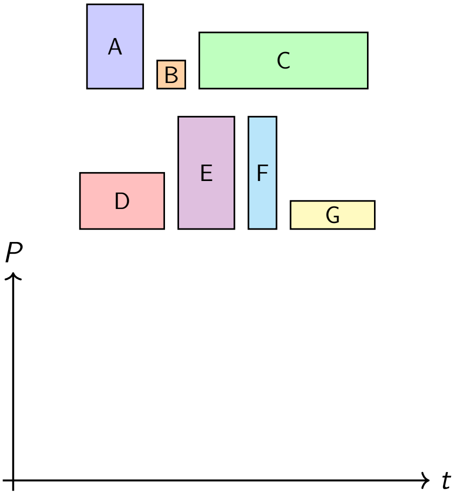
CatBatch: A New Algorithm for Online Scheduling of Rigid Task Graphs with Near-Optimal Competitive Ratio

Lucas Perotin, Hongyang Sun, Padma Raghavan
Vanderbilt University, Nashville, USA

July 10, 2025 – 18th Scheduling Workshop

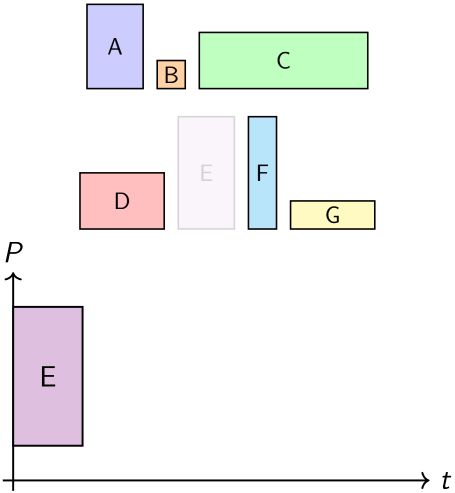
Outline

Scheduling Independent Rigid Tasks: A Simple Framework



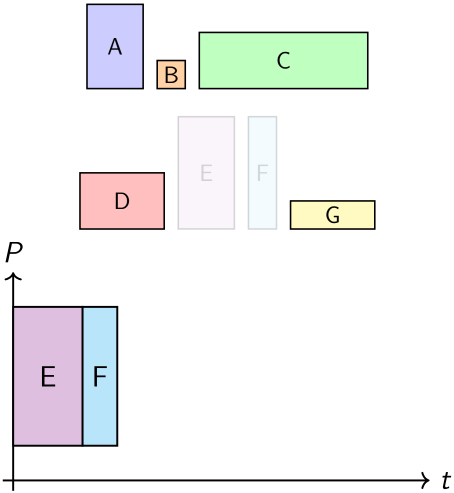
- n tasks to schedule on a platform with P processors
- Each task has length t and processor requirement p
- Tasks lengths are considered known prior execution
- Goal: minimize makespan T (total time of execution)

Scheduling Independent Rigid Tasks: A Simple Framework



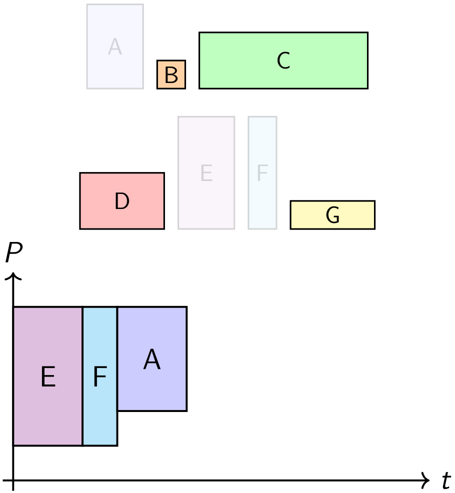
- n tasks to schedule on a platform with P processors
- Each task has length t and processor requirement p
- Tasks lengths are considered known prior execution
- Goal: minimize makespan T (total time of execution)
- Straightforward heuristic: arrange tasks by decreasing p and schedule them ASAP (no backfilling)

Scheduling Independent Rigid Tasks: A Simple Framework



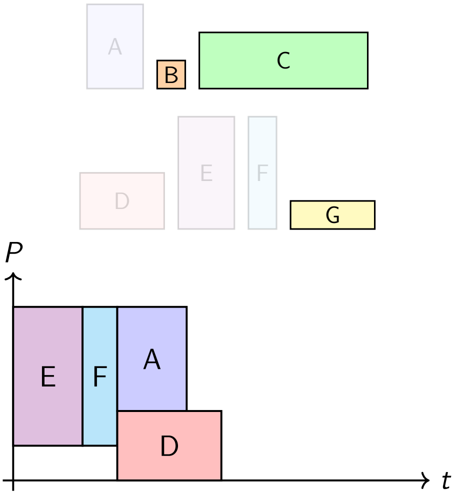
- n tasks to schedule on a platform with P processors
- Each task has length t and processor requirement p
- Tasks lengths are considered known prior execution
- Goal: minimize makespan T (total time of execution)
- Straightforward heuristic: arrange tasks by decreasing p and schedule them ASAP (no backfilling)

Scheduling Independent Rigid Tasks: A Simple Framework



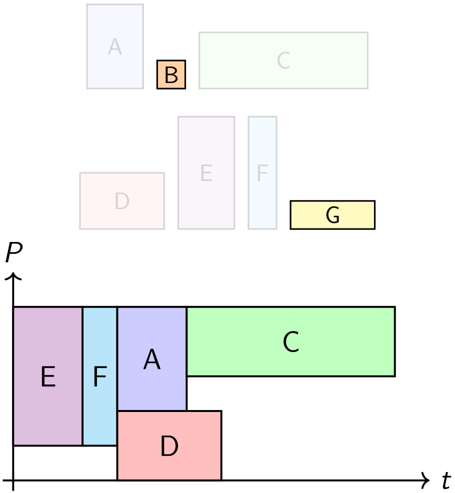
- n tasks to schedule on a platform with P processors
- Each task has length t and processor requirement p
- Tasks lengths are considered known prior execution
- Goal: minimize makespan T (total time of execution)
- Straightforward heuristic: arrange tasks by decreasing p and schedule them ASAP (no backfilling)

Scheduling Independent Rigid Tasks: A Simple Framework



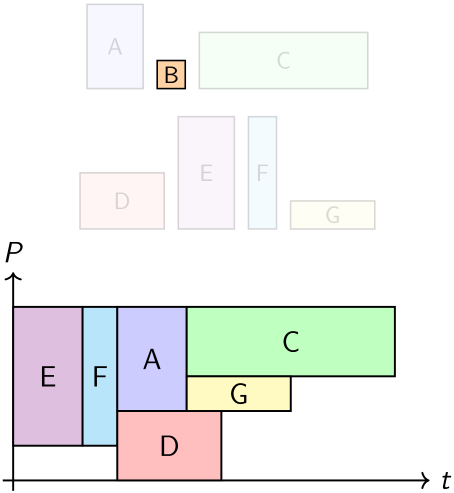
- n tasks to schedule on a platform with P processors
- Each task has length t and processor requirement p
- Tasks lengths are considered known prior execution
- Goal: minimize makespan T (total time of execution)
- Straightforward heuristic: arrange tasks by decreasing p and schedule them ASAP (no backfilling)

Scheduling Independent Rigid Tasks: A Simple Framework



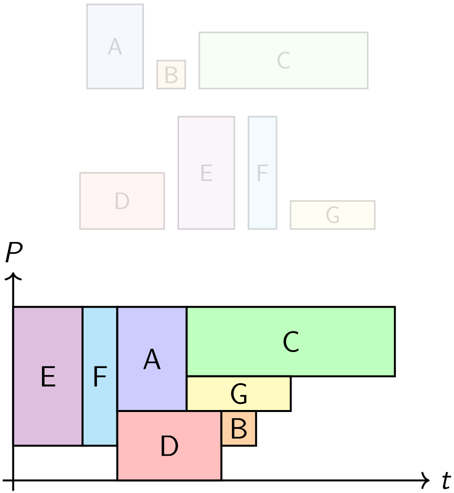
- n tasks to schedule on a platform with P processors
- Each task has length t and processor requirement p
- Tasks lengths are considered known prior execution
- Goal: minimize makespan T (total time of execution)
- Straightforward heuristic: arrange tasks by decreasing p and schedule them ASAP (no backfilling)

Scheduling Independent Rigid Tasks: A Simple Framework



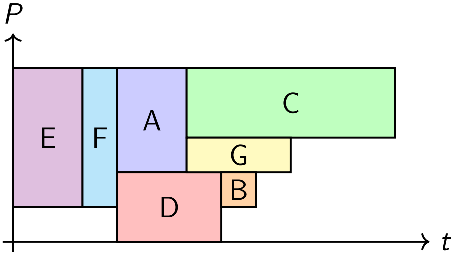
- n tasks to schedule on a platform with P processors
- Each task has length t and processor requirement p
- Tasks lengths are considered known prior execution
- Goal: minimize makespan T (total time of execution)
- Straightforward heuristic: arrange tasks by decreasing p and schedule them ASAP (no backfilling)

Scheduling Independent Rigid Tasks: A Simple Framework



- n tasks to schedule on a platform with P processors
- Each task has length t and processor requirement p
- Tasks lengths are considered known prior execution
- Goal: minimize makespan T (total time of execution)
- Straightforward heuristic: arrange tasks by decreasing p and schedule them ASAP (no backfilling)

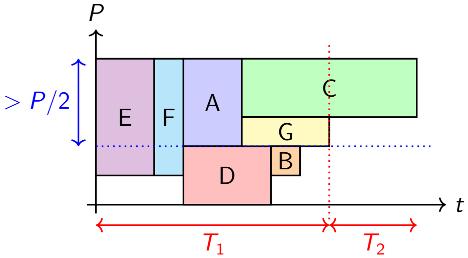
Assessing the Quality of an Heuristic



- Total Area : $\mathcal{A} = \sum t_i p_i$
- Longest task $\mathcal{C} = \max(t_i)$

  $T^{opt} \geq \max(\frac{\mathcal{A}}{P}, \mathcal{C})$

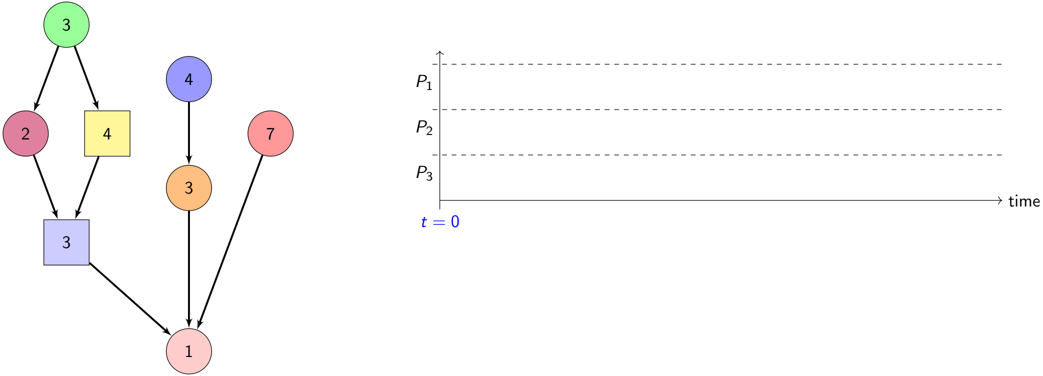
Assessing the Quality of an Heuristic



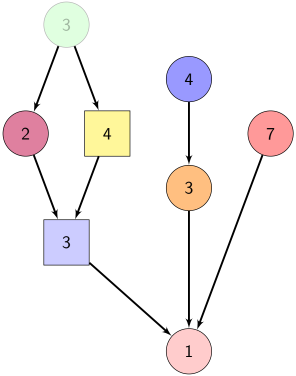
- Total Area : $\mathcal{A} = \sum t_i p_i$
- Longest task $\mathcal{C} = \max(t_i)$
  $T^{opt} \geq \max(\frac{\mathcal{A}}{P}, \mathcal{C})$
- During T_1 we were using more than $\frac{P}{2}$ processors
  $\Rightarrow \frac{P}{2} T_1 \leq \mathcal{A}$
- At $t = 8$ for the first time less than $\frac{P}{2}$ are used $\rightarrow$ all tasks have been scheduled
  $\Rightarrow T_2 \leq \mathcal{C}$

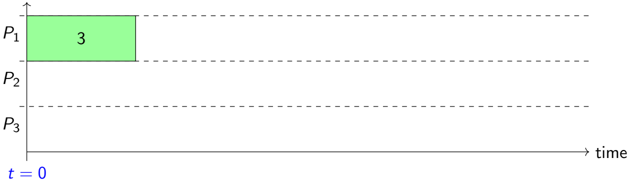
$\Rightarrow T_1 + T_2 \leq 2\frac{\mathcal{A}}{P} + \mathcal{C} \leq 3T^{opt}$
$\Rightarrow$ To remember: We are inefficient during at most $\max(t_i)$

Scheduling a Graph of Tasks (ASAP)

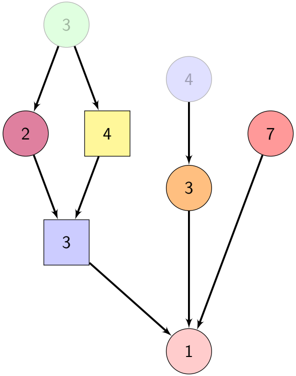


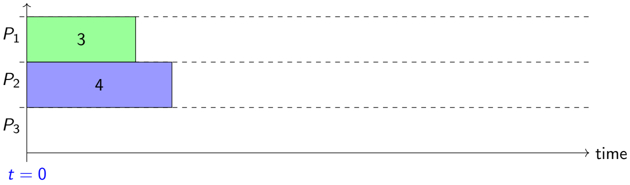
Scheduling a Graph of Tasks (ASAP)



