Équilibrage et Réduction de charge, Ordonnancement 101

Guillaume Pallez
Inria

M2 CISD, Enseirb-Matmeca,
Automne 2020

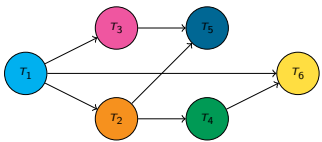
Plan du cours d'aujourd'hui

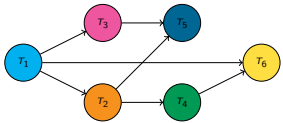
1. Définition d'un problème d'ordonnancement
   - Exemple
   - Règles du jeu
   - Critères
2. Analyse d'un problème d'ordonnancement
   - Exemple motivant
   - Décision vs Optimization
   - Taille du problème
   - Classes de complexité
   - Algorithmes d'approximation
   - Algorithmes gloutons
   - Graham's notation

Plan

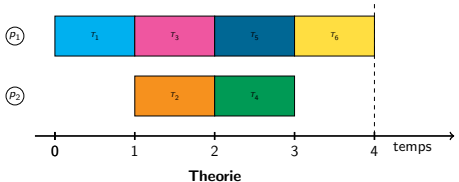
- p processeurs (ou nœuds ou cœurs ou unités de calcul).

- On représente une application par un DAG $\mathcal{G} = (V, E)$:
  - Les sommets sont des tâches (ou des fonctions à calculer)
  - Les arêtes représentent des dépendances en données (contraintes)

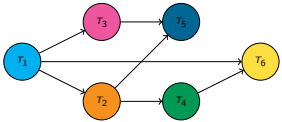




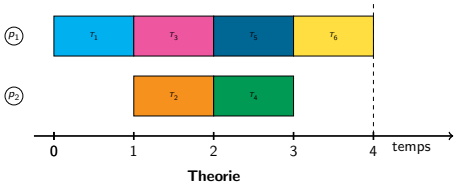
Tâches unitaires (quantité de travail), deux processeurs.

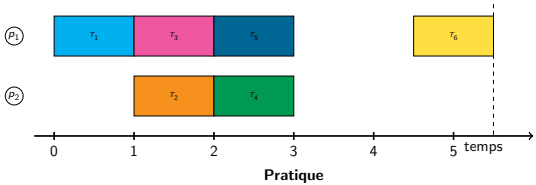


- En théorie, on a un ordonnancement optimal ☺



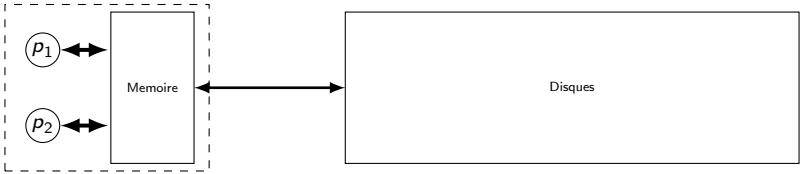
Tâches unitaires (quantité de travail), deux processeurs.





- En théorie, on a un ordonnancement optimal ☺
- En pratique, ce n'est pas ce qu'il se passe ☹

Pourquoi ?



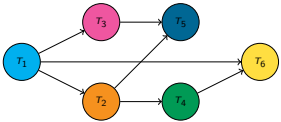
Modélisation (simplifiée) d'un ordinateur.

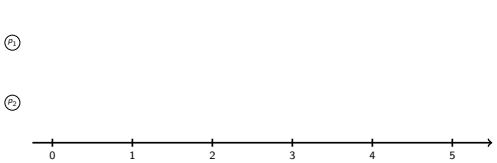
En général,

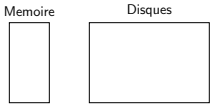
- **Mémoire** petite, accès rapides ;
- **Disques** larges, accès lents.

Un programme parallèle est en général une succession de rafales de **calculs** et de rafales de **communications**. La synchronisation créé en général du temps **d'inactivité**.
C'est le surcoût de parallélisation.

