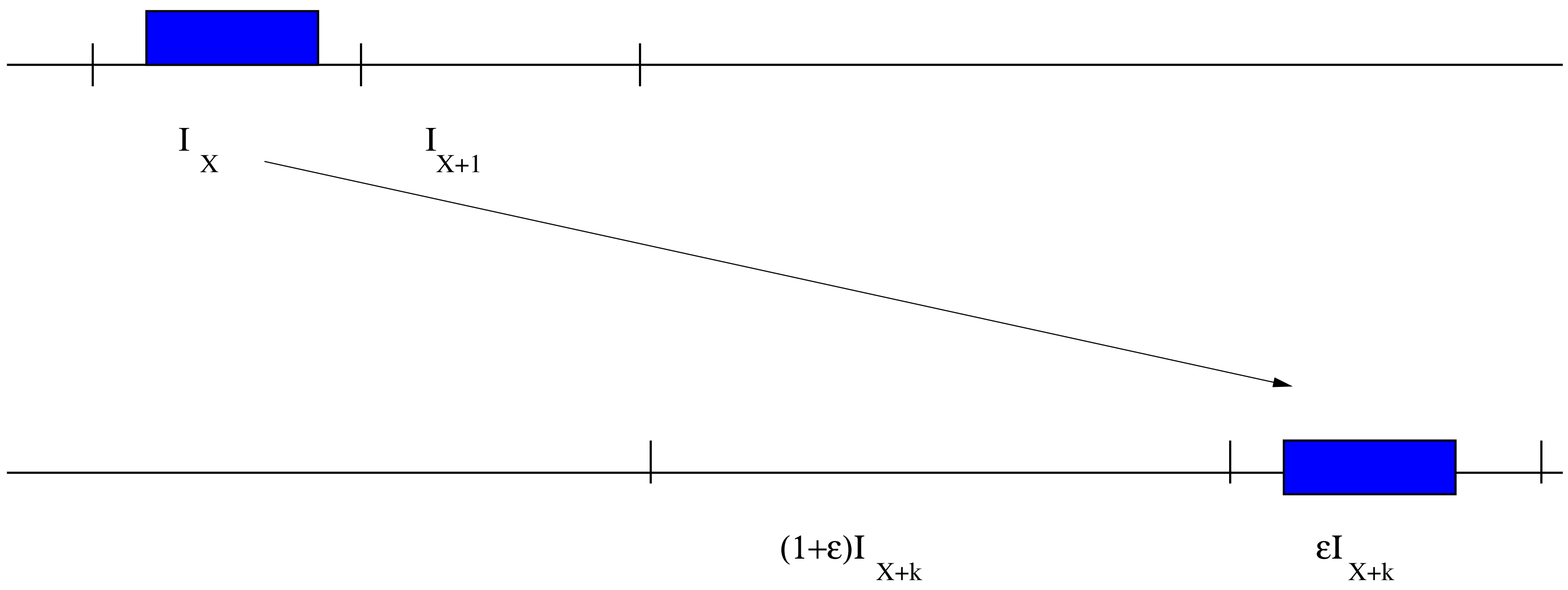
$$|I_{x+k}| \geq \frac{1}{\epsilon^4}|I_x|,$$

then

$$p_j \leq \epsilon^2 |I_{x+k}|.$$

the jobs are now small in the interval they run, and $\frac{1}{\epsilon}$ "fit" in the new interval expanded by $(1 + \epsilon)$.

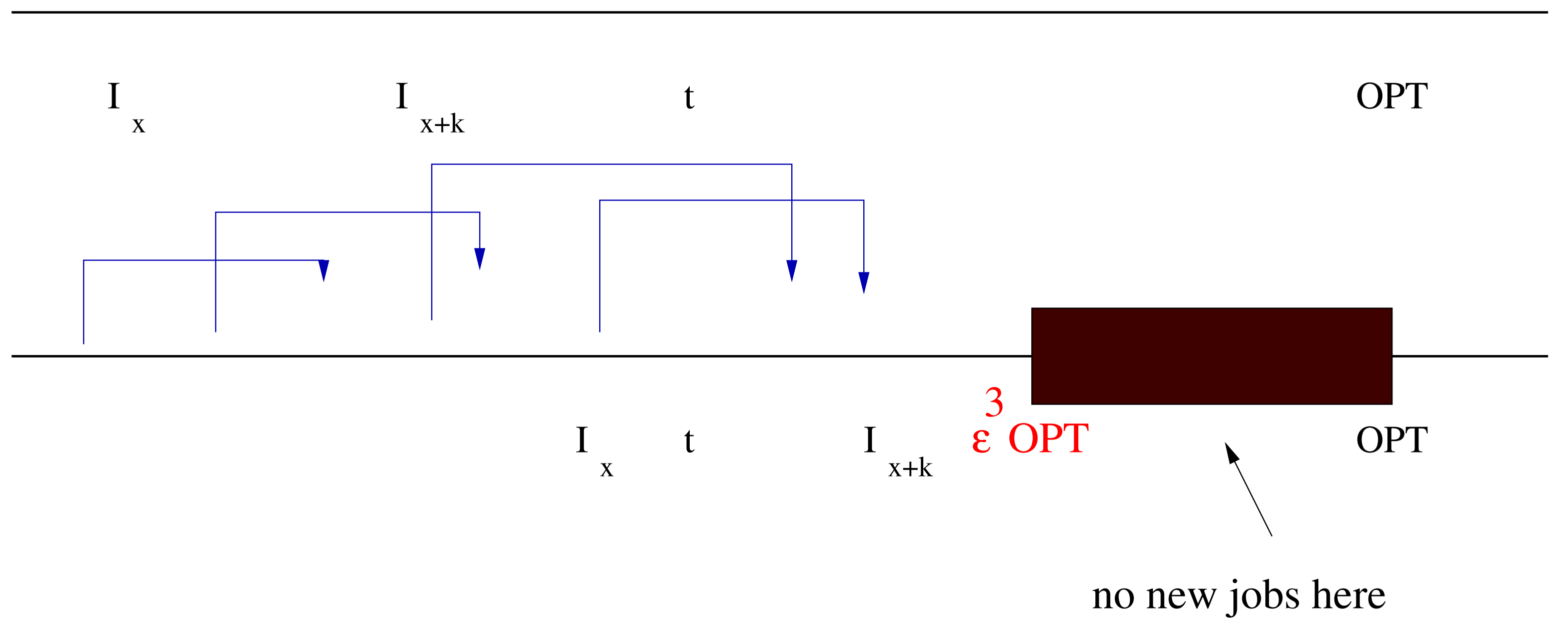
Moving forward large jobs



Making jobs small (cont)

Problem: We have blown up processing times of these jobs too much ($\frac{1}{\epsilon^4}$ factor).

Solution: True, but the total increase in processing time is small relative to OPT.



- Largest new completion time is about $\epsilon^{-4}t = \epsilon^3 OPT.$
- There can be at most ϵ^{-1} jobs that move forward into an interval. Thus they contribute a total of $\epsilon^2 OPT.$

Making jobs small (cont)

- **Completion times of jobs moved forward form a geometric series:**

$$\begin{aligned}\sum_{j \text{ moved forward}} \frac{C_j}{\epsilon^4} &= \epsilon^2 OPT + \frac{\epsilon^2 OPT}{1+\epsilon} + \frac{\epsilon^2 OPT}{(1+\epsilon)^2} \\ &= \epsilon^2 OPT \left(\frac{1+\epsilon}{\epsilon}\right) \\ &= O(\epsilon OPT)\end{aligned}$$

- **Additional complication:** I_x **may be full, so job may have to move** s **intervals later. This only loses a constant factor, which can be added to solution.**

Simple Algorithm

1. Guess OPT, $t = \epsilon^7 OPT$.
2. For each job, set $r_j = \max\{r_j, \epsilon p_j\}$, rounded up to a power of $(1+\epsilon)$.
3. If j is large in the interval containing r_j, set r_j to the minimum R_x so that p_j is small in interval I_x.
4. Round processing times up to a power of $(1+\epsilon)$.
5. Guess which $\frac{1}{\epsilon^7}$ jobs will complete after time t. (Call them B).
6. Run SPT on J-B.
7. Enumerate schedules for the jobs in B.