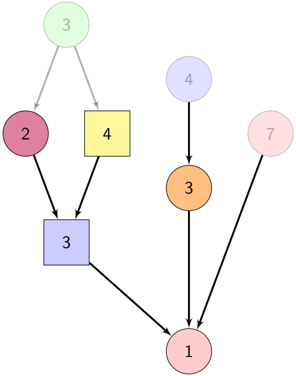


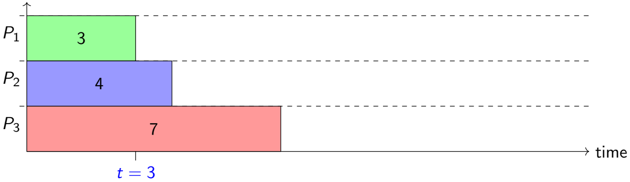
Scheduling a Graph of Tasks (ASAP)





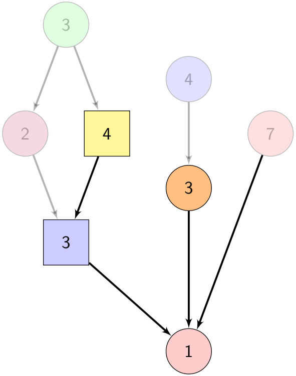
Scheduling a Graph of Tasks (ASAP)

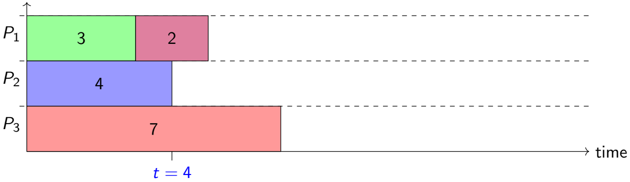




At time $t = 3$, tasks Red(2) and Yellow(4) are ready. Task Yellow(4) needs 2 processors, but only 1 is free. We schedule Red(2)

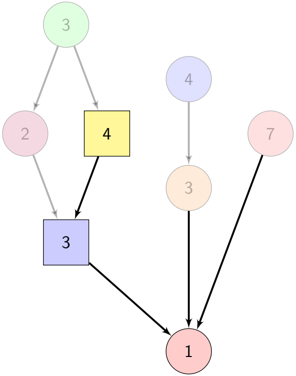
Scheduling a Graph of Tasks (ASAP)

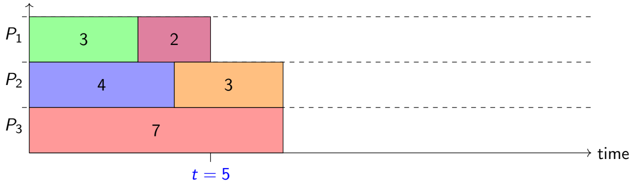




At time $t = 4$, tasks Yellow(4) and Orange(3) are ready. Task Yellow(4) needs 2 processors, but only 1 is free. We schedule Orange(3)

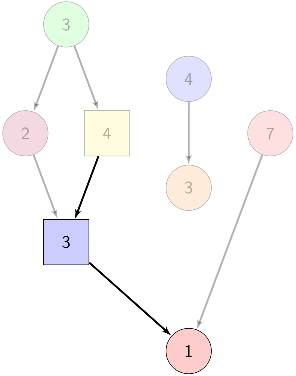
Scheduling a Graph of Tasks (ASAP)

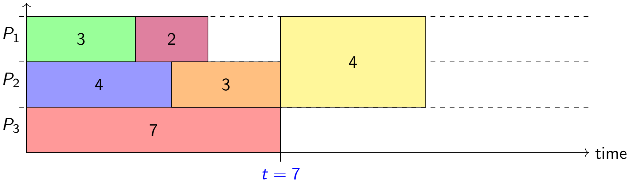




At time $t = 5$, task Yellow(4) is ready, but only one processor is idle

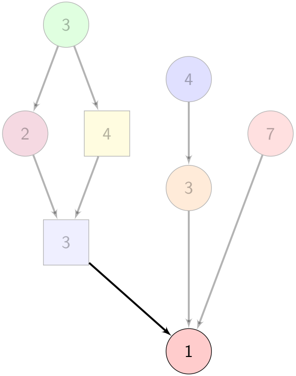
Scheduling a Graph of Tasks (ASAP)

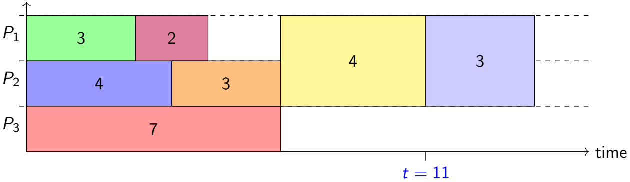




At time $t = 7$ we can finally launch Yellow(4), but a long path remains

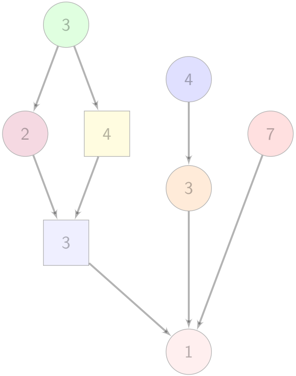
Scheduling a Graph of Tasks (ASAP)

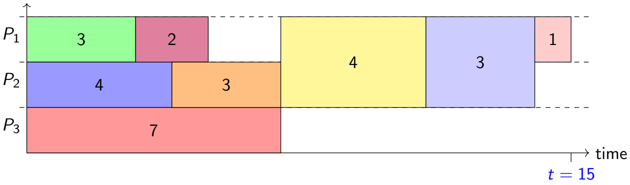




At time $t = 7$ we can finally launch Yellow(4), but a long path remains

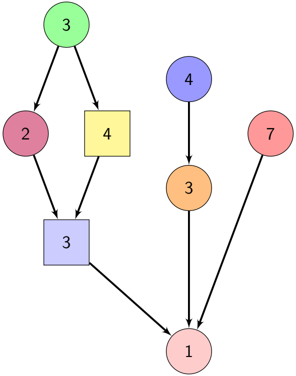
Scheduling a Graph of Tasks (ASAP)

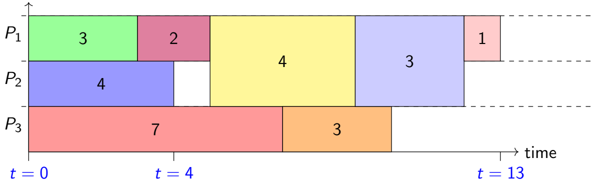




At time $t = 7$ we can finally launch Yellow(4), but a long path remains

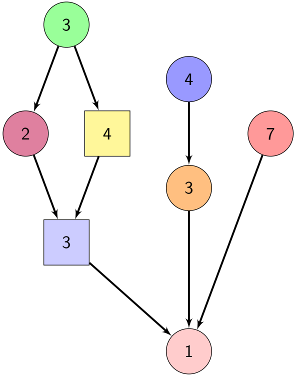
Optimal Schedule: Delay Orange

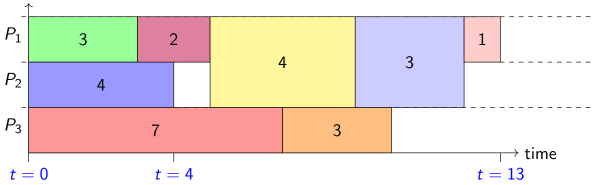




If we chose not to launch anything at $t = 4$ although Orange(3) was ready, Yellow(4) would have started earlier and we would have gained time

Optimal Schedule: Delay Orange

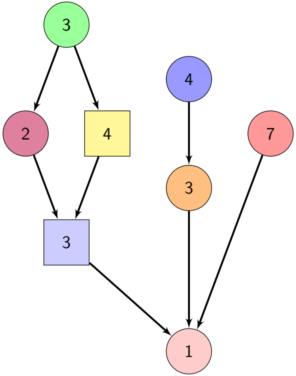


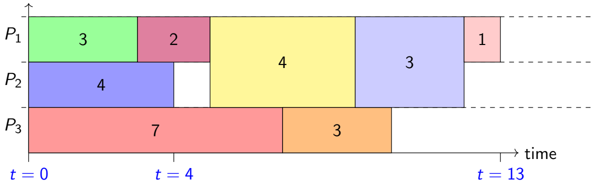


- Longest Path Length : $\mathcal{C} = 11$
- Total Area : $\mathcal{A} = 34$

$\Rightarrow T^{opt} \geq \mathrm{LB} = \max(\frac{\mathcal{A}}{P}, \mathcal{C})$

Optimal Schedule: Delay Orange

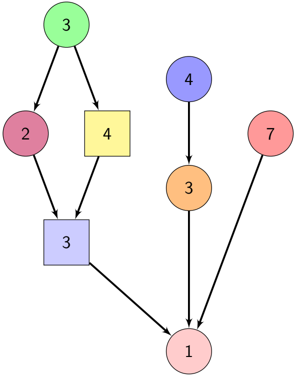


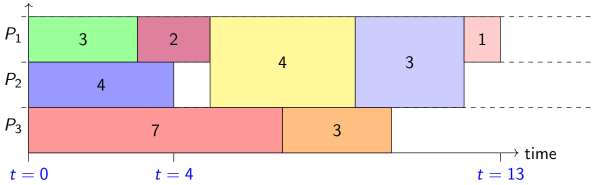


- Best known algorithm is a $2 + \log(n+1)$-approx ($T \leq (2 + \log(n+1))\mathrm{LB}$ for all instance)
- There exists an instance such that $T^{opt} > \frac{\log(n)}{2}\mathrm{LB}$

$\Rightarrow$ $o(\log(n))$-approx would require groundbreaking approaches

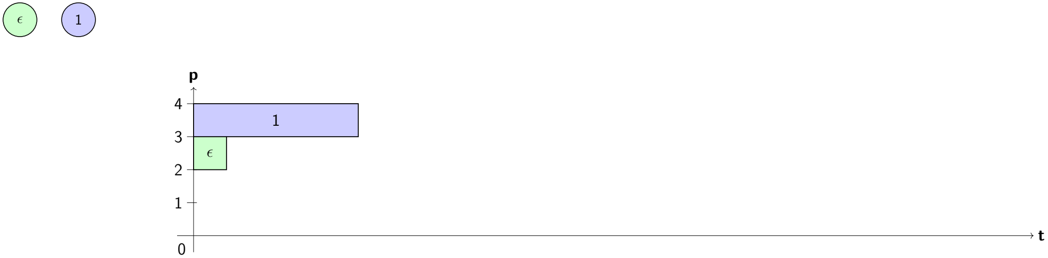
Optimal Schedule: Delay Orange





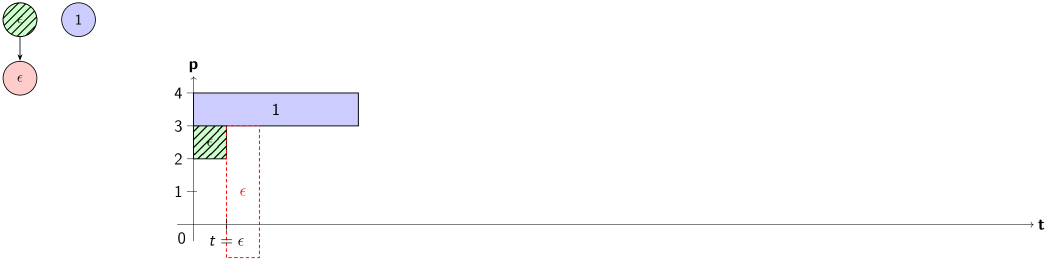
(~~AI-washing~~ Link with AI: This is roughly the same model as the one presented by Oliver Sinnen for DNN Inference on GPU)

Online Version: ASAP? How bad can it get?



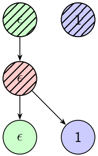
Only available tasks are visible by the scheduler, the rest of the graph is unknown: just launch them?

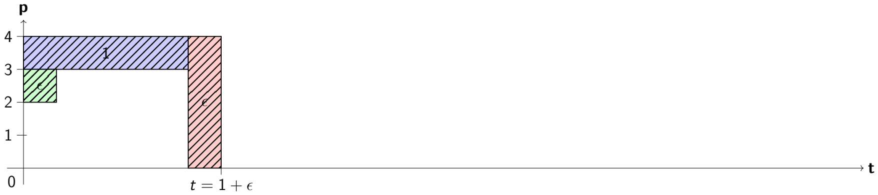
Online Version: ASAP? How bad can it get?



When green ϵ is complete, a new task is revealed, but requires all P processors $\to$ we have to wait until completion of the blue task

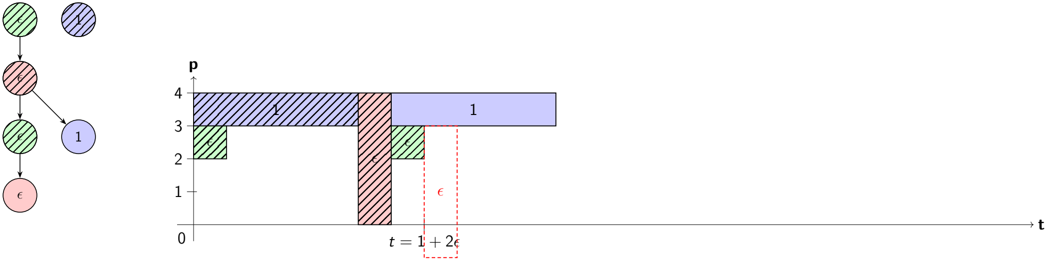
Online Version: ASAP? How bad can it get?