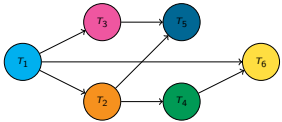
Retour à notre ordonnancement

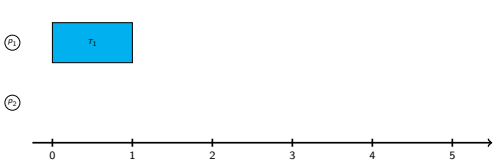


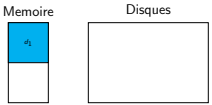




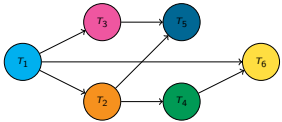
Retour à notre ordonnancement

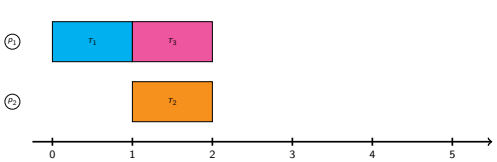


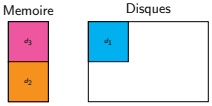




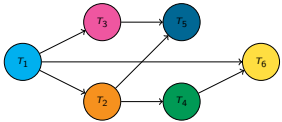
Retour à notre ordonnancement

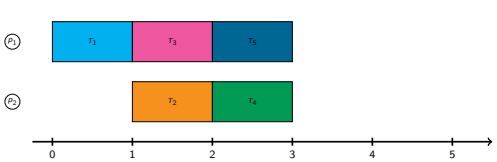


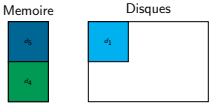




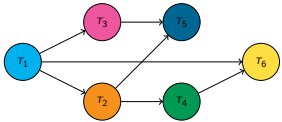
Retour à notre ordonnancement

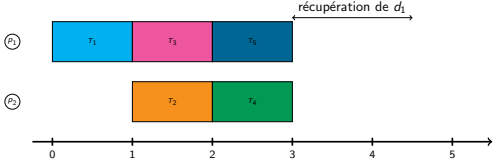


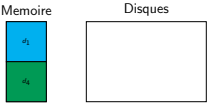




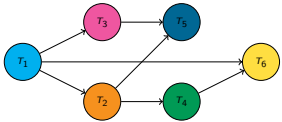
Retour à notre ordonnancement

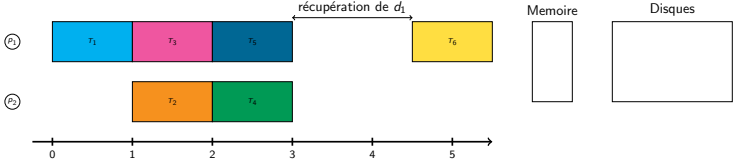




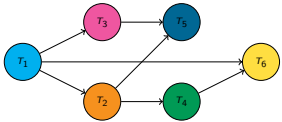


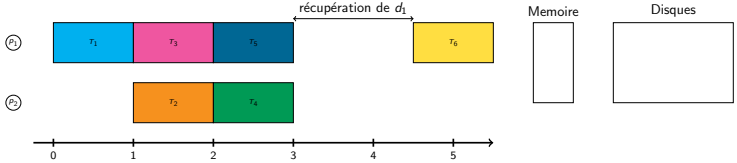
Retour à notre ordonnancement





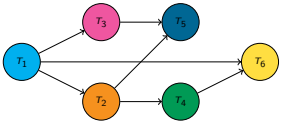
Retour à notre ordonnancement

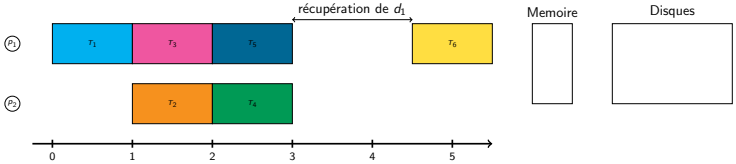




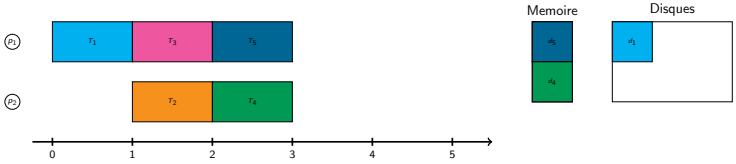
Est-ce qu'on peut faire mieux (ou prouver qu'on ne peut pas) ?

Retour à notre ordonnancement

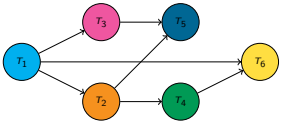


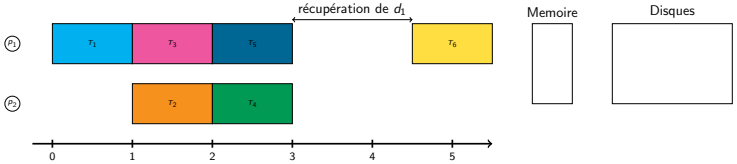


Est-ce qu'on peut faire mieux (ou prouver qu'on ne peut pas) ?

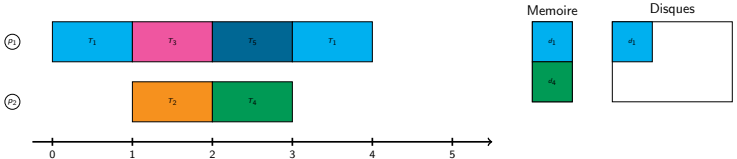


Retour à notre ordonnancement

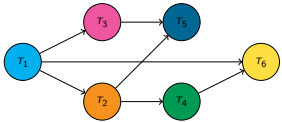


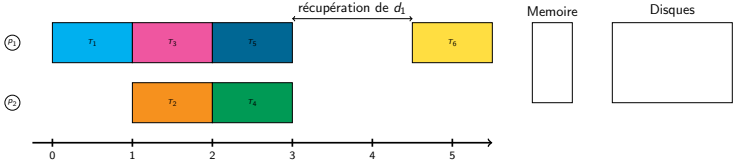


Est-ce qu'on peut faire mieux (ou prouver qu'on ne peut pas) ?

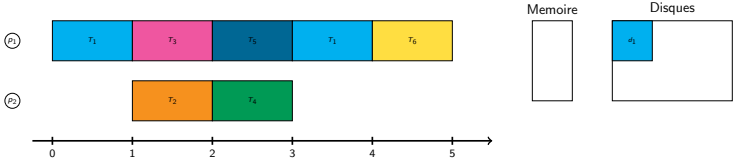


Retour à notre ordonnancement





Est-ce qu'on peut faire mieux (ou prouver qu'on ne peut pas) ?



Des questions à se poser

- Quelle application ?
- Quel modèle de machine ?
- Quel fonction objective ?