Running time is $O(\epsilon^{-7}!(n \log n) n^{\epsilon^{-7}})$

Better Algorithm

1. **For each job, set $r_j = \max\{r_j, \epsilon p_j\}$, rounded up to a power of $(1+\epsilon)$.**
2. **If j is large in the interval containing r_j, set r_j to the minimum R_x so that p_j is small in interval I_x.**
3. **Round processing times up to a power of $(1+\epsilon)$.**
4. **Run SPT on the modified instance, until there are $\frac{3}{\epsilon^7}$ jobs remaining.**
5. **Enumerate schedules for these remaining jobs, output best one.**

Running time is $O(\epsilon^{-7}! + (n \log n))$

Note: **Can decrease dependence on ϵ to about $2^{1/\epsilon^3}$, with more careful enumeration.**

Comments

1. Simple algorithm.
2. Nice dependence on ϵ. Any chance it is practical?
3. Is any PTAS practical?

Some hope: Some results by [Hepner Stein, 2002] suggest that a modification of the algorithm is competetive for certain sized-inputs and moderate values of ϵ (say 10%).

Back to original sum of flow time objective

- If we focus solely on worst-case analysis, we should give up and go home

What do we do if we want to solve this problem?

- If we focus solely on worst-case analysis, we should give up and go home

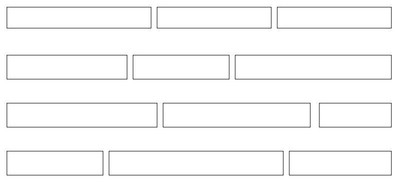
We have to look beyond traditional worst case analysis

- Is may be needlessly pessimistic.
- It has failed to differentiate between algorithms whose performance is observed empirically to be rather different.
- We really want to solve these problems.

NP-completeness proof

Reduction from 3-partition (Given $3n$ numbers $x_1, \ldots, x_{3n}$, can they be partitioned into n sets, each summing to $B = \sum x_i / n$

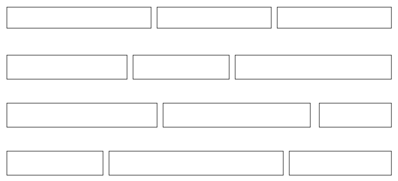
Sample Yes instance:



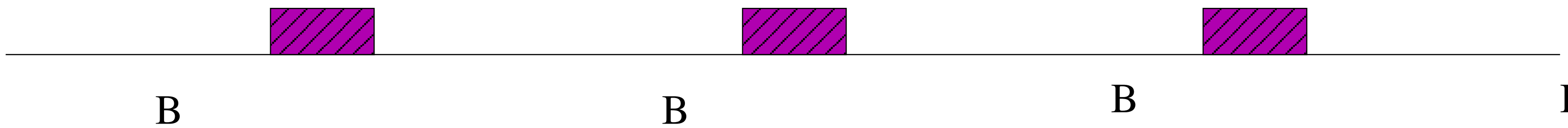
NP-completeness proof

Reduction from 3-partition (Given $3n$ numbers $x_1, \ldots, x_{3n}$, can they be partitioned into n sets, each summing to $B = \sum x_i / n$

Sample Yes instance:



Create a scheduling instance with gaps of size B .



- Partition is yes iff the jobs can be fit in the intervals
- Total flow time is small iff the jobs can fit in the intervals
- If total flow time is not small, it is really big (by $\sqrt{n}$ factor),

Observations

Observation : NP-completeness is not “robust.” A slight perturbation of the data destroys the proof (the entire $\sqrt{n}$ factor) disappears. (We’ll return to this)

Observation: (a slight digression)

- If we change our objective to $\sum C_j = \sum F_j + \sum r_j$, optimal solutions don’t change and the problem has a PTAS [Afrati et. al., 1999], and a simple $e/(e-1)$ -approximation [Chekuri, Motwani, Natarajan, Stein, 1997]
- Maybe if your algorithm doesn’t exploit the fact that $\sum C_j$ is a silly metric, it’s ok ...

End of digression

Analysis Technique

We want a worst-case analysis that identifies the case when hard instances are destroyed by small perturbations and where hard instances are "contrived" or "unnatural"

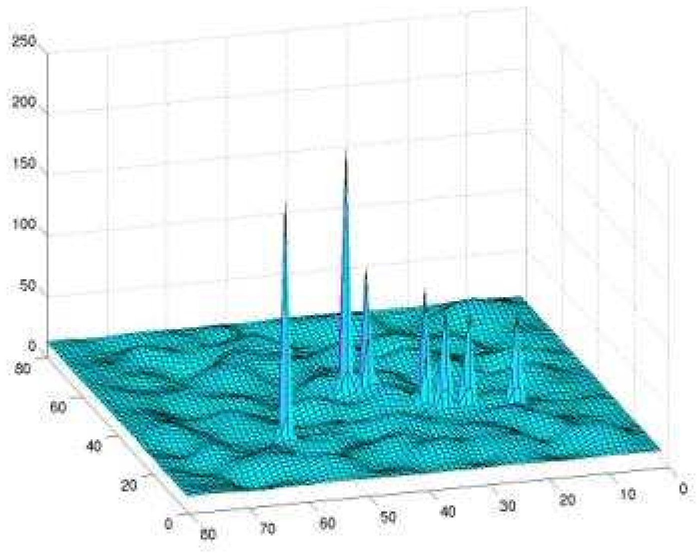
Analysis Technique

We want a worst case analysis that identifies the case when hard instances are destroyed by small perturbations, and where hard instances are "contrived" or "unnatural"

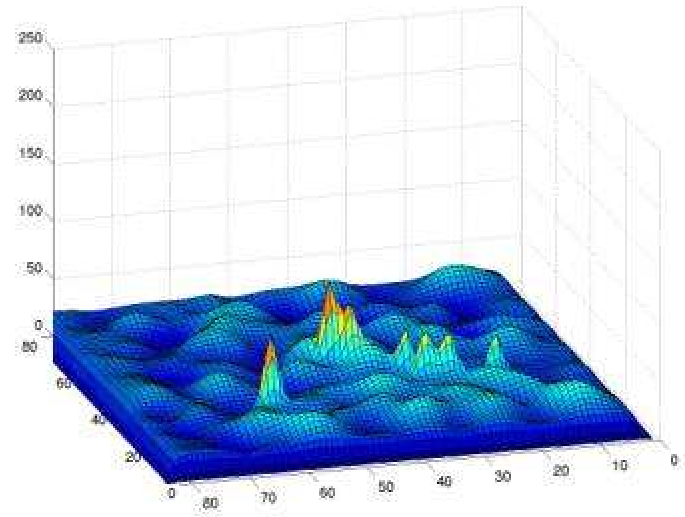
One example: Smoothed analysis [Spielman, Teng, 2001]

- Worst Case Complexity: $\max_x T(x)$
- Average case complexity: $\textbf{average}_r T(r)$
- Smoothed complexity : $\max_x \textbf{average}_r T(x + \epsilon r)$

Pictures from Dan Spielman's smoothed analysis web page:



250
200
150
100
50
0
80
60
40
20
0
80
70
60
50
40
30
20
10
0



250
200
150
100
50
0
80
60
40
20
0
80
70
60
50
40
30
20
10
0

Resource Augmentation

- Compare my algorithm using extra resources (faster/more machines) to an optimal algorithm that does not get the extra resources
- A s-speed ρ-approximation algorithm finds a schedule with objective at most ρ times OPT using a machine that is s times faster.
- Introduced explicitly in [Kalyanasundaram, Pruhs 2000] for on-line preemptive scheduling problems
- Applied to non-preemptive problems (and named) in [Phillips, Stein, Torng, Wein 1997]
- Used frequently in last 10 years, especially in on-line and scheduling problems.

Other related ideas: pseudoapproximation, etc.