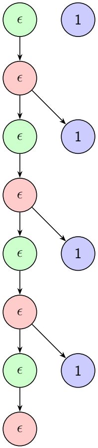
The first blue task had no successor, so we execute the red task, two new tasks are discovered and will be launched

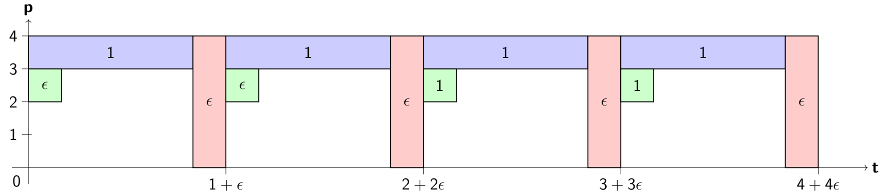
Online Version: ASAP? How bad can it get?



Again, after the green task, a new one is discovered and requires all processors $\rightarrow$ we wait

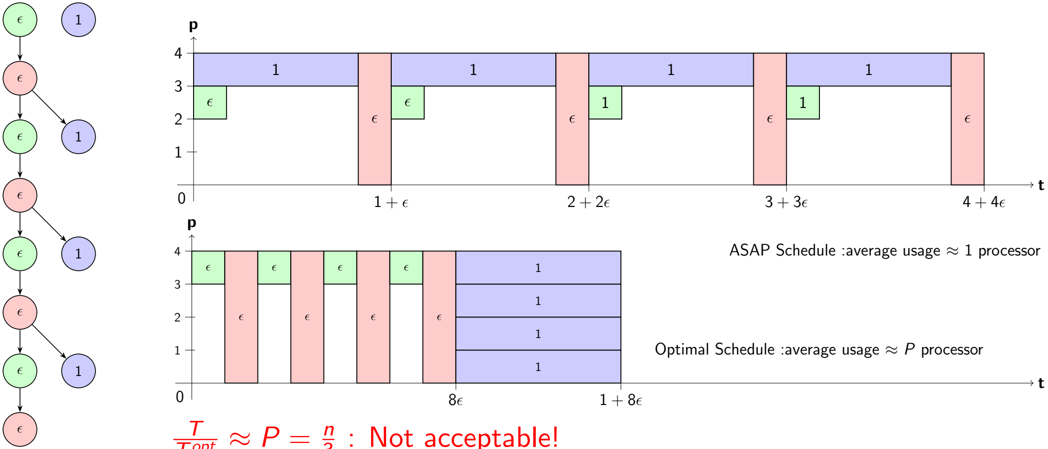
Online Version: ASAP? How bad can it get?





The same situation occurs P times

Online Version: ASAP? How bad can it get?



$\frac{T}{T^{opt}} \approx P = \frac{n}{3}$: Not acceptable!

Goal

- Design an algorithm with a competitive ratio as low as possible. In this work we compare with the lower bound LB: for all instance, $\frac{T}{T_{opt}} \leq \frac{T}{\mathrm{LB}} \leq R^{+}$
- Show that the best possible competitive ratio is above R^{-}: For any algorithm, there exist an (adversarial) instance such that $\frac{T}{T_{opt}} > R^{-}$
- R^{+} and R^{-} should be as close as possible!

Prior Works

- Many similar problems with slightly different flavors; with/without precedence constraints, online/offline, rigid/moldable (where processor allocation can be chosen by the scheduler) were studied. For all of these, good bounds for R^+ (and R^- for online settings) where derived
- For online rigid task graphs:
  - Some competitive ratios for ASAP in specific cases (e.g. number of processors per tasks bounded)
  - Lower bound results for specific ASAP algorithms with different priority rules (longest time first, most processor firsts...)
    $\Rightarrow$ Last slide+ 10 lines proof gives stronger negative results than these results combined

$\Rightarrow$ No algorithms other than ASAP explored, no competitive ratio derived in the most general cases, nor bounds for best possible algorithm! Only problem still among the closely similar models

Summary of Results

Let D be the ratio between the longest and the shortest tasks: $D = \frac{\max(t_i)}{\min(t_i)}$

	Offline Best Alg.	CatBatch (Online)	Best Possible Competitive Ratio
n	$\log(n+1)+2$	$\log(n)+3$	$\frac{\log(n)}{4}$
R	3 when $D=1$	$\log(D)+6$	$\frac{\log(D)}{4}$
P	P	P	$\frac{P}{2}$

- $\Rightarrow$ CatBatch is asymptotically optimal for all metrics

Summary of Results

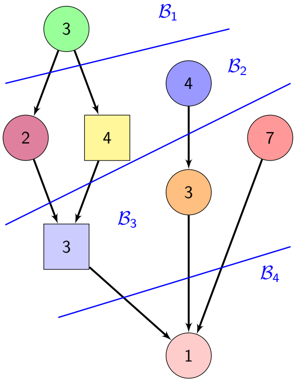
Let D be the ratio between the longest and the shortest tasks: $D = \frac{\max(t_i)}{\min(t_i)}$

	Offline Best Alg.	CatBatch (Online)	Best Possible Competitive Ratio
n	$\log(n+1)+2$	$\log(n)+3$	$\frac{\log(n)}{4}$
R	3 when $D=1$	$\log(D)+6$	$\frac{\log(D)}{4}$
P	P	P	$\frac{P}{2}$

- $\Rightarrow$ CatBatch is asymptotically optimal for all metrics
- The ratios for CatBatch are derived in respect to $\mathrm{LB} = \max(\frac{A}{P}, \mathcal{C})$: we always have $(\frac{T}{T^{opt}} \leq) \frac{T}{\mathrm{LB}} \leq \log(n)+3$. Surprising since $\frac{T^{opt}}{\mathrm{LB}} > \frac{\log(n)}{2}$ is possible.
- Best Possibles are derived with an adversary: any algorithm may have $\frac{T}{T^{opt}} > \frac{\log(n)}{4+\epsilon}$

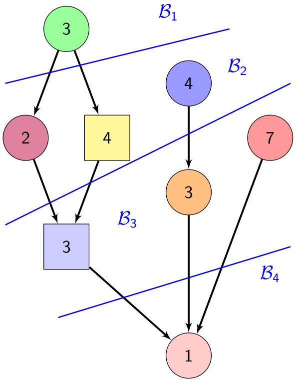
Outline

Idea 1: Split the Graph in Batches of Independent Tasks...



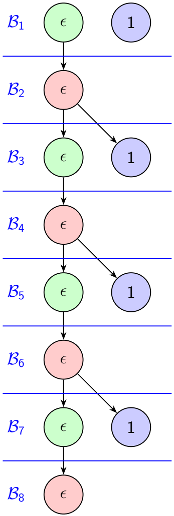
- We can process batch of independent tasks efficiently ($\leq \frac{P}{2}$ processors are used during at most $\max(t_i)$)
  $\Rightarrow$ We split the graph into batches that will be processed one after the other

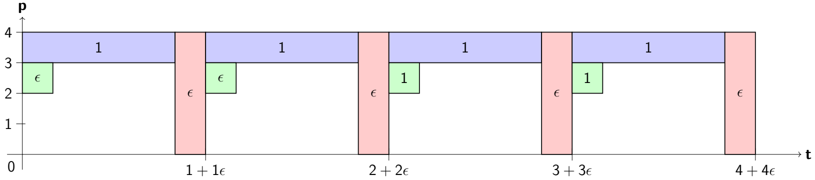
Idea 1: Split the Graph in Batches of Independent Tasks...



- We can process batch of independent tasks efficiently ($\leq \frac{P}{2}$ processors are used during at most $\max(t_i)$)
  $\Rightarrow$ We split the graph into batches that will be processed one after the other
- We must make sure that all predecessors of tasks in a batch have been completed in previous batches
- Batches must be computed online

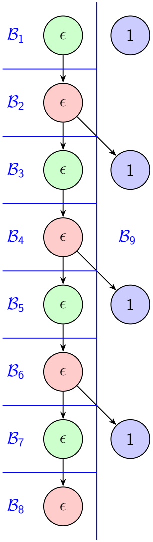
Idea 1: Split the Graph in Batches of Independent Tasks... But How?

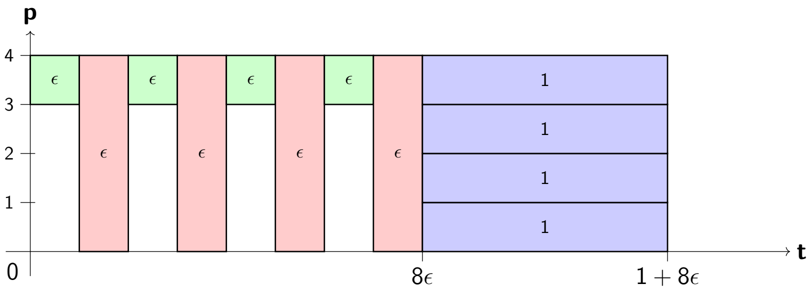




If we split into batches from top to bottom, the schedule is again, very inefficient

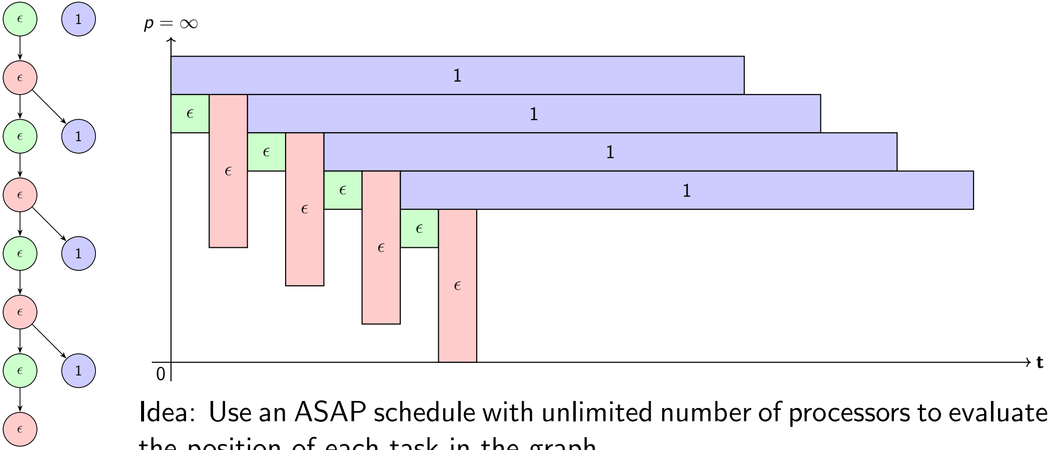
Idea 1: Split the Graph in Batches of Independent Tasks... But How?





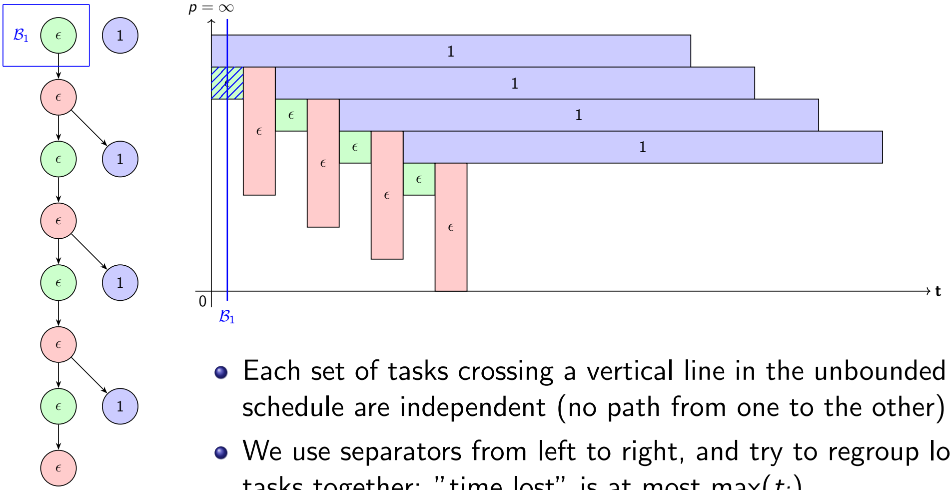
- Best schedule is obtained by splitting the graph left/right first, then splitting left side from top to bottom
- But we have to wait until full completion of previous batches before starting a new batch!