Plan

- L'ordonnancement en Informatique et en Recherche Opérationnelle
- De manière large : l'allocation temporelle **d'activités** sur des **ressources** pour atteindre un **objectif** voulu
- Quelques exemples :
  - L'allocation d'ouvrier·es sur des machines en usines pour (supposément) augmenter la productivité ;
  - L'allocation de salles pour des cours à l'université pour maximiser le nombre de salles vides le vendredi ;
  - L'allocation d'utilisateurs sur un téléscope où l'on paye à l'heure pour maximiser les profits ;
  - **L'allocation de calculs sur des processeurs et de communications sur un réseau pour minimiser le temps d'exécution de l'application.**

- L'ordonnancement en Informatique et en Recherche Opérationnelle
- De manière large : l'allocation temporelle **d'activités** sur des **ressources** pour atteindre un **objectif** voulu
- Quelques exemples :
  - L'allocation d'ouvrier·es sur des machines en usines pour (supposément) augmenter la productivité ;
  - L'allocation de salles pour des cours à l'université pour maximiser le nombre de salles vides le vendredi ;
  - L'allocation d'utilisateurs sur un téléscope où l'on paye à l'heure pour maximiser les profits ;
  - **L'allocation de calculs sur des processeurs et de communications sur un réseau pour minimiser le temps d'exécution de l'application.**

On définit un problème d'ordonnancement selon trois composants :

1. Une description des ressources
2. Une description des applications
3. Une description de l'objectif désiré (critère)

Utilisation (max) : pourcentage d'utilisation utile du CPU.

Makespan (min) : temps à la complétion du dernier job.

Temps de réponse (min) : différence entre le temps de complétion et le temps d'arrivée dans le système.

Stretch (min) : facteur de ralentissement pour un job par rapport à l'exécution dans un système vide.

Ces quantités sont centrées sur les taches ou les CPU. Elles peuvent être ensuite aggrégée en une unique fonction objective :

- max (i.e. le pire cas)
- moyenne arithmétique (i.e. la somme)
- la variance (équité entre les taches)
- etc.

On définit pour une tâche T_i :

- Son temps d'exécution p_i
- Sa date d'apparition r_i
- Son temps de complétion C_i
- (le nombre de processors requis q_i)
- (la deadline d_i)

Temps de complétion (makespan)

$$C_{\max} = \max_i C_i$$

- Une métrique qui a plutôt du sens quand on s'intéresse à une seule application.

On définit pour une tâche T_i :

- Son temps d'exécution p_i
- Sa date d'apparition r_i
- Son temps de complétion C_i
- (le nombre de processors requis q_i)
- (la deadline d_i)

Temps de réponse (response time)

$$F_i = C_i - r_i$$

- Le temps de Flow maximum : $F_{\max} = \max_i F_i$

On définit pour une tâche T_i :

- Son temps d'exécution p_i
- Sa date d'apparition r_i
- Son temps de complétion C_i
- (le nombre de processors requis q_i)
- (la deadline d_i)

Temps d'attente moyen (waiting time)

$$W_i = C_i - r_i - p_i$$

- Le temps de réponse moyen : $SW = \sum_i W_i$

On définit pour une tâche T_i :

- Son temps d'exécution p_i
- Sa date d'apparition r_i
- Son temps de complétion C_i
- (le nombre de processors requis q_i)
- (la deadline d_i)

Ralentissement (slowdown)

$$S_i = \frac{C_i - r_i}{p_i}$$

- Maximum stretch : $S_{\max} = \max_i S_i$

Commencons par un problème simple : $\mathrm{INDEP}(2)$

1. Deux machines identiques, P_1 et P_2
   - Chaque machine ne peut executer qu'une tache à la fois
2. Application : n taches de calcul
   - La tache i peut s'executer sur P_1 ou P_2 en a_i secondes
   - Les taches sont *independentes* : elles peuvent s'exécuter dans n'importe quel ordre
3. Objectif : minimiser $\max(M_1, M_2)$ (*Makespan*)
   - M_i est le temps où la machine P_i finit ses calculs

Cas simple : tous les jobs sont identiques

n jobs de taille a :
$T_1, T_2, \ldots, T_n$

Deux machines :
P_1, P_2

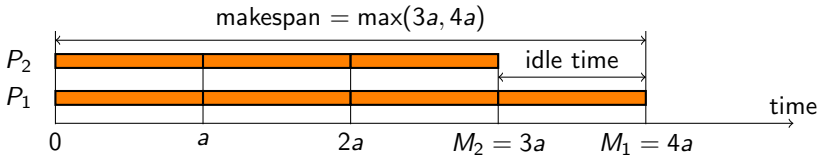
Objectif :
minimiser $\max(M_1, M_2)$

n jobs de taille a :
$T_1, T_2, \ldots, T_n$

Deux machines :
P_1, P_2

Objectif :
minimiser $\max(M_1, M_2)$

- Solution simple : allouer $\lceil n/2 \rceil$ jobs à P_1, et $\lfloor n/2 \rfloor$ jobs à P_2 ;
  $\rightarrow$ Idée : s'arranger pour que P_1 et P_2 finissent en même temps.



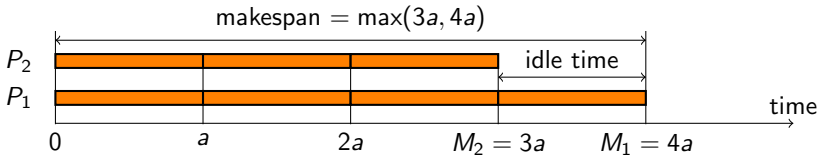
Gantt chart for INDEP(2) with 7 identical tasks

n jobs de taille a :
$T_1, T_2, \ldots, T_n$

Deux machines :
P_1, P_2

Objectif :
minimiser $\max(M_1, M_2)$

- Solution simple : allouer $\lceil n/2 \rceil$ jobs à P_1, et $\lfloor n/2 \rfloor$ jobs à P_2 ;
  $\rightarrow$ Idée : s'arranger pour que P_1 et P_2 finissent en même temps.
- La taille du problème est $O(n)$ (toutes les tâches sont identifiées)
- L'algorithme a une complexité $O(n)$. On dit qu'on a un algorithme en temps polynomial (dans ce cas, linéaire) :
  $\rightarrow$ Pour chaque tâche, on compare son indice à $n/2$, si c'est inférieur on met sur P_1, si c'est supérieur, sur P_2.
- Le problème est considéré comme *résolu*.