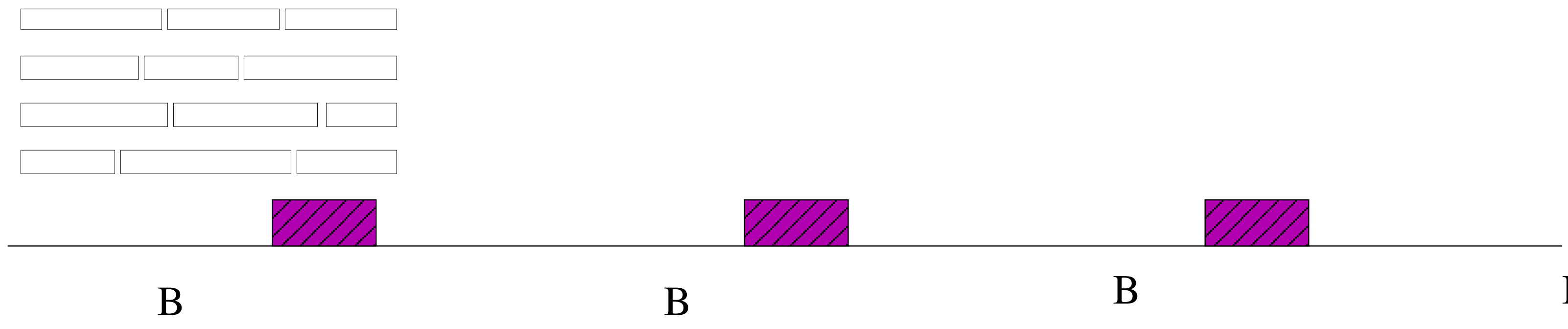
Thesis

Resource augmentation, using a small amount of additional resources is a natural way to analyze scheduling problems

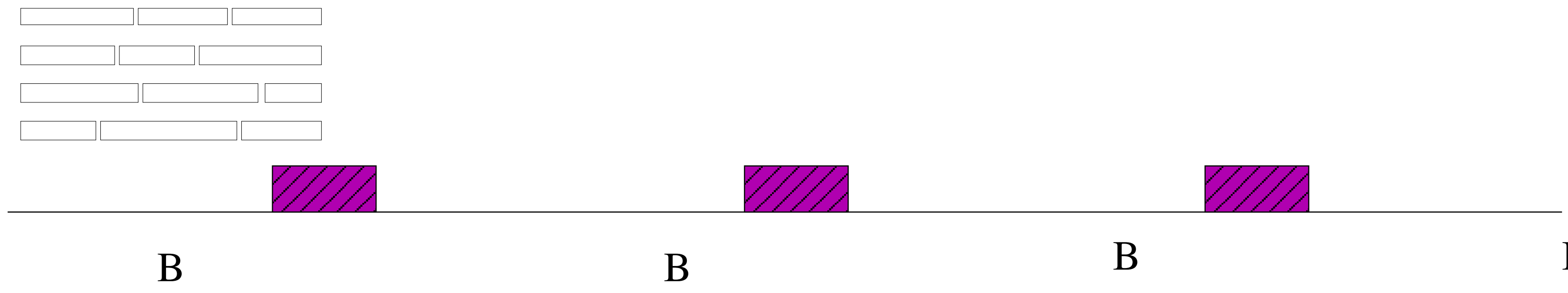
- Identifies when worst cases are specially tuned to the speed of the machine (maybe exact values of speed are artificial)
- It is often reasonable to buy/allocate more resources.
- Machines are getting faster at rates of a constant factor per year.
- A 2-speed algorithm is one that does as well as the optimal off-line did 18 months ago

How might speed help

Consider the hard instance:



Now speed up the machine slightly (20%)



All of a sudden, the problem becomes easy.

Summary: For this instance, if you speed the machine up by a few percent, a polynomial time algorithm can improve the flow time by $O(n^{1/2})$ factor.

Comclusion: This observation and other similar ones give hope.

State of Results (one year ago):

- Previously no $O(1)$ -speed approximation algorithms were known for minimizing flow time non-preemtively on one machine.
- Only logarithmic speed algorithms. [Phillips, Stein, Torng, Wein 1997]
- The logarithimic speed algorithms
  - are unnatural,
  - have too large a speed needed.

Our Results

Input: **1 machines, jobs have**

- **release date** r_j
- **processing time** p_j
- **(possible) deadline** d_j
- **weight** w_j

Schedule **is non-preemptive, assigns completion times** C_j **. Other functions are:**

- **Flow time** $F_j = C_j - r_j$
- **Tardiness** $T_j = \max\{C_j - d_j, 0\}$
- **Throughput** $\bar{U}_j = [C_j \leq d_j]$

Our Results

Input: **1 machines, jobs have**

- **release date** r_j
- **processing time** p_j
- **(possible) deadline** d_j
- **weight** w_j

New Results

[Bansal, Chen, Kandehar, Pruhs, Schieber, Stein 2007]

Schedule is non-preemptive, assigns completion times C_j . Other functions are:

– Flow time $F_j = C_j - r_j$

– Tardiness $T_j = \max\{C_j - d_j, 0\}$

– Throughput $\bar{U}_j = [C_j \leq c_j]$

Results We give the first $O(1)$ -speed $O(1)$ -approximation algorithms for

– Weighted Flow Time ($\sum w_j F_j$)

– Total Tardiness ($\sum T_j$)

– Broadcast Scheduling Version of Weighted Flow Time ($\sum w_j F_j$)

– Throughput Maximization ($\sum \bar{U}_j$) (exact value of objective)

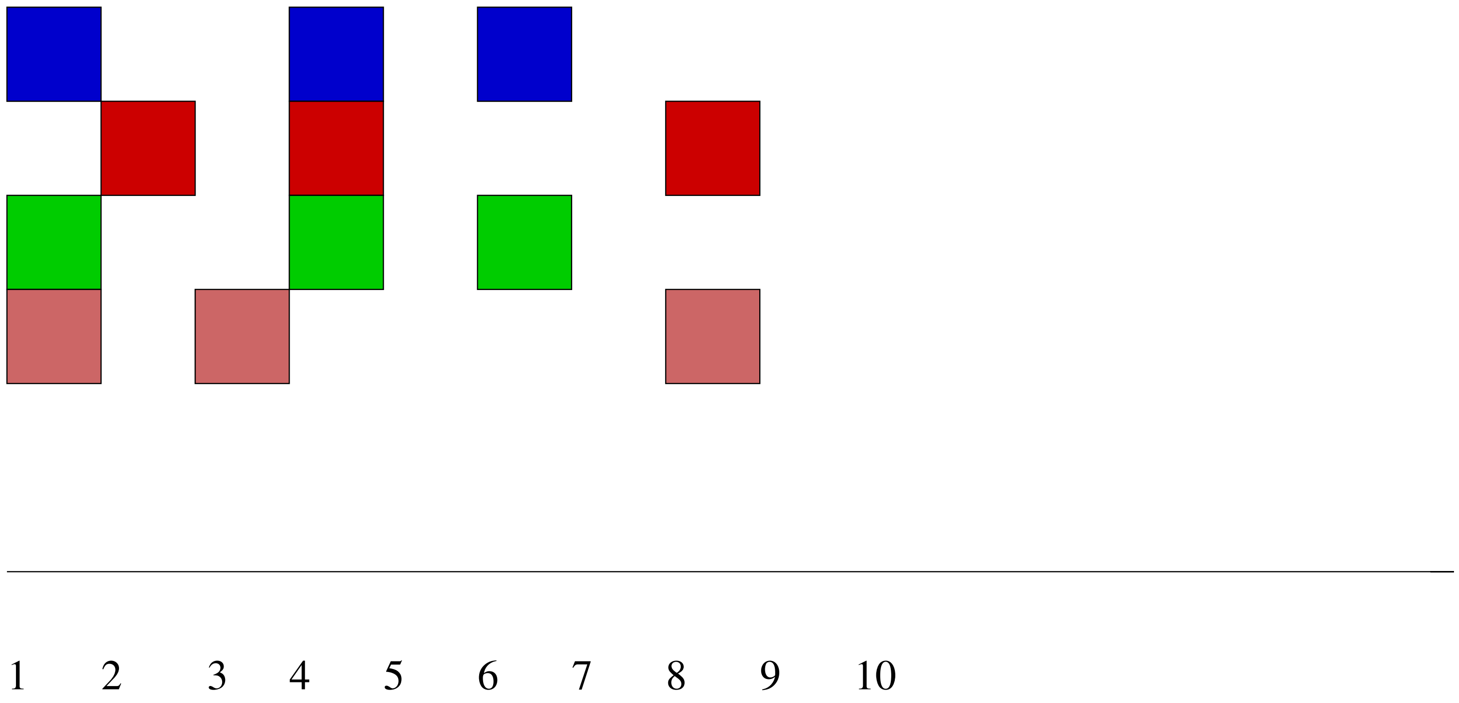
– Weighted Tardiness ($\sum w_j T_j$) (using extra machines also)