Idea 2: Be Fair, and Regroup Long Tasks Together



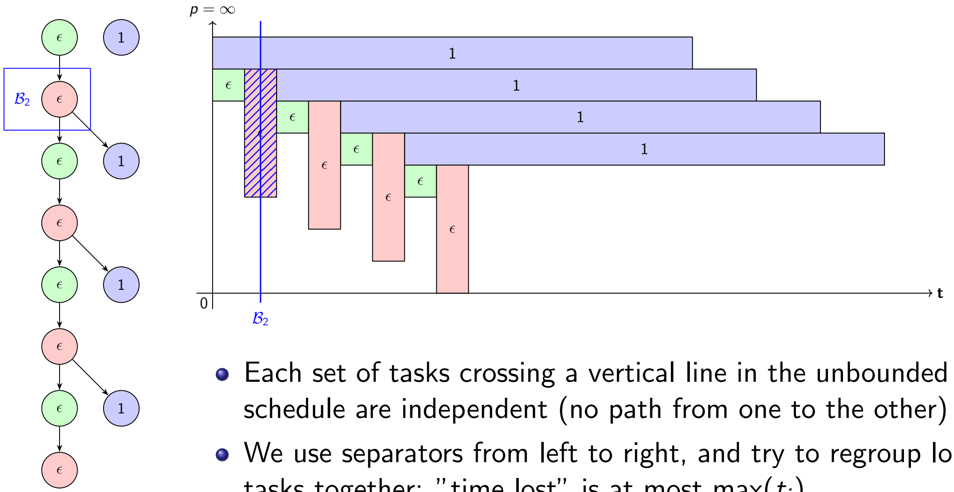
Idea: Use an ASAP schedule with unlimited number of processors to evaluate the position of each task in the graph

Idea 2: Be Fair, and Regroup Long Tasks Together



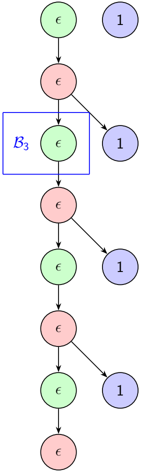
- Each set of tasks crossing a vertical line in the unbounded ASAP schedule are independent (no path from one to the other)
- We use separators from left to right, and try to regroup long tasks together: "time lost" is at most $\max(t_i)$

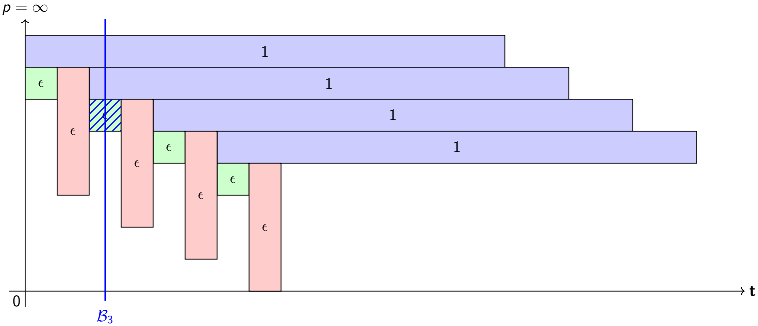
Idea 2: Be Fair, and Regroup Long Tasks Together



- Each set of tasks crossing a vertical line in the unbounded ASAP schedule are independent (no path from one to the other)
- We use separators from left to right, and try to regroup long tasks together: "time lost" is at most $\max(t_i)$

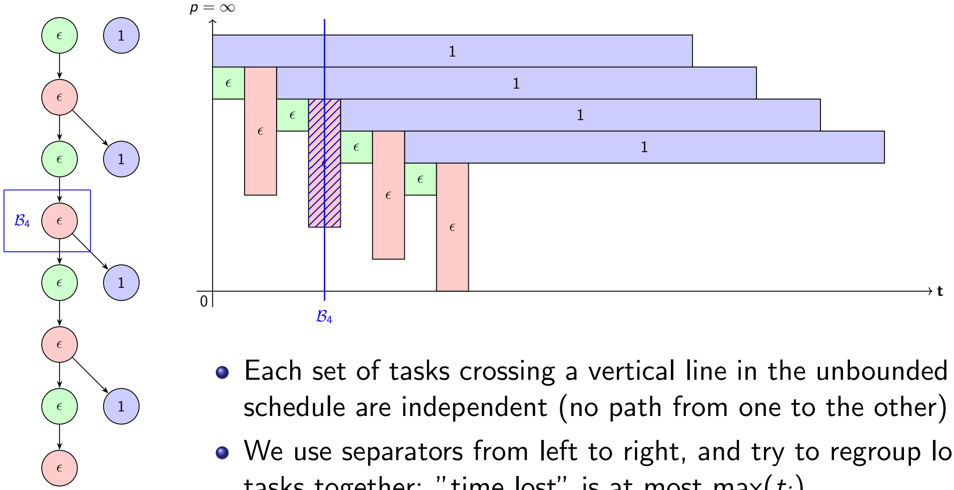
Idea 2: Be Fair, and Regroup Long Tasks Together





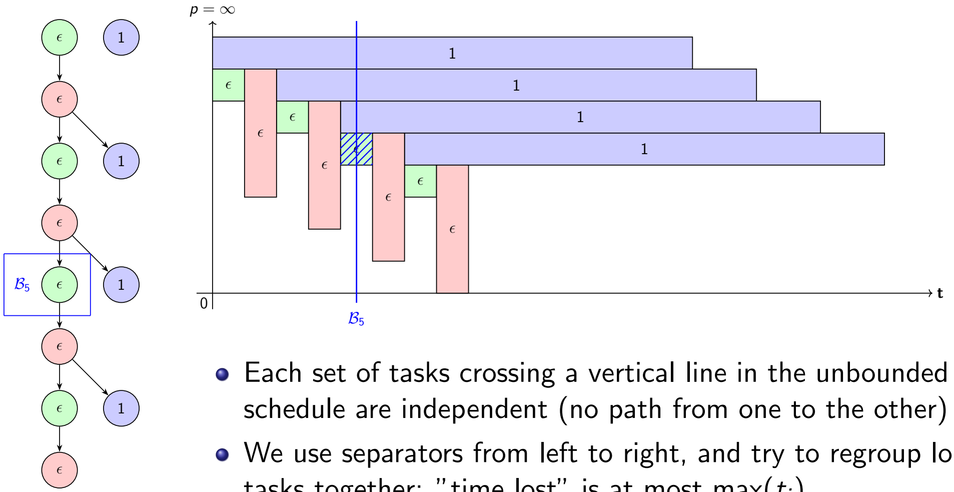
- Each set of tasks crossing a vertical line in the unbounded ASAP schedule are independent (no path from one to the other)
- We use separators from left to right, and try to regroup long tasks together: "time lost" is at most $\max(t_i)$

Idea 2: Be Fair, and Regroup Long Tasks Together



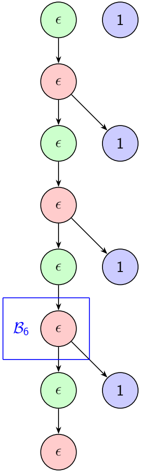
- Each set of tasks crossing a vertical line in the unbounded ASAP schedule are independent (no path from one to the other)
- We use separators from left to right, and try to regroup long tasks together: "time lost" is at most $\max(t_i)$

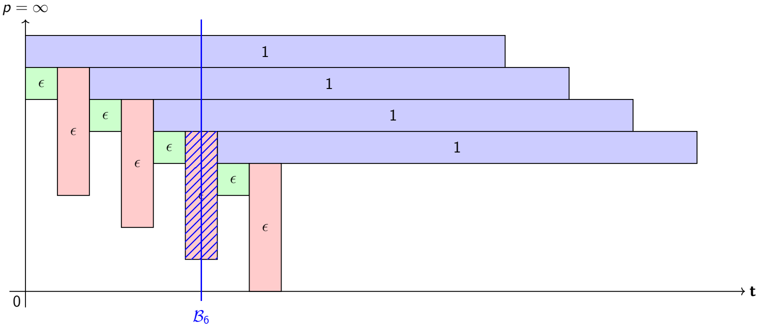
Idea 2: Be Fair, and Regroup Long Tasks Together



- Each set of tasks crossing a vertical line in the unbounded ASAP schedule are independent (no path from one to the other)
- We use separators from left to right, and try to regroup long tasks together: "time lost" is at most $\max(t_i)$

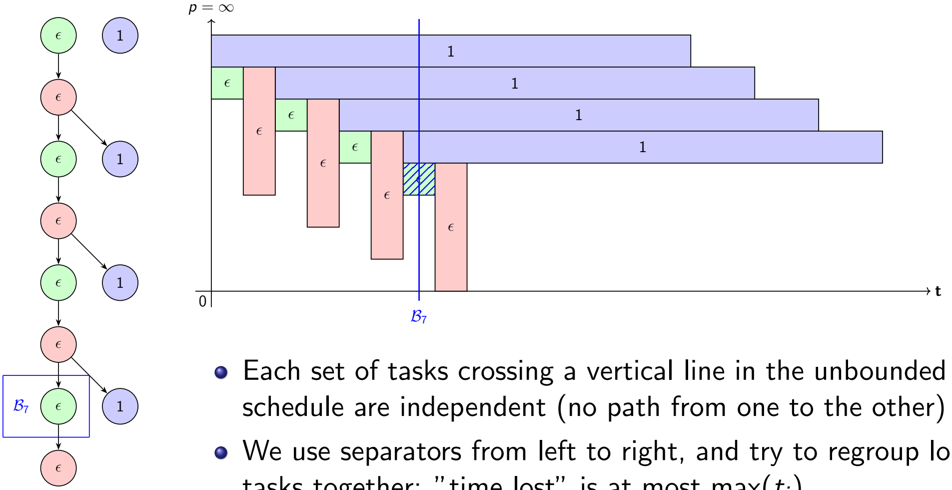
Idea 2: Be Fair, and Regroup Long Tasks Together





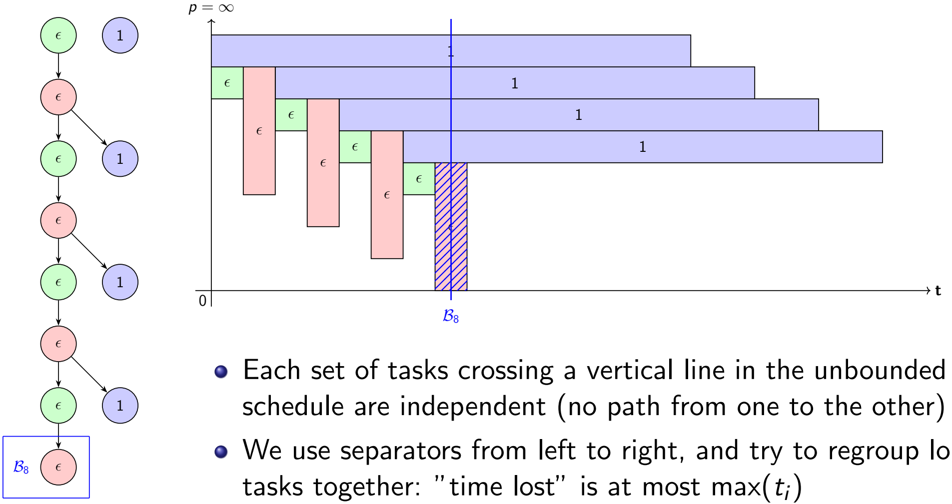
- Each set of tasks crossing a vertical line in the unbounded ASAP schedule are independent (no path from one to the other)
- We use separators from left to right, and try to regroup long tasks together: "time lost" is at most $\max(t_i)$

Idea 2: Be Fair, and Regroup Long Tasks Together



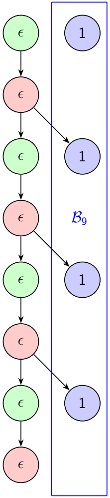
- Each set of tasks crossing a vertical line in the unbounded ASAP schedule are independent (no path from one to the other)
- We use separators from left to right, and try to regroup long tasks together: "time lost" is at most $\max(t_i)$

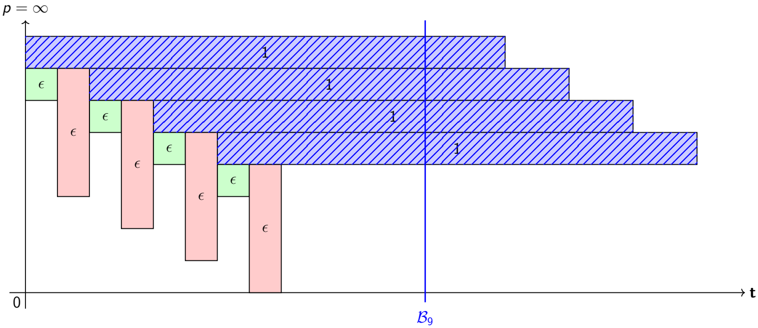
Idea 2: Be Fair, and Regroup Long Tasks Together



- Each set of tasks crossing a vertical line in the unbounded ASAP schedule are independent (no path from one to the other)
- We use separators from left to right, and try to regroup long tasks together: "time lost" is at most $\max(t_i)$

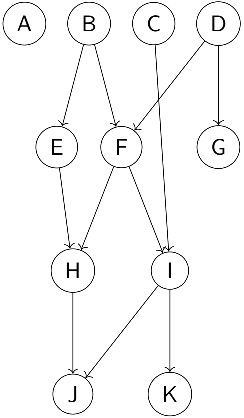
Idea 2: Be Fair, and Regroup Long Tasks Together

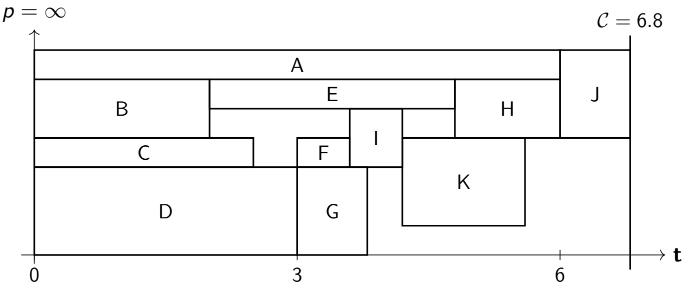




How to do this online??

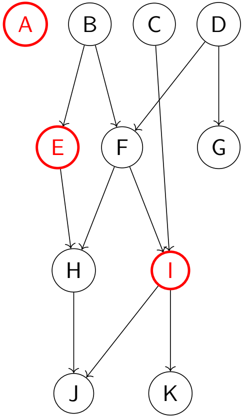
Solution: Categories!

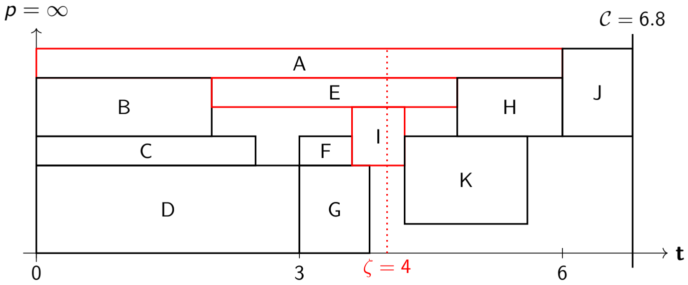




We first assume the ASAP schedule is known to better understand concepts (how to compute categories online will be shown later)

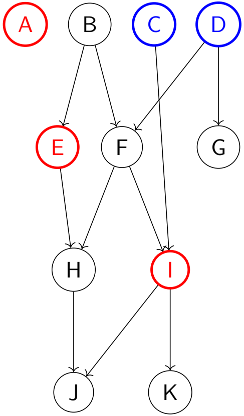
Solution: Categories!