Gantt chart for INDEP(2) with 7 identical tasks

- Task T_i (for $i = 1, \ldots, n$) takes time $a_i \geq 0$
- Problem size : $Size = O(n + \sum_{i=1}^{n} \log a_i)$
- We say a problem is "easy" when we have a polynomial-time (p-time) algorithm :
  - Number of elementary operations is $O(f(Size))$, where f is a polynomial and $Size$ is the problem size
- $\mathcal{P}$ is the set of problems that can be solved with a p-time algorithm
- Question : is there a p-time algorithm to solve $\mathrm{INDEP}(2)$?
- Disclaimer : Some of you may be familiar with algorithms and computational complexity, so bear with me while I review some fundamental background

- Complexity theory is for *decision problems*, i.e., problems that have a yes/no answer
- Scheduling problems are optimization problems

Decision and Optimization

$Dec(M)$: Is there a schedule σ such that $Obj(\sigma) \leq M$?
Opt : Find $M^\star$ such that $M^\star = \min_\sigma Obj(\sigma)$.

- Complexity theory is for *decision problems*, i.e., problems that have a yes/no answer
- Scheduling problems are optimization problems

Decision and Optimization

$Dec(M)$: Is there a schedule σ such that $Obj(\sigma) \leq M$?
Opt : Find $M^\star$ such that $M^\star = \min_\sigma Obj(\sigma)$.

- Decision version of $\mathrm{INDEP}(2)$: for an integer bound M, is there a schedule whose MS is lower than M ?

- Complexity theory is for *decision problems*, i.e., problems that have a yes/no answer
- Scheduling problems are optimization problems

Decision and Optimization

$Dec(M)$: Is there a schedule σ such that $Obj(\sigma) \leq M$?
Opt : Find $M^\star$ such that $M^\star = \min_\sigma Obj(\sigma)$.

- Decision version of $\mathrm{INDEP}(2)$: for an integer bound M, is there a schedule whose MS is lower than M ?

If we have a p-time algo for Opt, then we have p-time algo for $Dec(M)$:

- Run the optimization algorithm, and check whether the makespan is lower than M

$\texttt{Complexity}(Dec(M)) = \texttt{Complexity}(Opt)$

- Complexity theory is for *decision problems*, i.e., problems that have a yes/no answer
- Scheduling problems are optimization problems

Decision and Optimization

$Dec(M)$: Is there a schedule σ such that $Obj(\sigma) \leq M$?
Opt : Find $M^\star$ such that $M^\star = \min_\sigma Obj(\sigma)$.

- Decision version of $\mathrm{INDEP}(2)$: for an integer bound M, is there a schedule whose MS is lower than M ?

If we have a p-time algo for Opt, then we have p-time algo for $Dec(M)$:

- Run the optimization algorithm, and check whether the makespan is lower than M

$\texttt{Complexity}(Dec(M)) = \texttt{Complexity}(Opt)$

If we have a p-time algo for $Dec(M)$, then we often have p-time algo for Opt :

- Binary search for the lowest M ($M \leq n \times \max_i a_i$)

$\texttt{Comp}(Opt) = \log(n \max a_i) \cdot \texttt{Comp}(Dec(n \max a_i))$

Plan

- One has to be careful when defining the problem size

- For $\mathrm{INDEP}(2)$:
  - We need to enumerate n integers (the a_i's), so the size is at least polynomial in n
  - Each a_i must be encoded (in binary) in $\lceil \log(a_i) \rceil$ bits
  - The data is $O(f(n) + \sum_{i=1}^{n} \lceil \log(a_i) \rceil)$, where f is a polynomial

- A problem is in $\mathcal{P}$ only if an algorithm exist that is polynomial in the data size as defined above

- It is often possible to find algorithms polynomial in a quantity that is exponential in the (real) problem size
- For instance, to solve $\mathrm{INDEP}(2)$, one can resort to dynamic programming to obtain an algorithm with complexity $O(n \times \sum_{i=1}^{n} a_i)$ EXERCISE
- This is a polynomial algorithm if the a_i's are encoded in unary, i.e., polynomial in the numerical values of the a_i's
- But with the a_i's encoded in binary, $\sum_{i=1}^{n} a_i$ is exponential in the problem size!
  - To a log, linear is exponential ☺

We say that this algorithm is *pseudopolynomial*

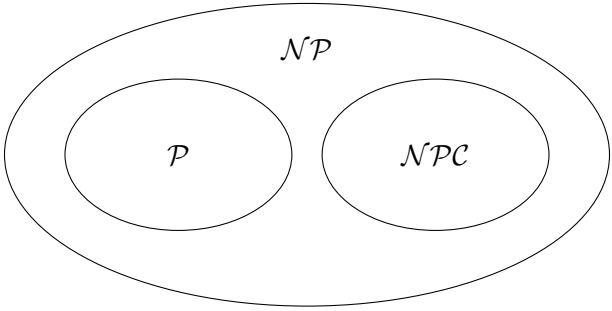
- Some problems in $\mathcal{NP}$ are at least as difficult as all other problems in $\mathcal{NP}$
- They are called $\mathcal{NP}$-complete, and their set is $\mathcal{NPC}$
- Cook's theorem : The SAT problem is in $\mathcal{NPC}$
  - Satisfiability of a boolean conjunction of disjunctions

- Some problems in $\mathcal{NP}$ are at least as difficult as all other problems in $\mathcal{NP}$
- They are called $\mathcal{NP}$-complete, and their set is $\mathcal{NPC}$
- Cook's theorem : The SAT problem is in $\mathcal{NPC}$
  - Satisfiability of a boolean conjunction of disjunctions