Bonus Feature: Unified approach to different metrics

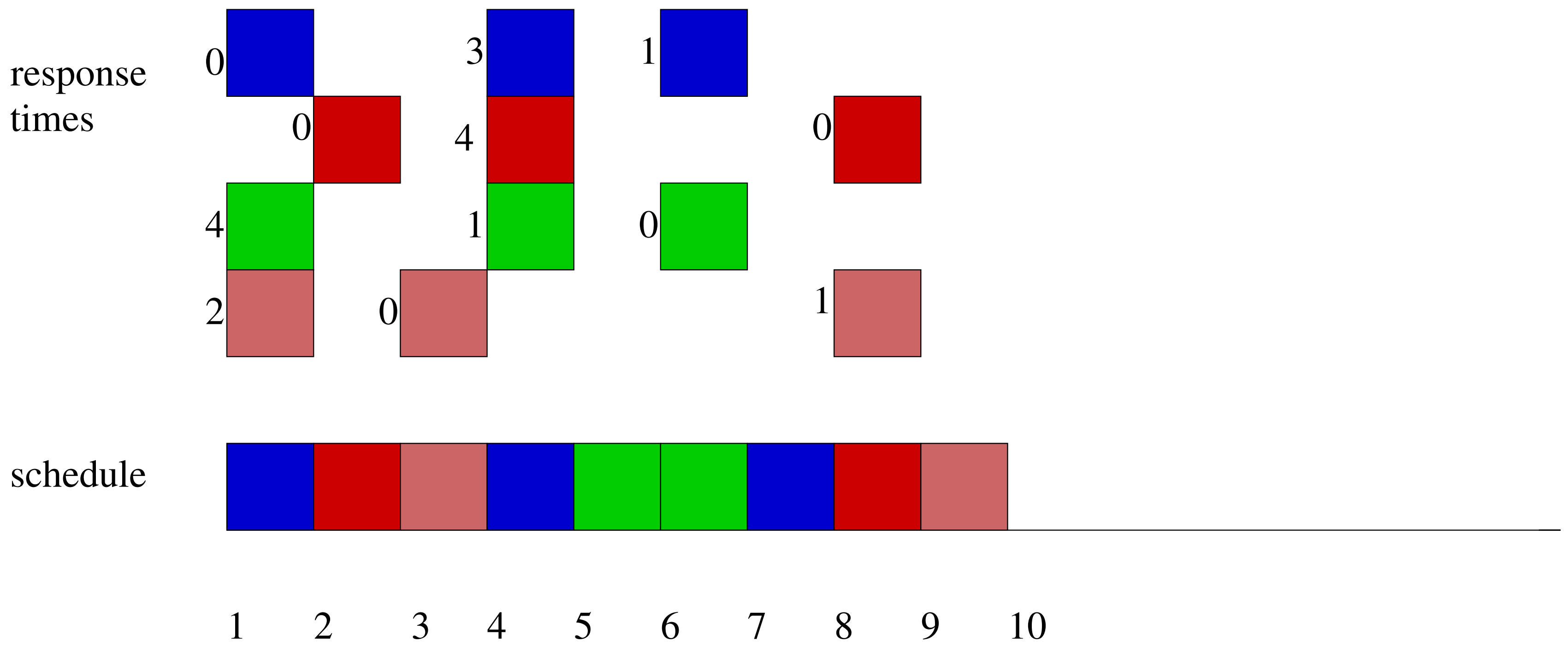
Broadcast Scheduling

- Requests arrive at a time r_j for a particular x_j and may have a weight w_j .
- At each integer time step, one item x' is broadcast, and all requests for which $x_j = x'$ are satisfied.

Input:



Solution



Technical Details

General approach:

- **Formulate an IP**
- **Solve the LP-relaxation**
- **Round the LP-relaxation**

Technical Details

General approach:

- Formulate an IP We augment a time-indexed formulation with a new set of constraints
- Solve the LP-relaxation We use some of the problem structure to approximately solve the exponential-sized LP in polynomial time
- Round the LP-relaxation We use a careful rounding procedure that uses the extra speed crucially

Because the rounding procedure doesn't move jobs too much, we can handle other objectives (tardiness, throughput) with small modifications.

Time Indexed LP

Variables x_{jt} **denote whether job** j **starts at time t.**

Consider the well-known exact IP formulation

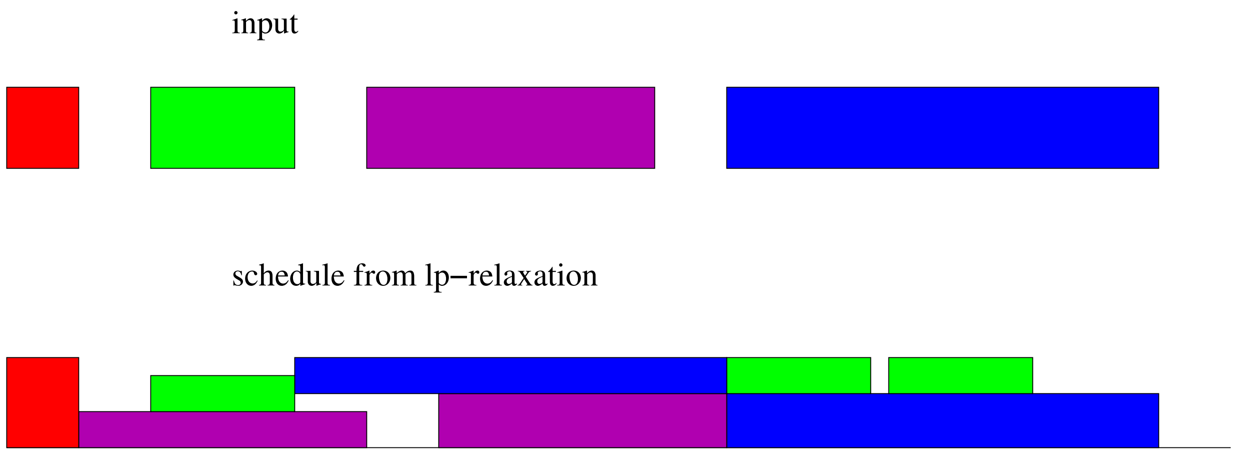
$$\begin{array}{rrcll}
\textbf{min} & & & \sum_{j\in J} w_j F_j & \\
\textbf{s.t.} & & & & \\
& \sum_t x_{jt} & = & 1 & \forall j \in J \\
& \sum_{j\in J}\sum_{\tau:\tau\in(t-p_j,t]} x_{j\tau} & \leq & 1 & \forall t \in \mathbb{Z} \\
& F_j & = & \sum_t (t+p_j-r_j)x_{jt} & \forall j \in J \\
& x_{jt}, F_j & \geq & 0 & \forall j \in J, t \in \mathbb{Z} \\
& x_{jt}, F_j & & \textbf{integer} & \forall j \in J, t \in \mathbb{Z}
\end{array}$$

Time Indexed LP

Variables x_{jt} **denote whether job** j **starts at time t.**
Consider the LP relaxation

$$
\begin{array}{lrcll}
\textbf{min} & \sum_{j \in J} w_j F_j & & & \\
\textbf{s.t.} & & & & \\
& \sum_t x_{jt} & = & 1 & \forall j \in J \\
& \sum_{j \in J} \sum_{\tau : \tau \in (t - p_j, t]} x_{j\tau} & \leq & 1 & \forall t \in \mathbb{Z} \\
& F_j & = & \sum_t (t + p_j - r_j) x_{jt} & \forall j \in J \\
& x_{jt}, F_j & \geq & 0 & \forall j \in J, t \in \mathbb{Z} \\
& x_{jt}, F_j & & \textbf{integer} & \forall j \in J, t \in \mathbb{Z}
\end{array}
$$

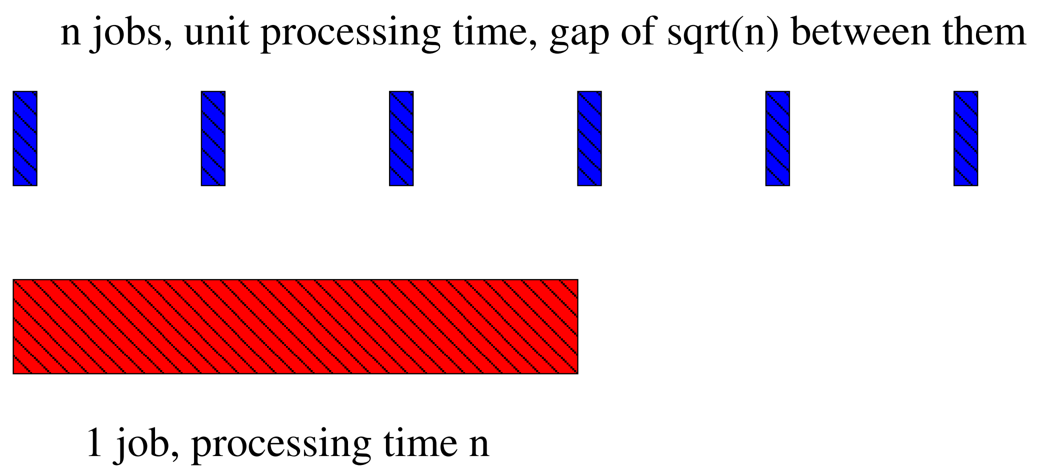
Fractional Version: **Schedules a job multiple times using a fraction of the machine**