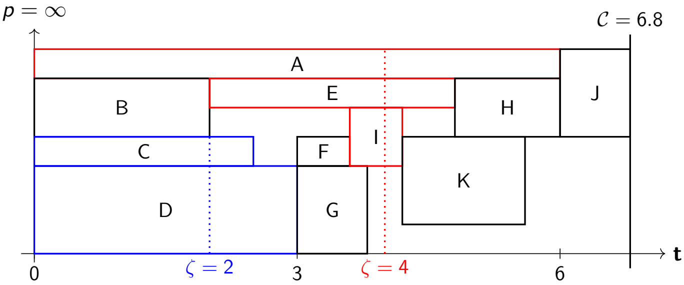




The dominating category includes all tasks crossing the highest power of two in $[0, \mathcal{C})$, here $\zeta = 2^2$, corresponding to A, E, I

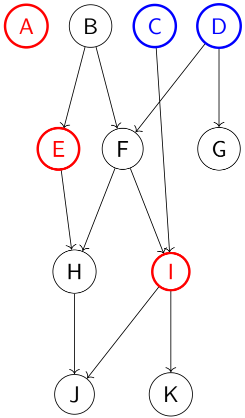
Solution: Categories!

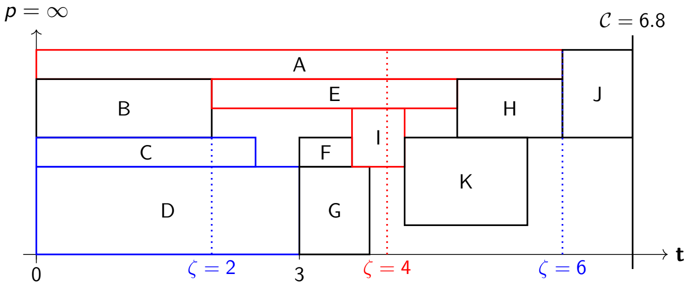




- The second highest layer are the tasks crossing $k2^1$, k odd. If $k = 1$, C and D get into category 2
- Note that tasks of category 2 may not be longer than 4, otherwise it would have crossed the $\zeta = 4$ mark

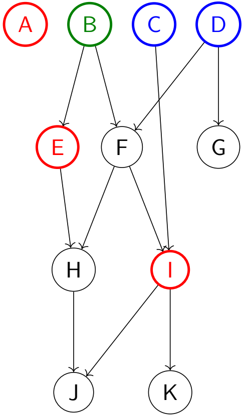
Solution: Categories!

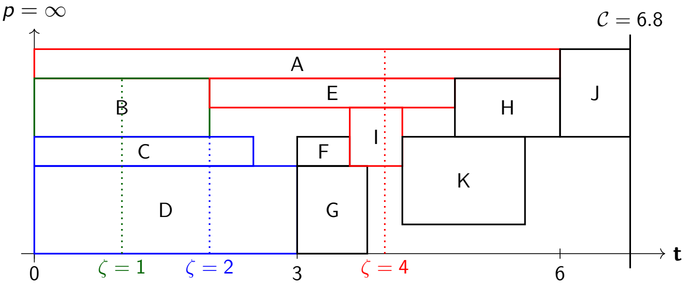




With $k = 3$, i.e. $\zeta = 6$, no tasks cross this line

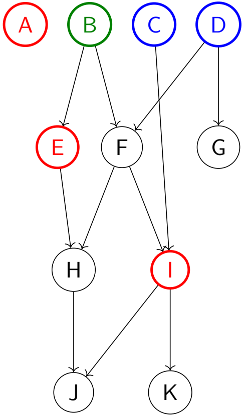
Solution: Categories!

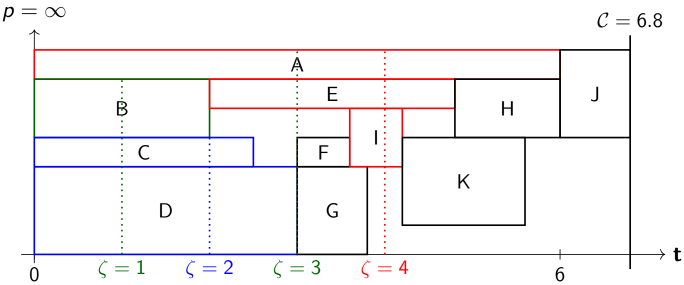




- Now tasks crossing any line $\zeta = k2^0$ are selected (k odd). With $k = 1$, B gets into category $\zeta = 1$
- Note that tasks of category $k2^0$ may not be longer than 2, otherwise it would have crossed a category $\zeta = k2^1$

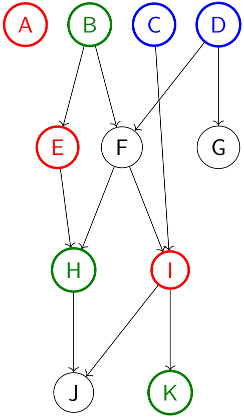
Solution: Categories!

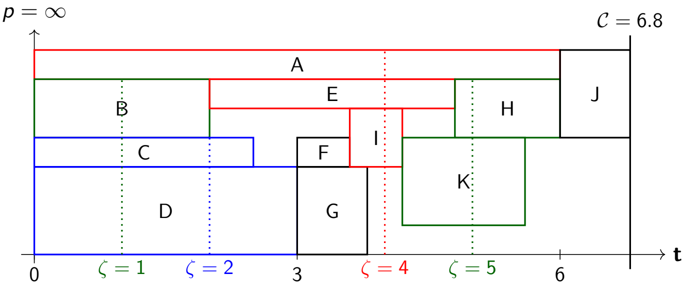




With $k = 3$, i.e. $\zeta = 3$, no tasks cross this line

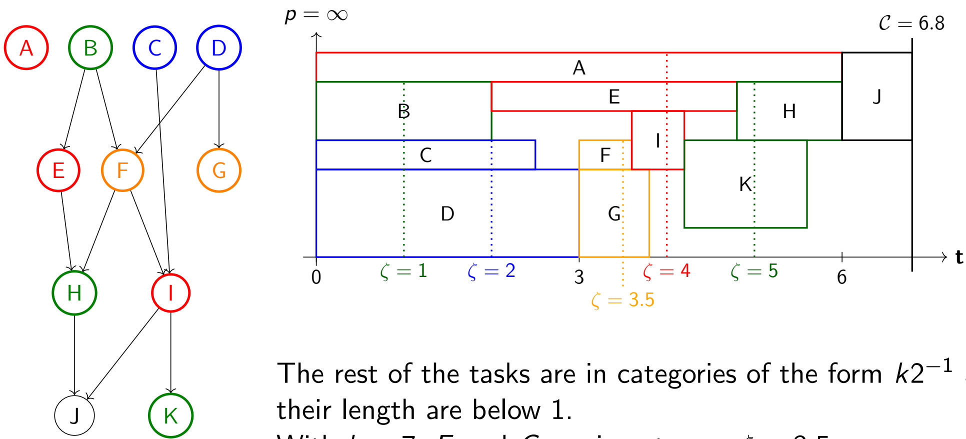
Solution: Categories!





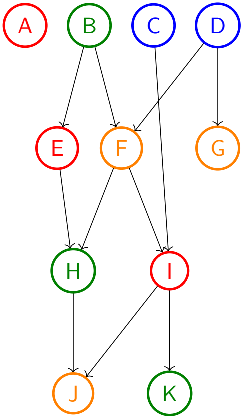
With $k = 5$, i.e. $\zeta = 5$, H and K cross the line

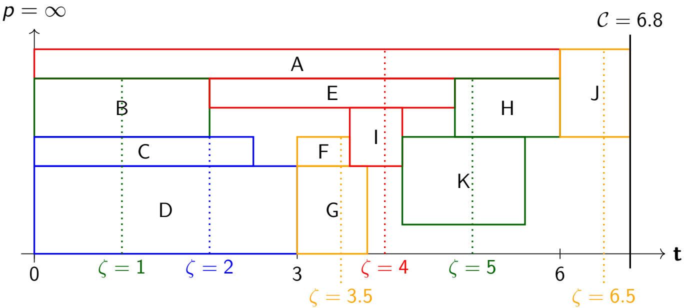
Solution: Categories!



The rest of the tasks are in categories of the form $k2^{-1}$ and their length are below 1.
With $k = 7$, F and G are in category $\zeta = 3.5$

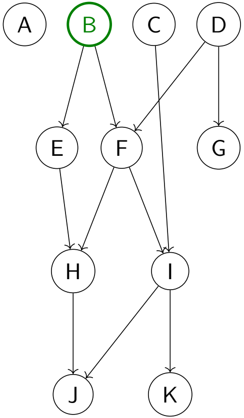
Solution: Categories!

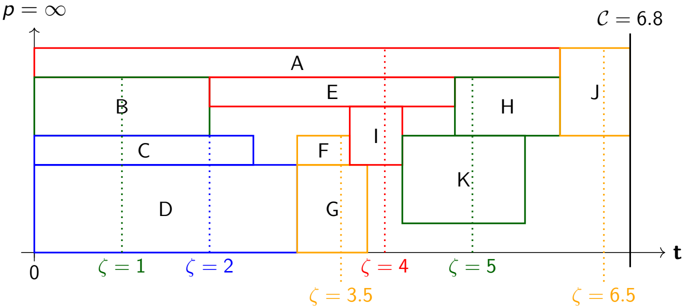




With $k = 13$, J is in category $\zeta = 6.5$

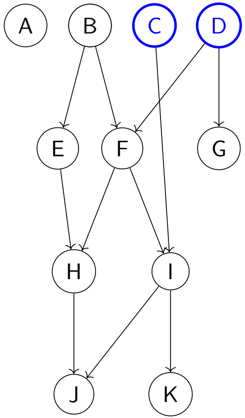
Solution: Categories!

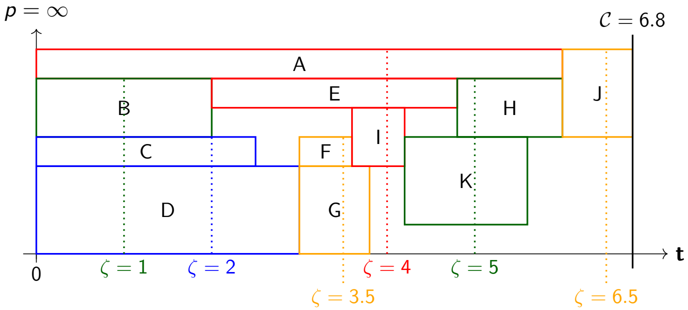




Let's see them again in order !
First category, $\zeta = 1 \times 2^0 = 1$: B

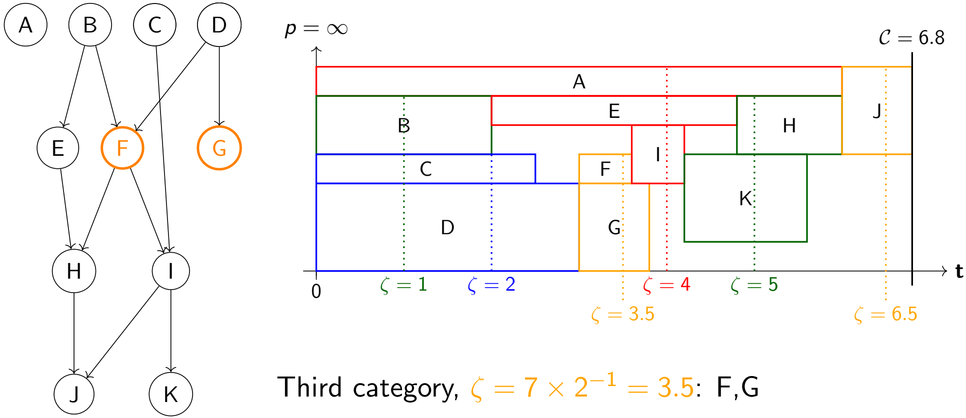
Solution: Categories!





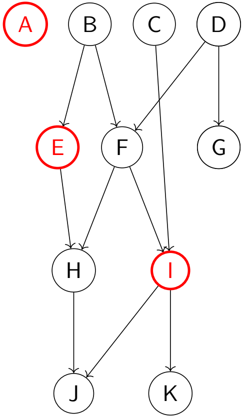
Second category, $\zeta = 1 \times 2^1 = 2$: C,D

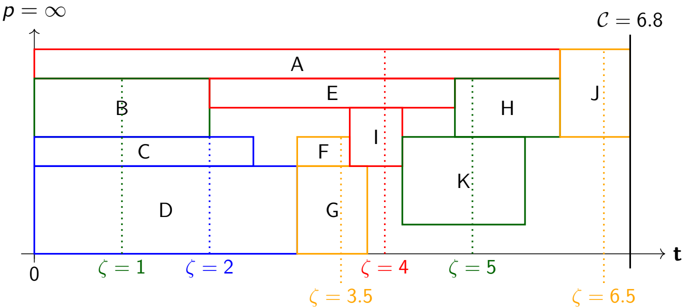
Solution: Categories!