- How to prove that a problem, P, is $\mathcal{NP}$-complete :
  - Prove that $P \in \mathcal{NP}$ (typically easy)
  - Prove that P *reduces* to Q, where $Q \in \mathcal{NPC}$ (can be hard)
    - For an instance I_Q, construct in p-time an instance I_P
    - Prove that I_P has a solution if and only if I_Q has a solution

- By now, we know many problems in $\mathcal{NPC}$
- Goal : pick $Q \in \mathcal{NPC}$ so that the reduction is easy

Well-known complexity classes



INDEP(2) is $\mathcal{NP}$-complete

- INDEP(2) (decision version) is in $\mathcal{NP}$
  - Certificate : for each a_i whether it is scheduled on P_1 or P_2
  - In linear time, compute the makespan on both processors, and compare to k to answer "Yes"
- Let us consider an instance of 2-PARTITION $\in \mathcal{NPC}$:
  - Given n integers x_i, is there a subset I of $\{1, \ldots, n\}$ such that $\sum_{i \in I} x_i = \sum_{i \notin I} x_i$?
- Let us construct an instance of INDEP(2) :
  - Let $k = \frac{1}{2} \sum x_i$, let $a_i = x_i$
- The proof is trivial
  - If k is non-integer, neither instance has a solution
  - Otherwise, each processor corresponds to one subset
- In fact, INDEP(2) is essentially identical to 2-PARTITION

So what ?

- This $\mathcal{NP}$-completeness proof is probably the most trivial in the world ☺
- But now we are thus pretty sure that there is no p-time algorithm to solve $\mathrm{INDEP}(2)$

- What we look for now are *approximation algorithms*...

Plan

- Consider an optimization problem
- A p-time algorithm is a λ-*approximation algorithm* if it returns a solution that is at most a factor λ from the optimal solution (the closer λ to 1, the better)
  - λ is called the *approximation ratio*

- Consider an optimization problem
- A p-time algorithm is a λ-*approximation algorithm* if it returns a solution that is at most a factor λ from the optimal solution (the closer λ to 1, the better)
  - λ is called the *approximation ratio*
- *Polynomial Time Approximation Scheme* (PTAS) : for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm (may be non-polynomial in $1/\epsilon$)

- Consider an optimization problem
- A p-time algorithm is a λ-*approximation algorithm* if it returns a solution that is at most a factor λ from the optimal solution (the closer λ to 1, the better)
  - λ is called the *approximation ratio*
- *Polynomial Time Approximation Scheme* (PTAS) : for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm (may be non-polynomial in $1/\epsilon$)
- *Fully Polynomial Time Approximation Scheme* (FPTAS) : for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm polynomial in $1/\epsilon$

- Consider an optimization problem
- A p-time algorithm is a λ-*approximation algorithm* if it returns a solution that is at most a factor λ from the optimal solution (the closer λ to 1, the better)
  - λ is called the *approximation ratio*
- *Polynomial Time Approximation Scheme* (PTAS) : for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm (may be non-polynomial in $1/\epsilon$)
- *Fully Polynomial Time Approximation Scheme* (FPTAS) : for any ϵ, there exists a $(1+\epsilon)$-approximation algorithm polynomial in $1/\epsilon$
- Typical goal : find a FPTAS, if not find a PTAS, if not find a λ-approximation for a low value of λ, if not find *efficient* heuristics

Plan

A **greedy algorithm** is one that builds a solution step-by-step, via local incremental decisions

- It turns out that several greedy scheduling algorithms are approximation algorithms
  - Informally, they are not as "bad" as one may think

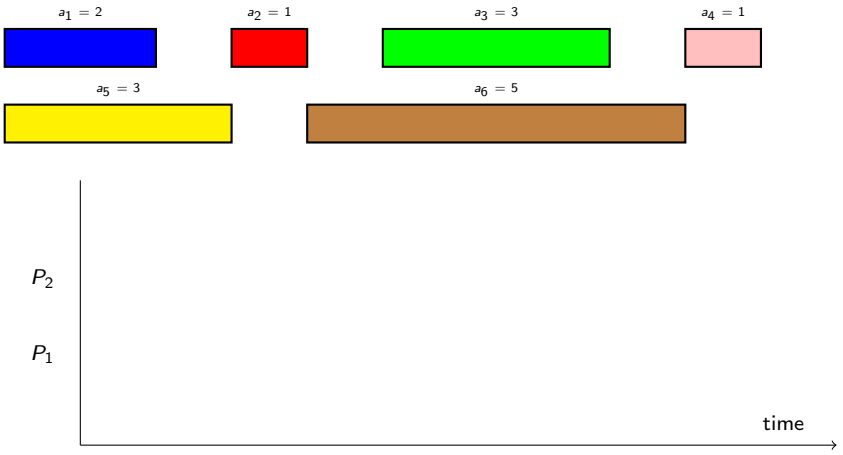
A **greedy algorithm** is one that builds a solution step-by-step, via local incremental decisions

- It turns out that several greedy scheduling algorithms are approximation algorithms
  - Informally, they are not as "bad" as one may think

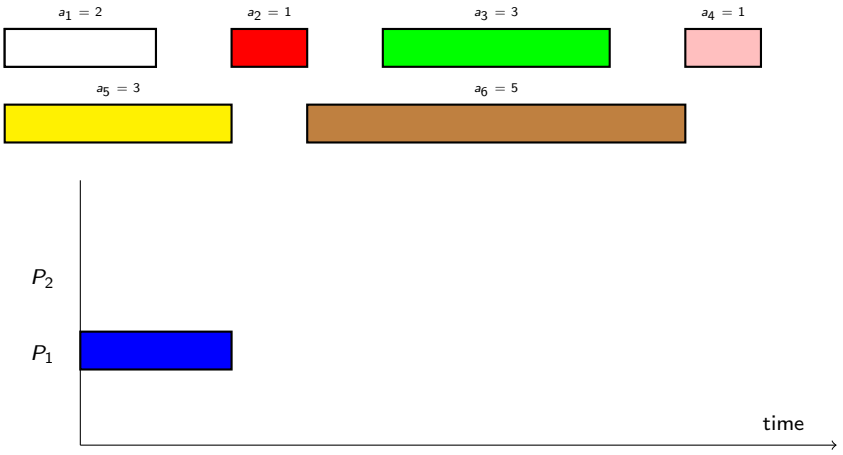
Two natural greedy algorithms for $\mathrm{INDEP}(2)$:

- **greedy-online** : take the tasks in arbitrary order and assign each task to the least loaded processor
  - We don't know which tasks are coming
- **greedy-offline** : sort the tasks by decreasing a_i, and assign each task in that order to the least loaded processor
  - We know all the tasks ahead of time

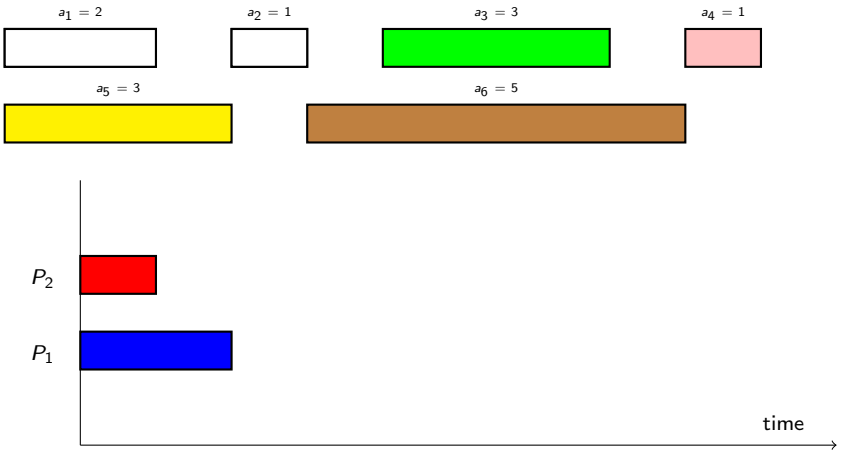
Example with 6 tasks: Online