Bad News about the LP relaxation

- **The LP, on a constant speed processor, has a super-constant integrality gap**
- **For the LP schedule, there are inputs such that the optimal non-preemptive flow time, given 1 speed-s machine with $s = o(n^{1/4})$ is polynomially larger than the optimal LP flow time given one unit speed machine. [PSTW]**

A suggestive example

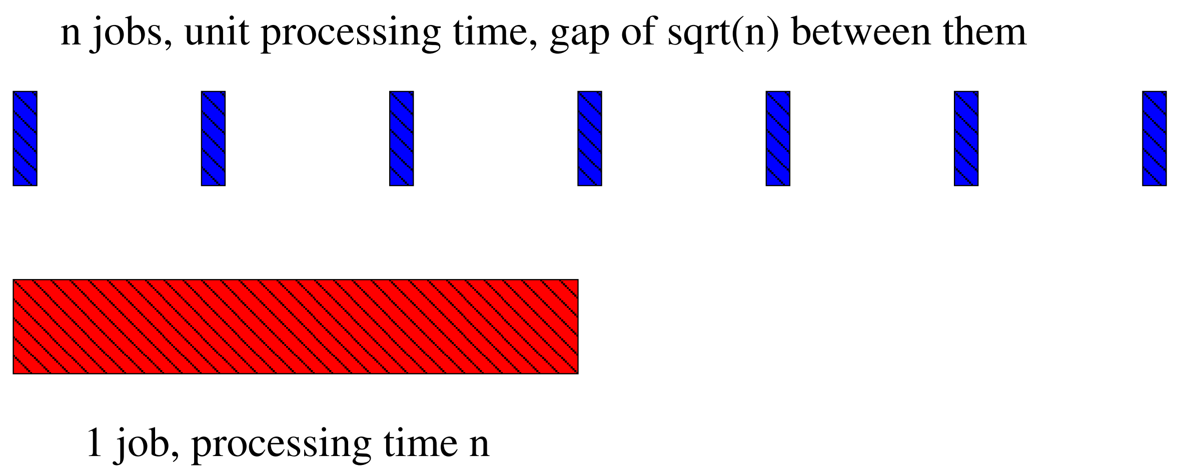


Optimal non-preemptive (IP) schedule **Two possibilities:**

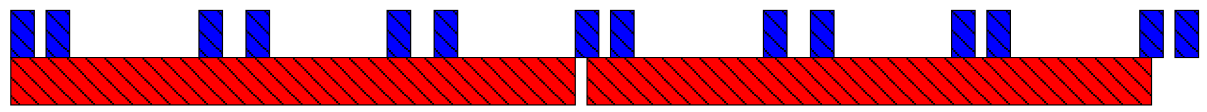
– **Run big job in last half of schedule, it has flow time** $\Theta(n^{3/2})$

– **Run big job in first half of schedule, it delays** $\sqrt{n}$ **jobs by** $\Theta(n)$ **each, for a total flow time of** $\Theta(n^{3/2})$ **.**

A suggestive example



Optimal LP schedule

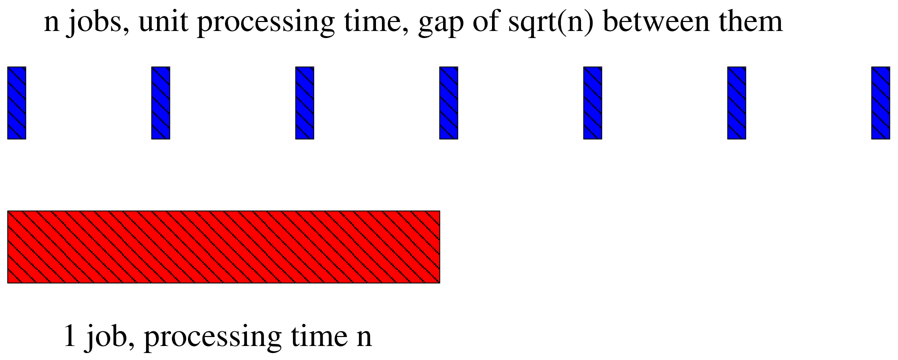


– **Small jobs have constant flow time**

– **Big job has flow time of** $O(n)$.

– **Total flow time is** $O(n)$.

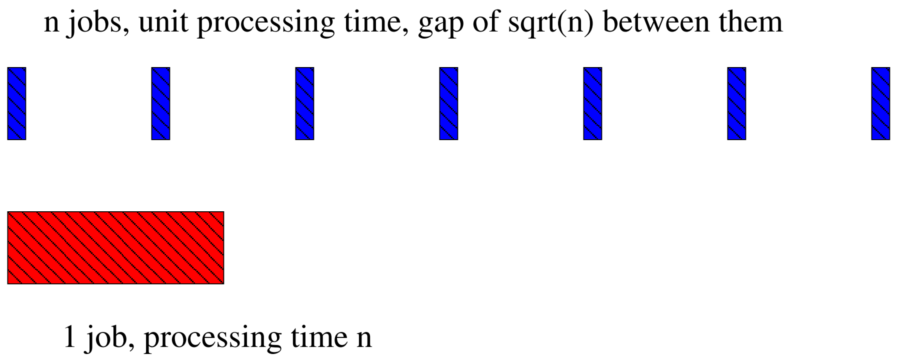
Extra speed doesn't help here



Optimal non-preemptive (IP) schedule **Two possibilities:**

– Run big job in last half of schedule, it has flow time $\Theta(n^{3/2})$

– Run big job in first half of schedule, it delays $\sqrt{n}$ jobs by $\Theta(n)$ each, for a total flow time of $\Theta(n^{3/2})$.



Same argument with extra speed (up to constants).

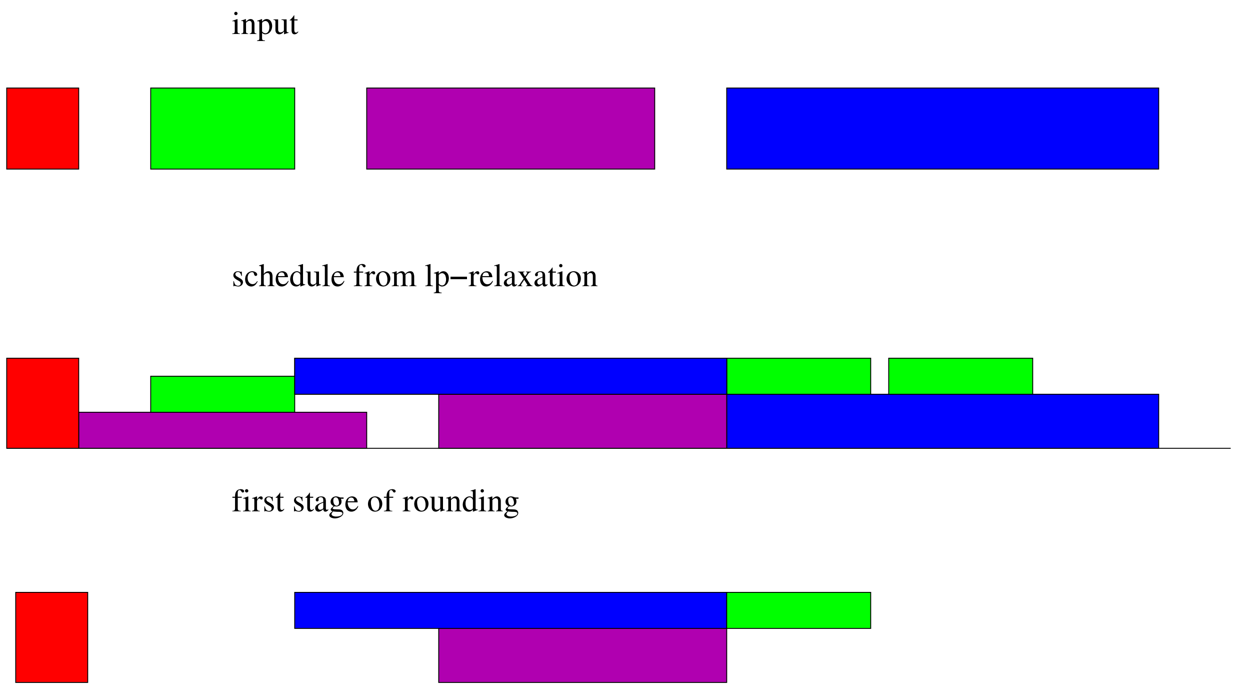
Gap is $O(n^{1/2})$. Need a stronger LP.