Third category, $\zeta = 7 \times 2^{-1} = 3.5$: F,G

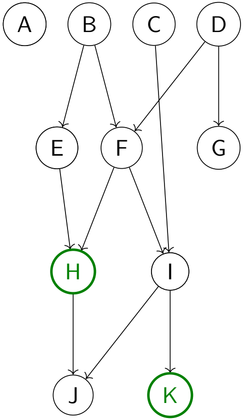
Solution: Categories!

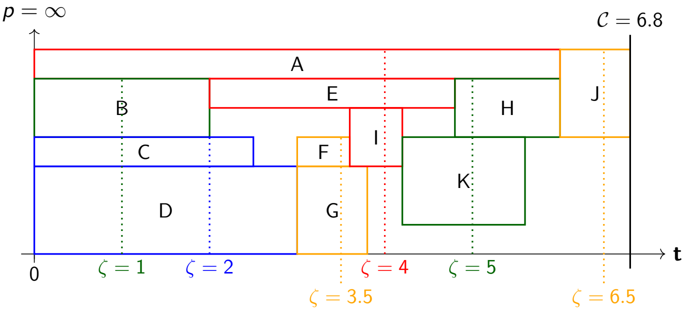




Fourth category, $\zeta = 1 \times 2^2 = 4$: A, E, I

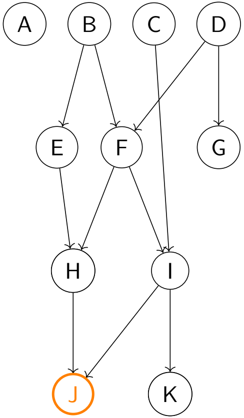
Solution: Categories!

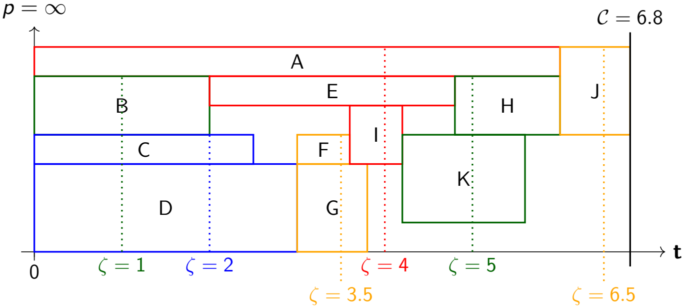




Fifth category, $\zeta = 5 \times 2^1$: H, K

Solution: Categories!





Sixth category, $\zeta = 13 \times 2^{-1} = 6.5$: J

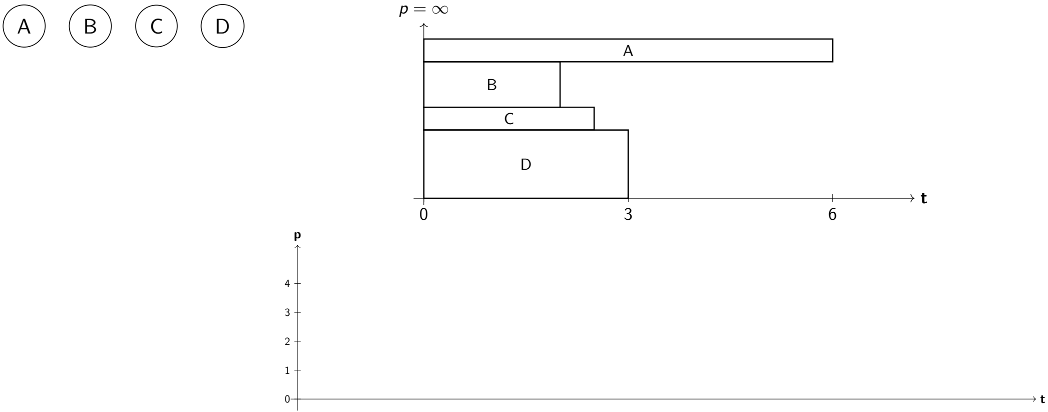
Outline

CatBatch Algorithm

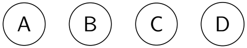
Repeat until all tasks are scheduled:

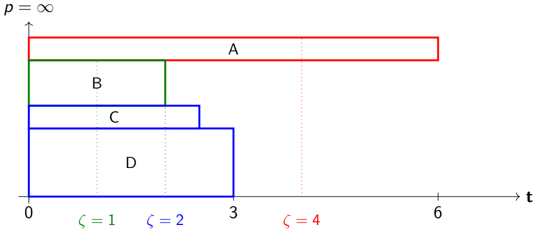
- Phase A/ Compute the category of all newly discovered tasks
- Phase B/ Greedily schedule the batch of tasks of smallest category