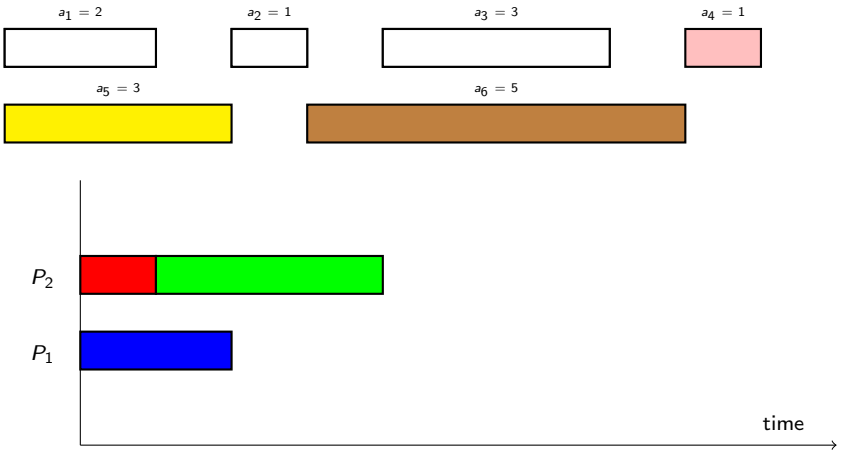
Example with 6 tasks: Online



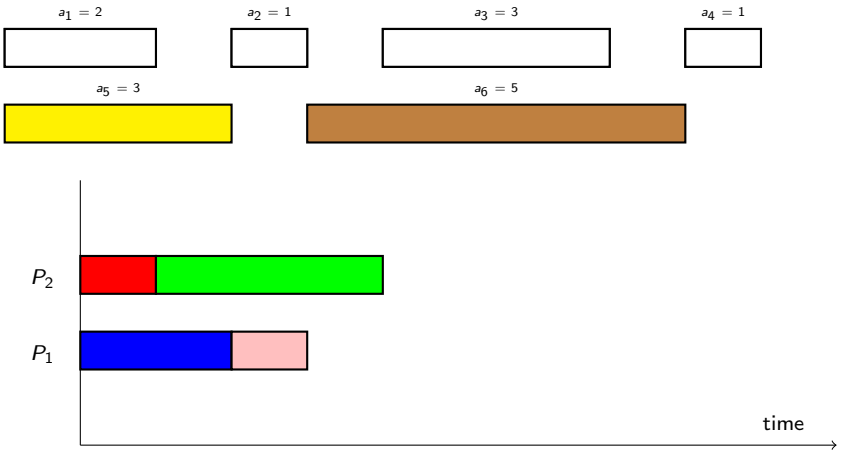
Example with 6 tasks: Online



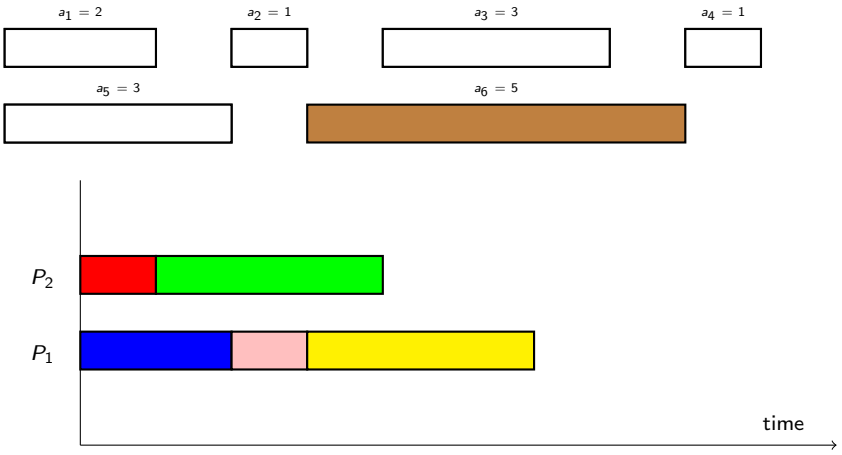
Example with 6 tasks: Online



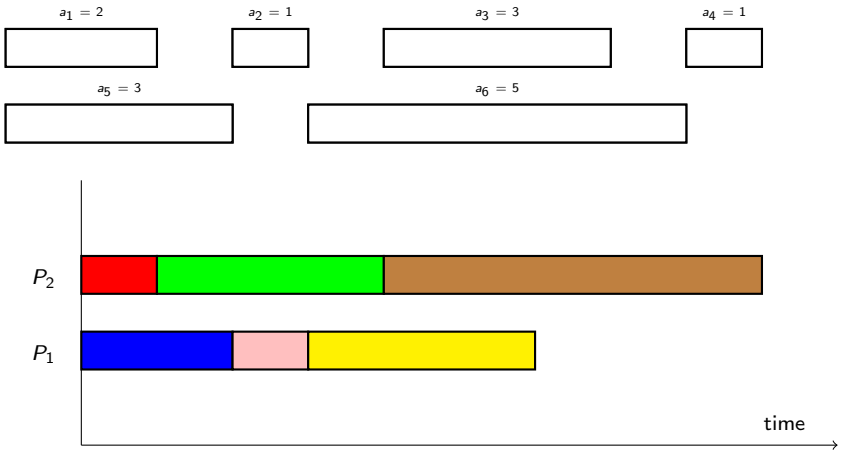
Example with 6 tasks: Online



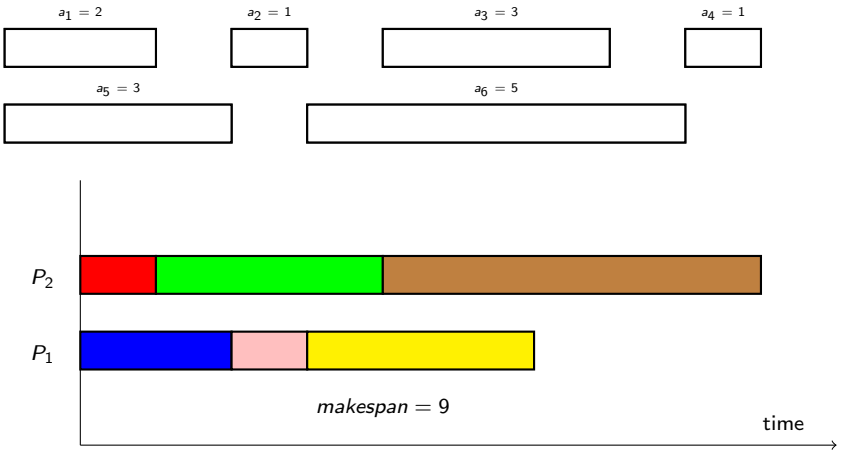
Example with 6 tasks: Online



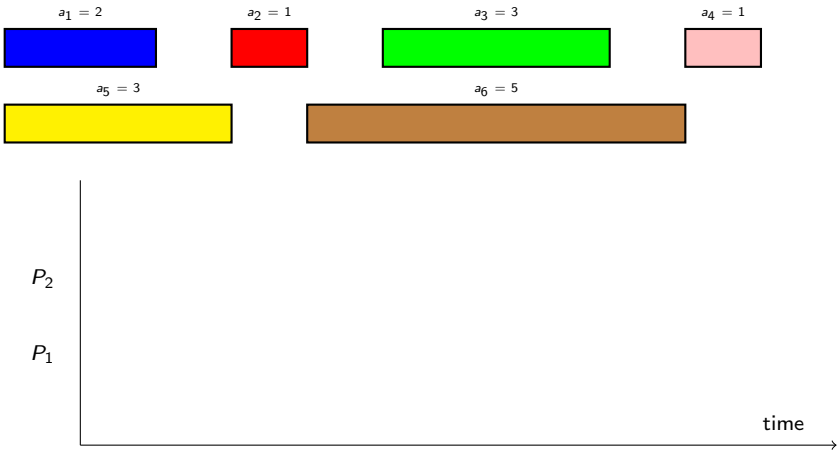