New Idea

- We will add constraints to the IP so that it is no longer exact (will be off by a factor of 2)
- But, this stronger IP will be easier to round and will not have the gap

What happens in any (including our) rounding algorith

– You solve the LP and each job is placed in multiple places, in multiple pieces.



- The LP "tells" you to run two jobs at the same time
- You need to run one after the other, and need to "charge" the additional time incurred by one to something.

What happens in any rounding algorithm

- **You solve the LP and each job is placed in multiple places, in multiple pieces.**
- **The LP “tells” you to run two jobs at the same time**
- **You need to run one after the other, and need to “charge” the additional time incurred by one to something.**

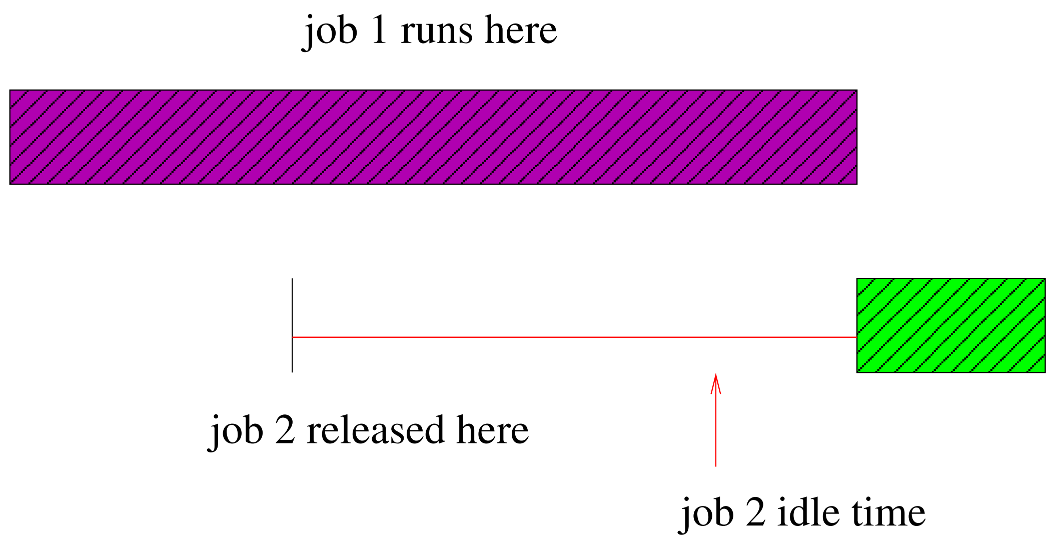
Recall: **Let C_j be completion time S_j be start time.**

$$\begin{aligned} F_j &= C_j - r_j \\ &= (S_j + p_j) - r_j \\ &= (S_j - r_j) + p_j \end{aligned}$$

- **So we can charge either against processing time, or elapsed time.**
- **What if both are small?**

New Constraints

Idea: **If job k starts at time t, and runs in the interval $I = [t, t + p_k]$, then any job released during interval I must start after interval I.**



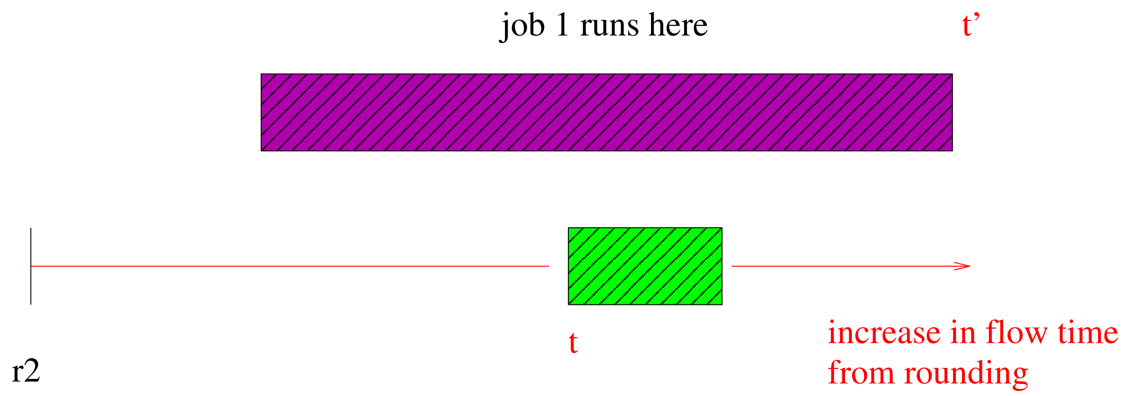
The New LP

$$\begin{array}{rrcll}
\textbf{min} & \sum_{j\in J} w_j F_j & & & \\
\textbf{s.t.} & & & & \\
& \sum_t x_{jt} & = & 1 & \forall j \in J \\
& \sum_{j\in J}\sum_{\tau:\tau\in(t-p_j,t]} x_{j\tau} & \leq & 1 & \forall t \in \mathbb{Z} \\
& F_j & = & \frac{1}{2}\left(\sum_t (t+p_j-r_j)x_{jt}\right. & \\
& & & \left. +p_j + \sum_{k:C_k^{-1}>C_j^{-1}} \quad \sum_{t\in[r_j-p_k+1,r_j]} (t+p_k-r_j)x_{kt}\right) & \forall j \in J \\
& x_{jt}, F_j & \geq & 0 & \forall j \in J, t \in \mathbb{Z}
\end{array}$$

The flow time of a job may be counted twice, so this is not an exact IP.

Intuition for why new constraint helps

- Consider a big job J_1 and a small job J_2 that the LP wants to run at the same time.
- Let t be the time that the LP schedules J_2 , t' the time that the LP completes J_1
- Suppose we want to move J_2 to run after J_1 . (the additional speed will help it fit in)
  - Already accumulated flow time in LP = $(t - r_j) + p_j$
  - Additional flow time from rounding = $t' - (t + p_j)$

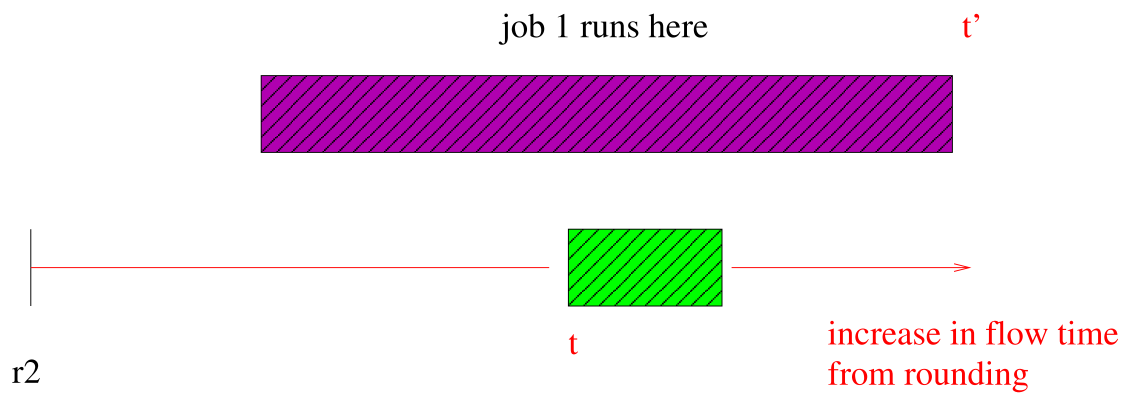


We need to charge the additional flow time of J_2 to something. In the traditional time indexed LP, we can charge against:

- Processing time of J_2 .
- Contribution to LP objective from J_2

Intuition for why new constraint helps

- Already accumulated flow time in LP = $(t - r_j) + p_j$
- Additional flow time from rounding = $t' - (t + p_j)$



We need to charge the additional flow time of J_2 to something. In the traditional time indexed LP, we can charge against:

- Processing time of J_2 .
- Contribution to LP objective from J_2

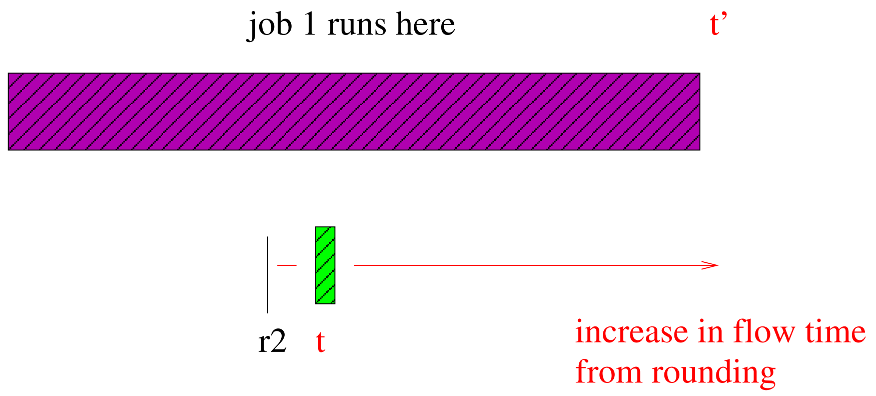
If either of these are large relative to $t' - (t + p_j)$, we are fine.