BatchCat Run: Example



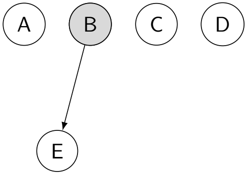
BatchCat Run: Example

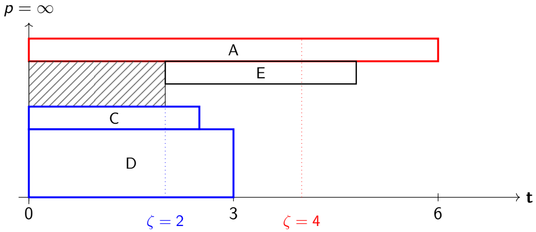


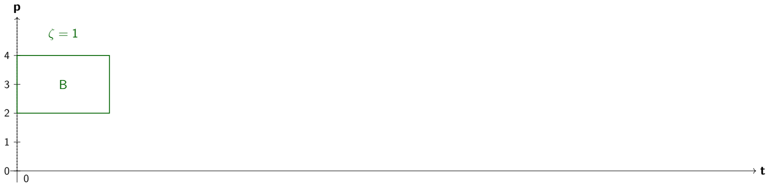


Phase A/ Compute the category of all newly discovered tasks

BatchCat Run: Example

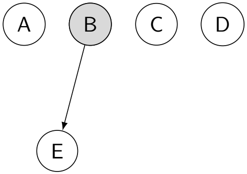


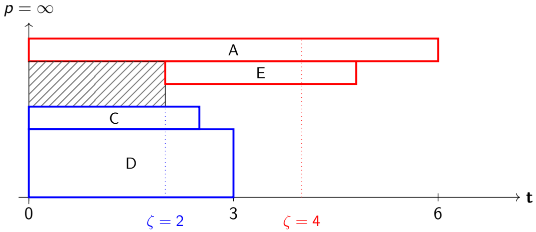


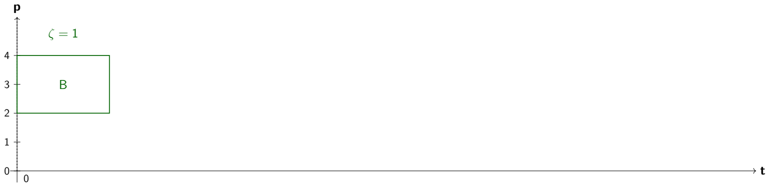


Phase B/ Greedily schedule tasks of smallest category

BatchCat Run: Example

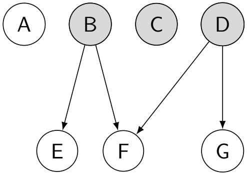


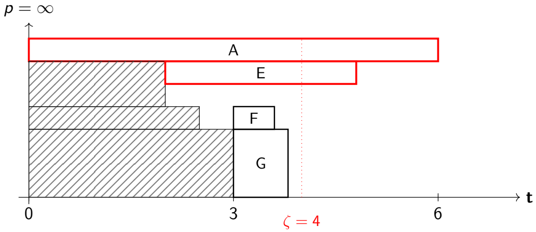


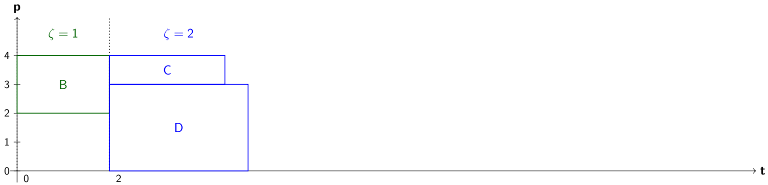


Phase A/ Compute the category of all newly discovered tasks

BatchCat Run: Example

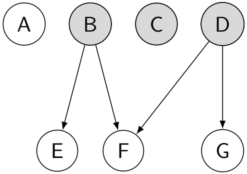


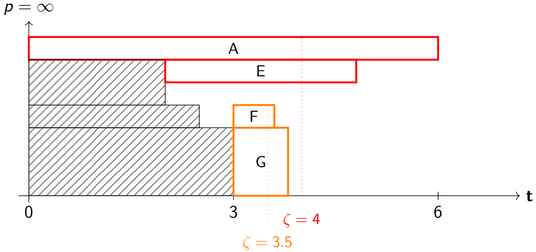


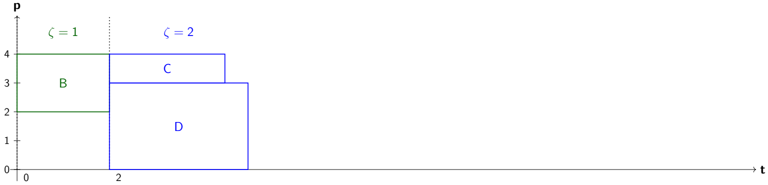


Phase B/ Greedily schedule tasks of smallest category

BatchCat Run: Example

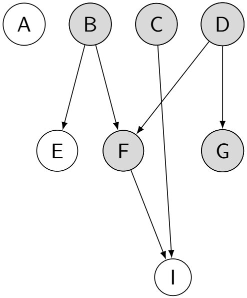


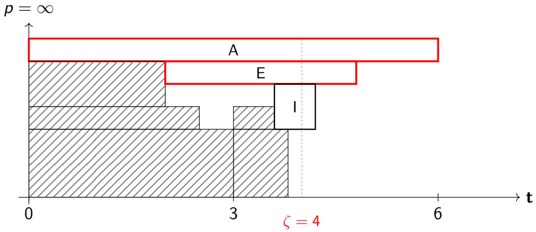


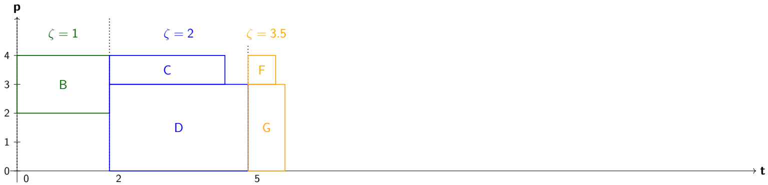


Phase A/ Compute the category of all newly discovered tasks

BatchCat Run: Example

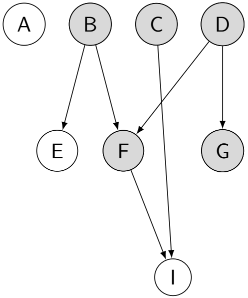


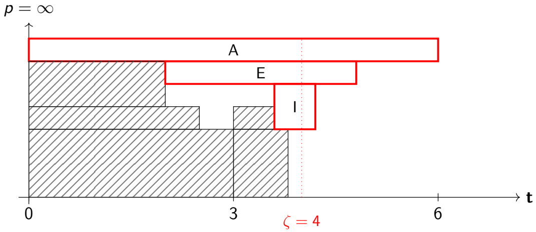


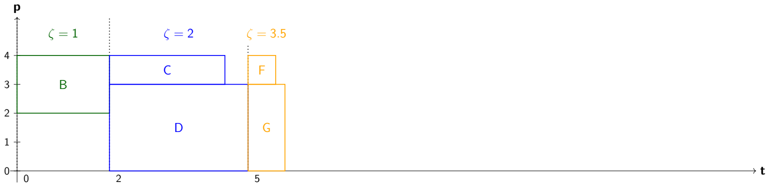


Phase B/ Greedily schedule tasks of smallest category

BatchCat Run: Example

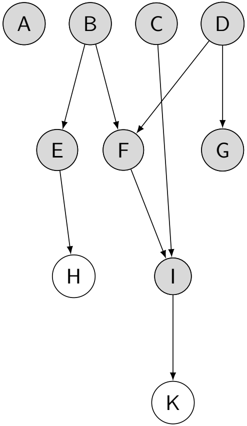


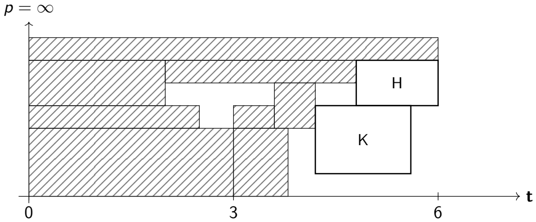


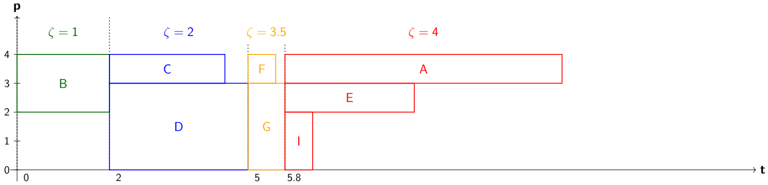


Phase A/ Compute the category of all newly discovered tasks

BatchCat Run: Example

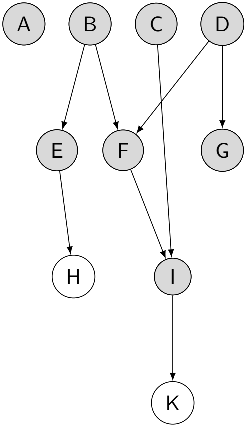


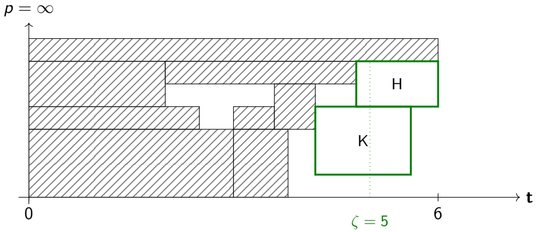


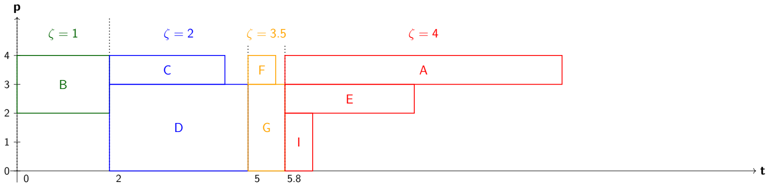


Phase B/ Greedily schedule tasks of smallest category

BatchCat Run: Example

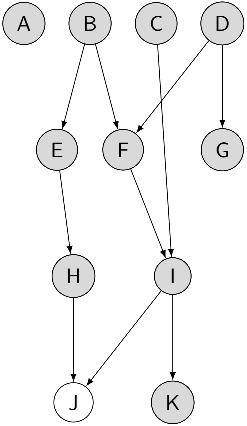


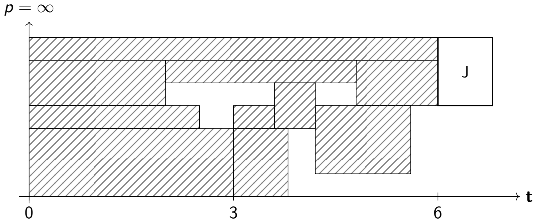


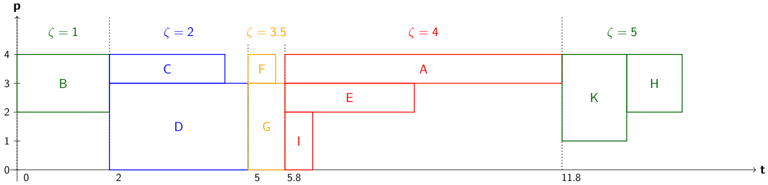


Phase A/ Compute the category of all newly discovered tasks

BatchCat Run: Example

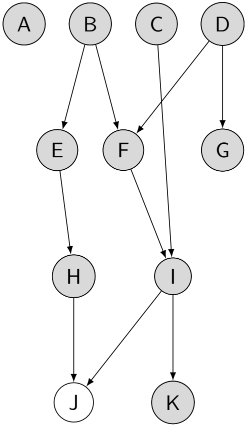


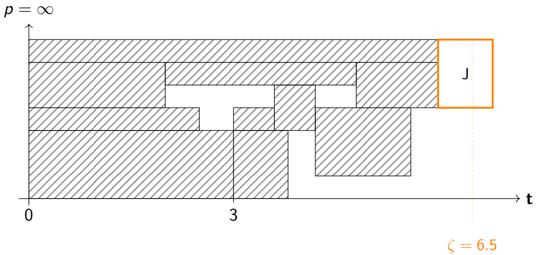


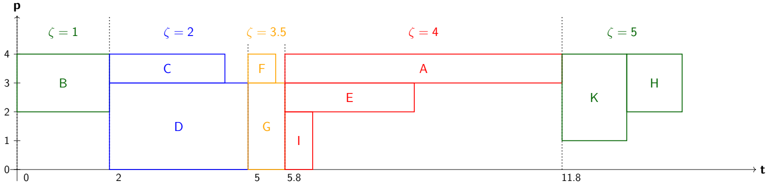


Phase B/ Greedily schedule tasks of smallest category

BatchCat Run: Example

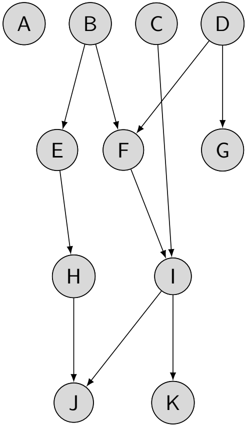


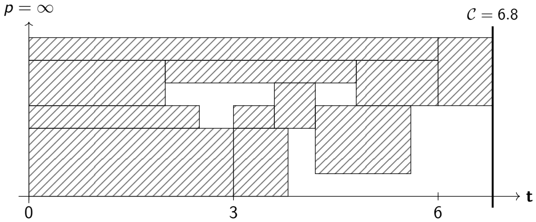


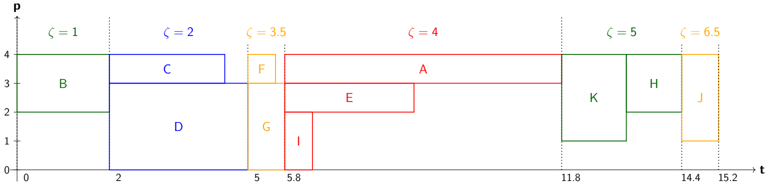


Phase A/ Compute the category of all newly discovered tasks

BatchCat Run: Example

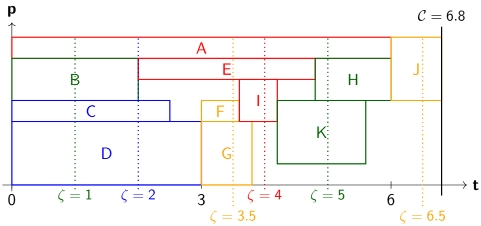


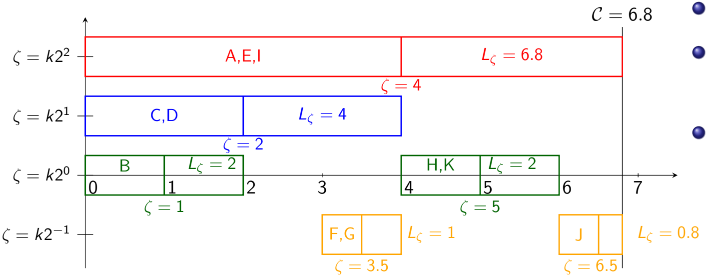




Phase B/ Greedily schedule tasks of smallest category

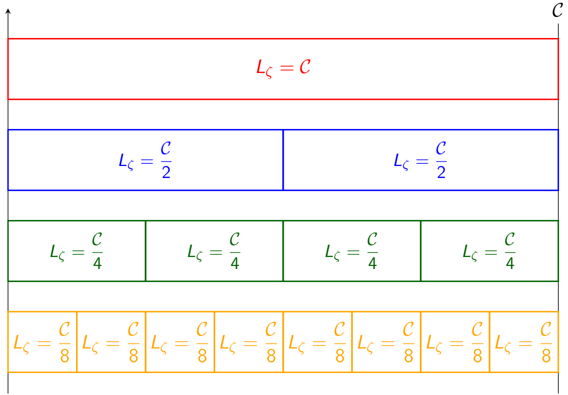
Why is CatBatch Good (in the worst case)





- Let L_ζ denotes the length of the longest possible task of category $\zeta = k2^\chi$
- $L_\zeta \leq 2^{\chi+1}$
- For each category, $T_\zeta \leq 2\frac{a_{cat}}{P} + L_\zeta$
- Overall, $T \leq 2\frac{\mathcal{A}}{P} + \sum L_\zeta$

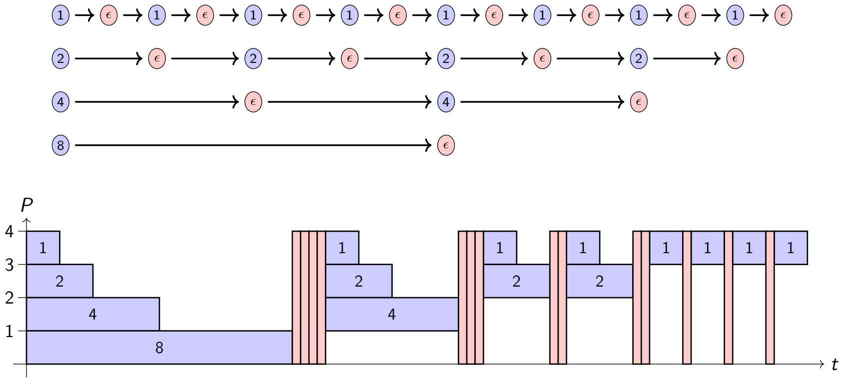
Why is CatBatch Good (in the worst case)



- Worst case, 1 task per category, categories as high as possible
- Approximate worse case: $T \leq 2\frac{\mathcal{A}}{P} + \mathcal{C} + \frac{\mathcal{C}}{2} + \frac{\mathcal{C}}{2} + \frac{\mathcal{C}}{4} + \frac{\mathcal{C}}{4} + \frac{\mathcal{C}}{4} + \frac{\mathcal{C}}{4} + \dots$
- Approximate worse case: $T \leq 2\frac{\mathcal{A}}{P} + \log(n)\mathcal{C}$
- Exact result: $\frac{T}{\mathrm{LB}} \leq \log(n) + 3$

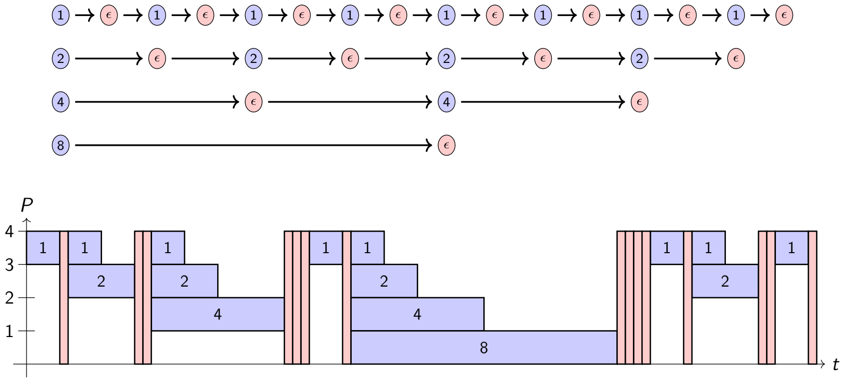
Outline

Instance where Efficiency is Impossible



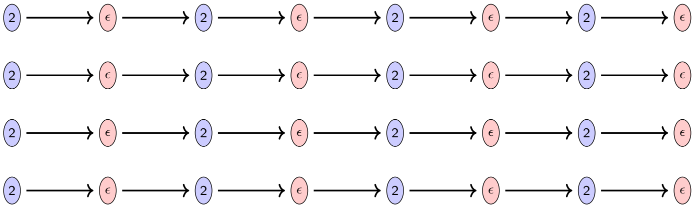
- Blue tasks use 1 processor, red tasks use P processors
- No matter how you schedule them, Average number processors used < 2 !

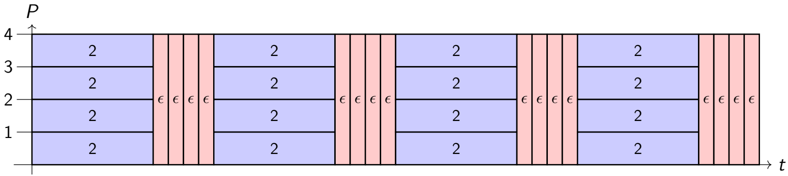
Instance where Efficiency is Impossible



- Blue tasks use 1 processor, red tasks use P processors
- No matter how you schedule them, Average number processors used < 2 !

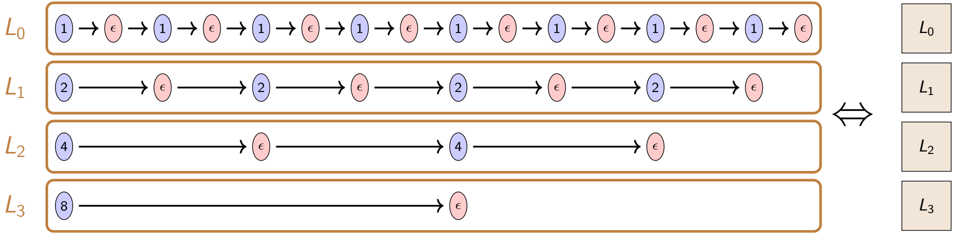
With Identical Chains, Perfect Schedule is Possible!





If all chains are identical (for any chain), a scheduler can repeatedly process a layer of blue tasks in parallel, then process all the subsequent red tasks sequentially

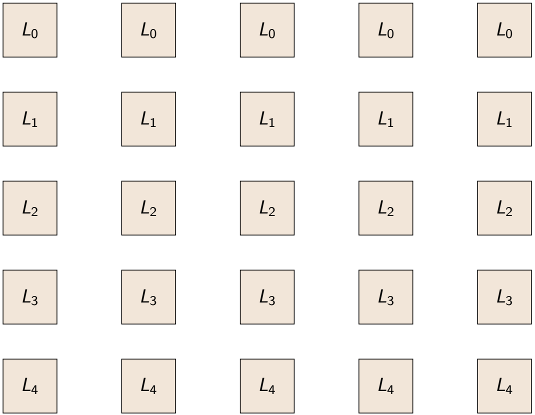
Inefficient Instance: Compact Expression



To simplify notation, L_i represents a chain of tasks of length 2^i separated by ϵ red tasks. Key points:

- If we schedule a layer of different L_is, the optimal average processor usage is below $\bar{p} = 2$
- If we schedule identical L_is, the optimal average processor usage is $\bar{p} = P$

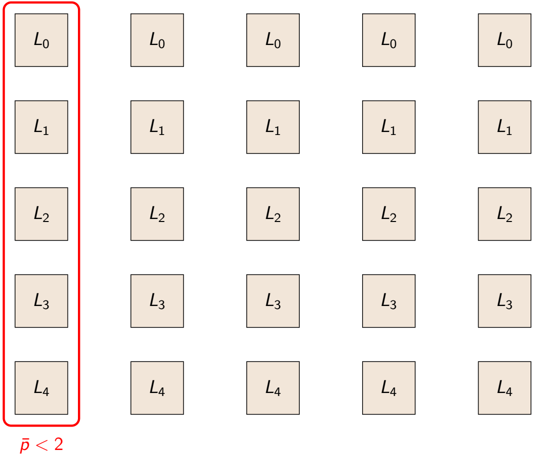
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

$\Rightarrow$ If the layers are processed from left to right, the processor usage is $\bar{p} < 2$.

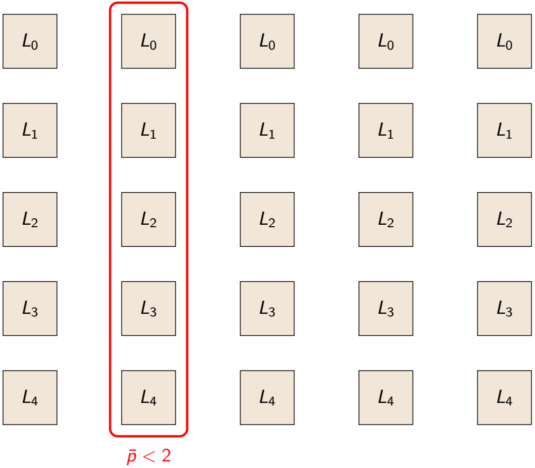
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

$\Rightarrow$ If the layers are processed from left to right, the processor usage is $\bar{p} < 2$.

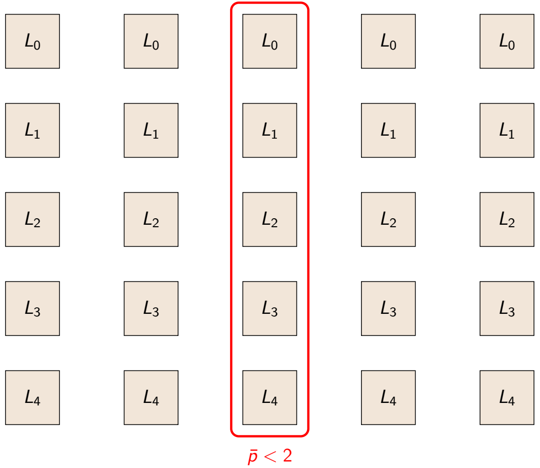
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

$\Rightarrow$ If the layers are processed from left to right, the processor usage is $\bar{p} < 2$.

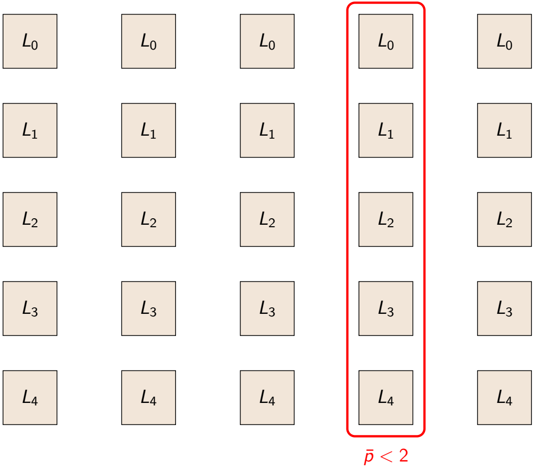
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

$\Rightarrow$ If the layers are processed from left to right, the processor usage is $\bar{p} < 2$.

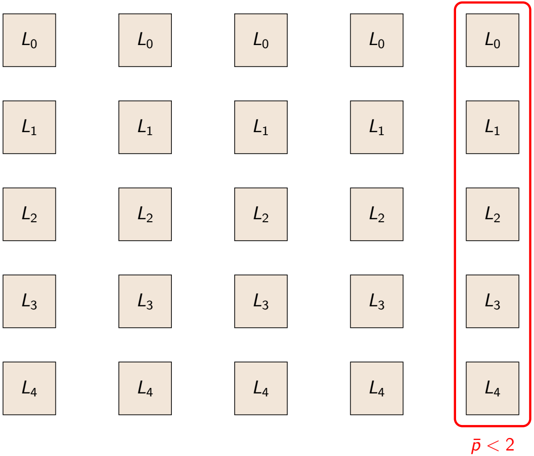
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

$\Rightarrow$ If the layers are processed from left to right, the processor usage is $\bar{p} < 2$.

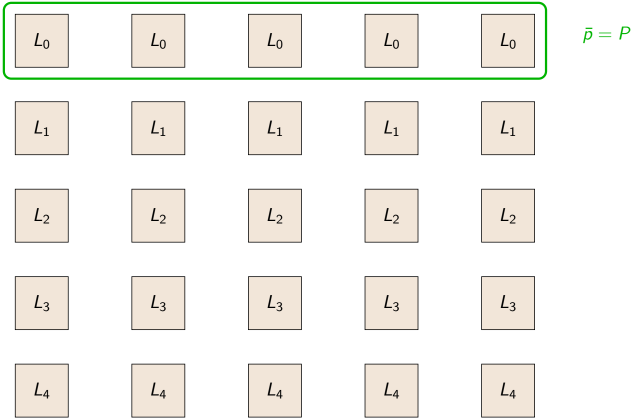
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

$\Rightarrow$ If the layers are processed from left to right, the processor usage is $\bar{p} < 2$.

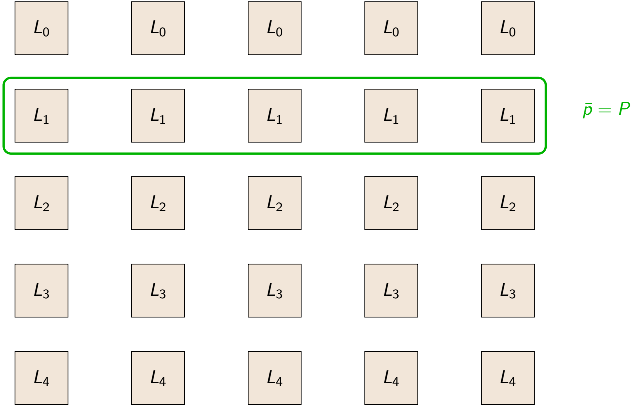
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

$\Rightarrow$ If the layers are processed from top to bottom, the processor usage is $\bar{p} < P$.

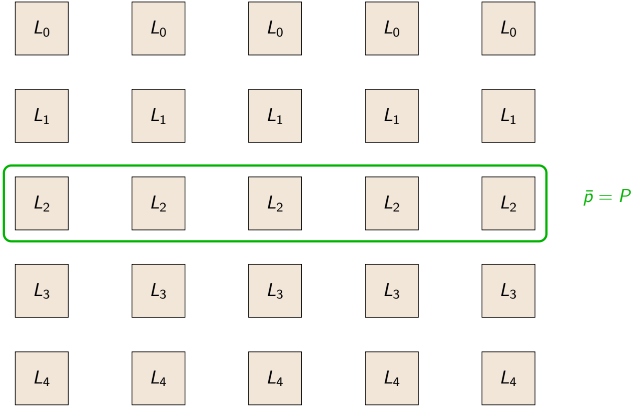
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

$\Rightarrow$ If the layers are processed from top to bottom, the processor usage is $\bar{p} < P$.

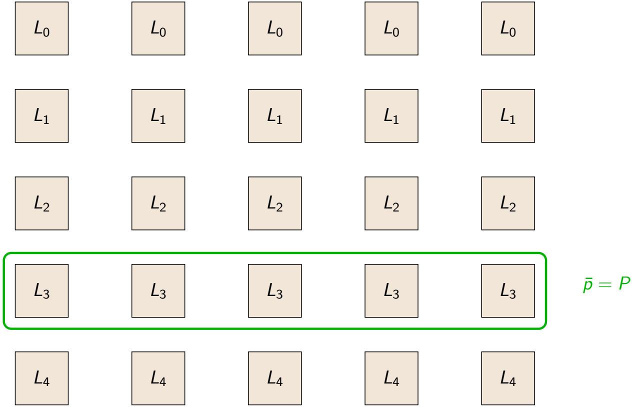
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

$\Rightarrow$ If the layers are processed from top to bottom, the processor usage is $\bar{p} < P$.

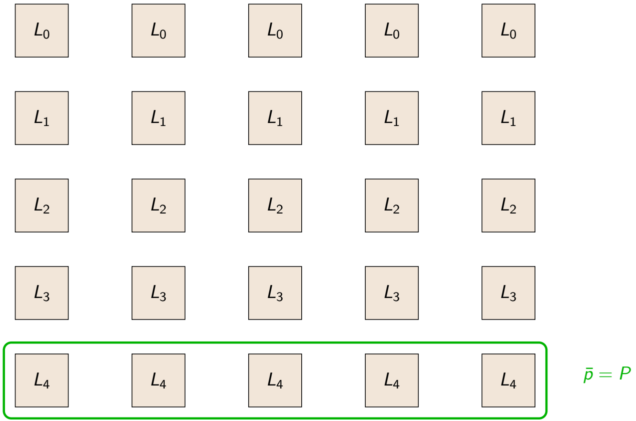
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

$\Rightarrow$ If the layers are processed from top to bottom, the processor usage is $\bar{p} < P$.

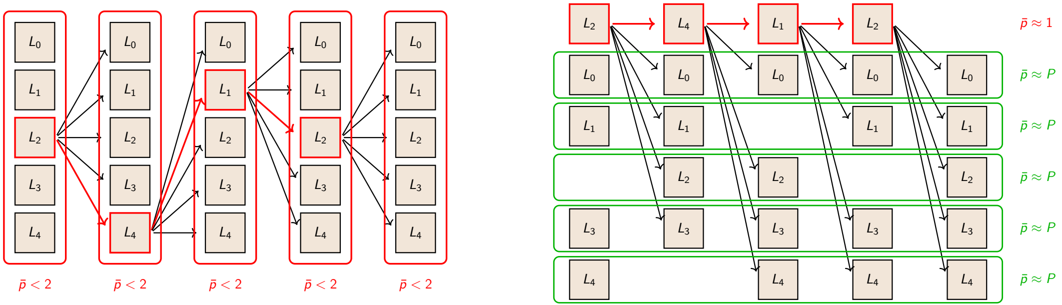
Proof of Lower Bound for Competitive Ratio



Idea: Our instance consists of a grid of chains L_i.

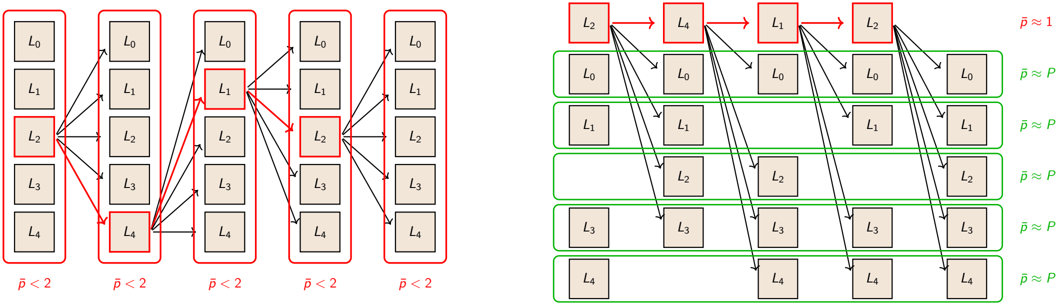
$\Rightarrow$ If the layers are processed from top to bottom, the processor usage is $\bar{p} < P$.

Proof of Lower Bound for Competitive Ratio



- Left schedule: Algorithm against any adversary. Next layer is released only after total completion of a layer ($\bar{p} < 2$)
- Right Schedule: A clairvoyant scheduler can process all the critical chains first, releasing the full graph, then processing it from top to bottom.

Proof of Lower Bound for Competitive Ratio



- A clairvoyant scheduler has an average processor usage above $\frac{P}{2}$, any algorithm could have one below 2
  $\Rightarrow$ No algorithm may be $\frac{P}{4}$-competitive
- Playing with the parameters, we get both $\frac{\log(n)}{4}$ and $\frac{\log(D)}{4}$ lower bounds
  $\Rightarrow$ CatBatch is asymptotically optimal (in competitive ratio) for both metrics!!

Outline

Near-Optimal in Competitive Ratio... Is it a good algorithm?

The following table gives my prediction of $\frac{T}{T_{opt}}$ if we tried CatBatch on real systems.

	Greedy algorithm (+backfilling)	CatBatch
99% of cases	probably < 1.5	probably > 2 or > 3
Worst case	$\frac{n}{3}$	$\log(n) + 3$

Is it good? **NO !** But ...

Perspectives

- Ana's perspective: We want to make intuitive and straightforward heuristics so that they may be implemented in real systems
- Yves's perspective: We want to give complicated answers to simple problems
  $\Rightarrow$ I grew up (research-wise) with Yves and Anne...

Perspectives

- Ana's perspective: We want to make intuitive and straightforward heuristics so that they may be implemented in real systems
- Yves's perspective: We want to give complicated answers to simple problems
  $\Rightarrow$ I grew up (research-wise) with Yves and Anne...

In this work:

- At long last, the *burning* question "what is the best competitive ratio achievable for online scheduling rigid task graphs?" has been answered
- Counter-intuitive algorithm with original constructions and new concepts
- Nice proofs (I may be biased)
- A new algorithm that behaves poorly in 99% of cases

$\Rightarrow$ I'm fulfilled! ☺

Current Work

- CatBatch is too extremely conservative to be viable as is...
- ... But category-based heuristics might be! CatBatch is a rough cornerstone for developing better heuristics
- They will hopefully be as efficient as greedy heuristics in most cases, and significantly better in some cases
- Happy to collaborate with Ana and others to test these in real HPC systems soon!

+ work on heuristics for more flexible model, where task length is estimated prior to execution, and not exactly known