Note: Another case is when J_2 runs before J_1.

Intuition

- **If additional flow time $t' - (t + p_j)$ is small relative to either $(t - r_j)$ or p_j then the rounding only increases J_2's flow time by a constant**

Problem case



- **Problem with original time-indexed LP when a job runs near its release date and p_j is small.**
- **New LP includes a term whose magnitude is exactly $t' - (t + p_j)$, and so we can charge increase against this.**

Algorithm

1. The original instance J is modified to create a new instance $\hat{J}$. In $\hat{J}$ the job sizes are rounded down so that the possible job sizes form a geometric sequence, in multiples of a parameter $\beta > 1$
2. From $\hat{J}$, a linear program LP is created. An integer solution to LP can be interpreted as an aligned schedule. An *aligned* schedule is one in which each job with size p is started at a time that is an integer multiple of p. The optimal solution to LP will be a lower bound on OPT
3. The linear program LP is then solved. An arbitrary solution is then converted into a canonical solution that essentially favors jobs which are released earlier.
4. The solution of LP is randomly rounded into a pseudo-schedule. In a pseudo-schedule each job is run exactly once, but more than one job may be running at each time.
5. Using some additional speed, this pseudo-schedule is converted into a feasible schedule for $\hat{J}$.
6. Finally, again using some additional speed, a feasible schedule for J is produced.

Details: Running Time

Running Time: **The rounding and alligning allow the LP to be solved in polynomial time giving up a** $1 + o(1)$ **factor.**

Method: **Geometric rounding, plus a grouping of consecutive intervals in which nothing interesting happens gives a polynomial sized-LP.**

Details: Rounding to a pseudoschedule:

- Jobs are in classes based on similar processing times.
- For each class independently
  - Within a class, order jobs by larger weight first , breaking ties by earlier release date, breaking ties by index.
  - For each class, pick a random offset $\alpha \in [0, 1)$ at random.
  - Schedule a job whenever total LP processing time reaches $\alpha, \alpha + 1, \alpha_2, \ldots$

The pseudoschedule has the following properties:

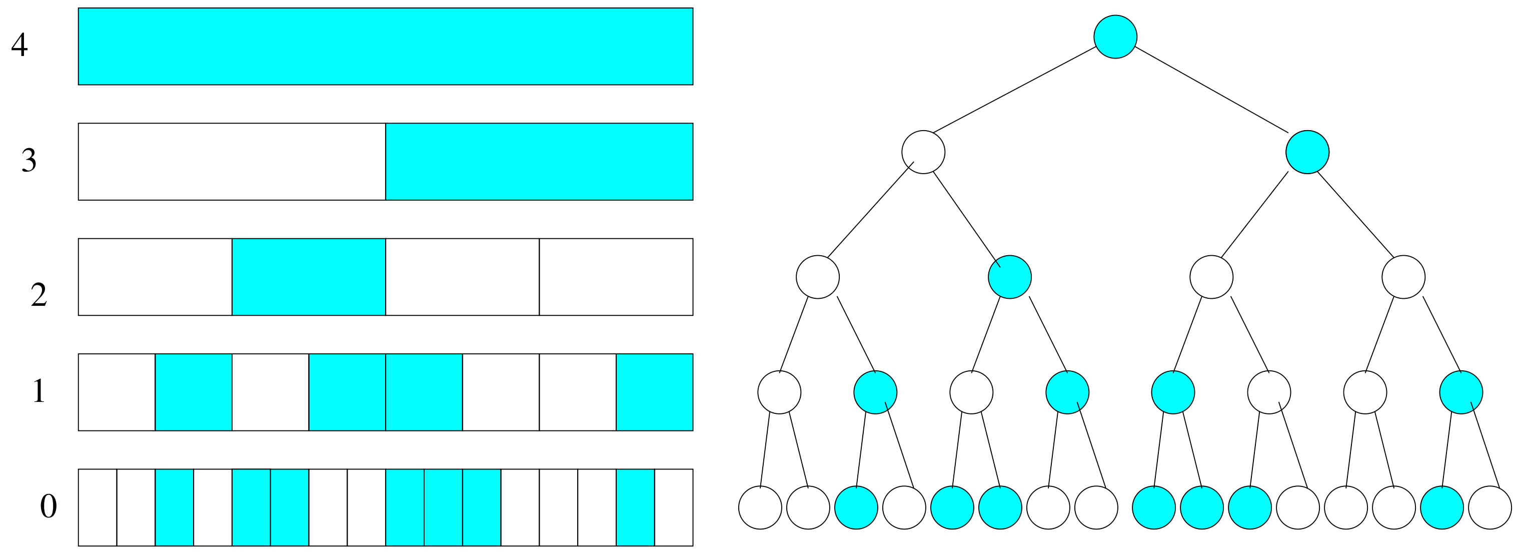
1. Each job $j \in J$ is scheduled exactly once
2. No two jobs from the same class $\mathcal{C}_i$ are scheduled in the same aligned β^i-interval.
3. Consider any aligned β^i-interval for $0 \leq i \leq \kappa$. The total size of all the jobs in classes $\mathcal{C}_0, \ldots, \mathcal{C}_i$ scheduled in this interval is at most $\beta^i + \frac{\beta^{i+1}-1}{\beta-1} < \beta^i(2 + \frac{1}{\beta-1})$.

Details: Converting pseudoschedule to Schedule

- **Rounding is done by shrinking jobs (extra speed) by a factor of $(2 + \frac{1}{\beta - 1})$ and then using the holes to pack jobs that are simultaneous in the pseudoschedule.**
- **The earlier intuition dealt with two job sizes, we have to deal with multiple job sizes.**

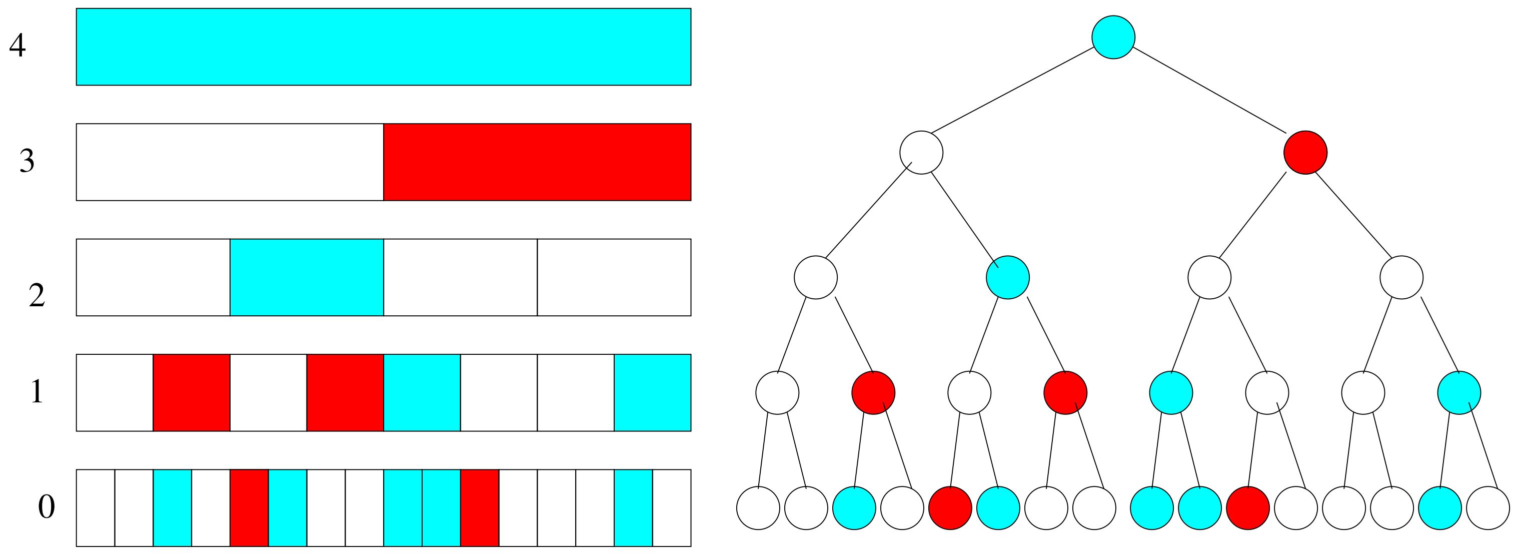
Details: Converting pseudoschedule to Schedule

First, look at the jobs in each class, alligned ($\beta = 2$ here)



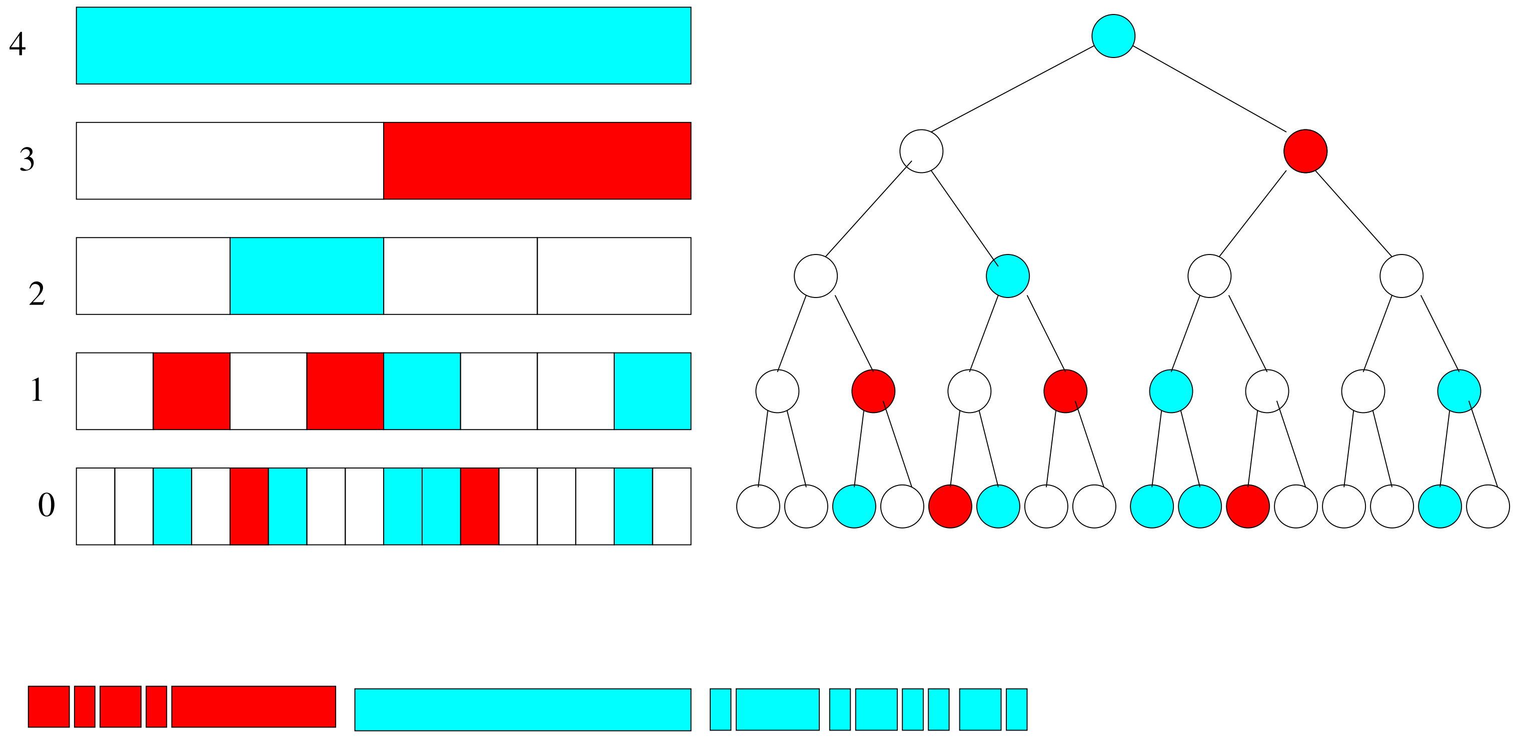
Details: Converting pseudoschedule to Schedule

Distinguish between early (red) jobs, those whose release date is before the big job, and other later (blue) jobs.



Details: Converting pseudoschedule to Schedule

- Schedule early jobs by postorder traversal
- Schedule big job
- Schedule late jobs by preorder traversal



Details: Converting pseudoschedule to Schedule

Claim **The schedule produced is feasible and has a flow time at most twice the expected flow time of the pseudoschedule.**

Details: making the algorithm polynomial time

Issue: **The time-indexed LP has an exponential number of variables and constraints.**

$$
\begin{array}{lrcll}
\textbf{min} & \sum_{j\in J} w_j F_j & & & \\
\textbf{s.t.} & & & & \\
& \sum_t x_{jt} & = & 1 & \forall j \in J \\
& \sum_{j\in J}\sum_{\tau:\tau\in(t-p_j,t]} x_{j\tau} & \leq & 1 & \forall t \in \mathbb{Z} \\
& F_j & = & \frac{1}{2}\left(\sum_t (t+p_j-r_j)x_{jt}\right. & \\
& & & \left. +p_j + \sum_{k:C_k^{-1}>C_j^{-1}} \quad \sum_{t\in[r_j-p_k+1,r_j]} (t+p_k-r_j)x_{kt}\right) & \forall j \in J \\
& x_{jt}, F_j & \geq & 0 & \forall j \in J, t \in \mathbb{Z}
\end{array}
$$

Ideas for making the LP polynomial sized

Unweighted flow time: We will reduce size of LP while increasing objective function by $1 + o(1)$ factor.

- Let $p_{\max} = \max_j p_j$.
- Each job j runs somewhere in the interval $L_j = [r_j, r_j + np_{\max}]$.
- Let $L = \cup_j L_j$ be the set of all times a job might possibly run in any optimal schedule.
- $|L| \leq n^2 p_{\max}$.
- Round all processing times to integer multiples of $p_{\text{small}} = p_{\max}/n^3$.
- In optimal schedule, each job's completion time is increased by at most np_{small} .
- total flow is increased by $n^2 p_{\text{small}} \leq p_{\max}/n \leq F_{\text{OPT}}/n$.
- Similarly, we can round up each release date to be a multiple of p_{small} , with the same increase in total flow.

Sketch of ideas for making the LP polynomial sized

Weighted flow time: We will reduce size of LP while increasing objective function by factor of 2.

- Recall that jobs are in classes based on processing time rounded to a power of 2.
- Recall that we can align intervals to be multiples of processing times rounded to a power of 2.
- We can't just round as before, because a small processing time job can have a very large weight.

Definition: An aligned β^i -interval I of the form $[\hat{r}_j + xp_j, \hat{r}_j + (x+1)p_j]$ for some job j contains an interesting time if one of the following holds:

- x is an integer power of 2, or
- I contains the release time of a job, or
- there is a variable of the form x_{kab} for a job $k \in C_l$ for $l < i$ where either a or b is properly contained in I .

Ideas:

- Intervals that are not interesting can be merged, and only a factor of 2 is lost in the objective. Call the associated varibles *smeared* . intervals.
- Call the remaining variables *regular* .
- Smeared intervals are either identical or disjoint.
- Reexpress LP in terms of these new variables.
- Can show inductively that there are only a polynomial number of such variables.

Recap

Results We give the first $O(1)$ -speed $O(1)$ -approximation algorithms for

- Weighted Flow Time ($\sum w_j F_j$)
- Total Tardiness ($\sum T_j$)
- Broadcast Scheduling Version of Weighted Flow Time ($\sum w_j F_j$)
- Throughput Maximization ($\sum \bar{U}_j$) (exact)
- Weighted Tardiness ($\sum w_j T_j$) (using extra machines also)

Additional Results

- Our new LP cannot achieve an $O(1)$ approximation using speed < 2
- Can achieve $O(\frac{-\log \epsilon}{\epsilon})$ -machine $1 + O(\epsilon)$ -speed $O(1)$ -approximation polynomial-time algorithm.

Open Questions

- Better constants (We have constants around 10 or so)
- Other minsum problems
- Multiple machines
- Other ways to deal with hard-to-approximate problems