Example with 6 tasks: Online



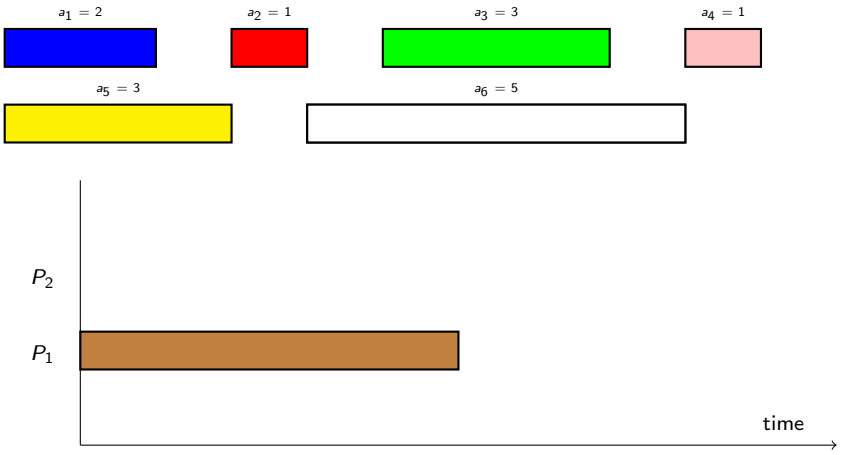
Example with 6 tasks: Online



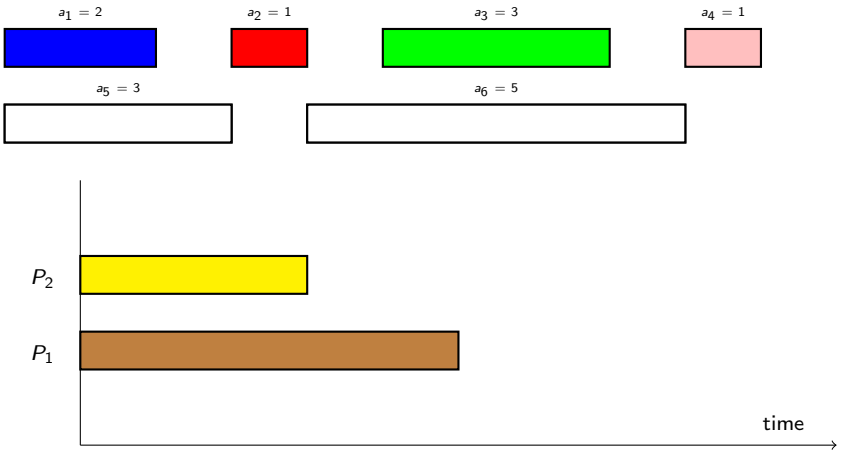
Example with 6 tasks: Offline



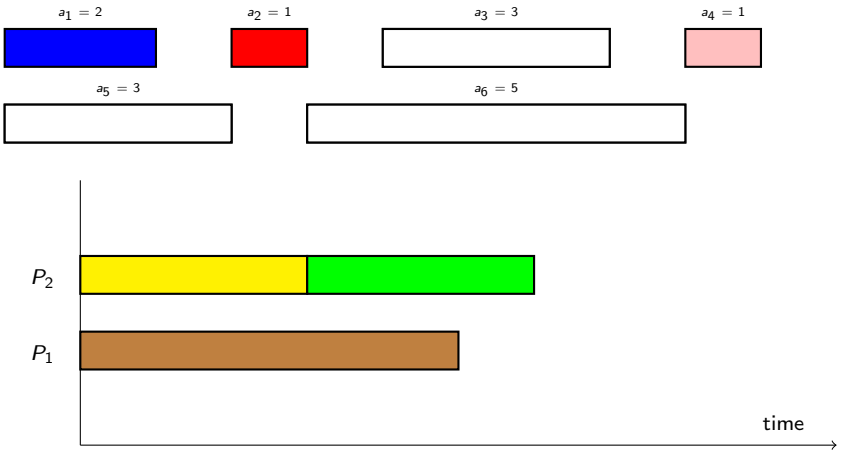
Example with 6 tasks: Offline



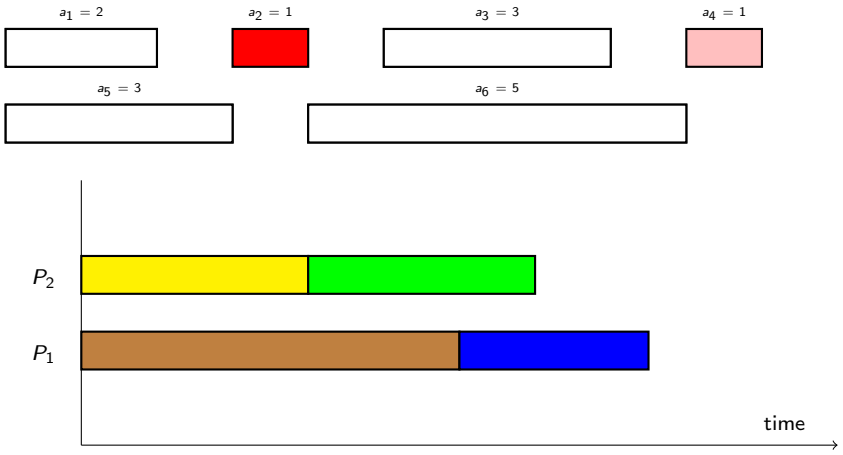
Example with 6 tasks: Offline



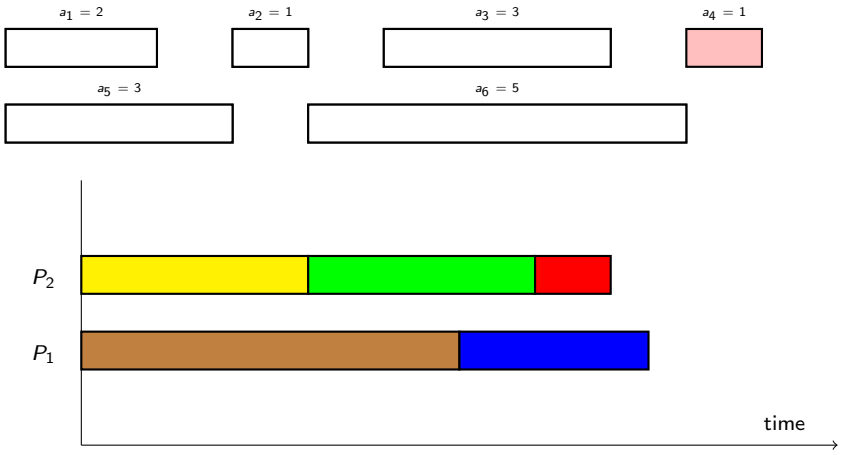
Example with 6 tasks: Offline



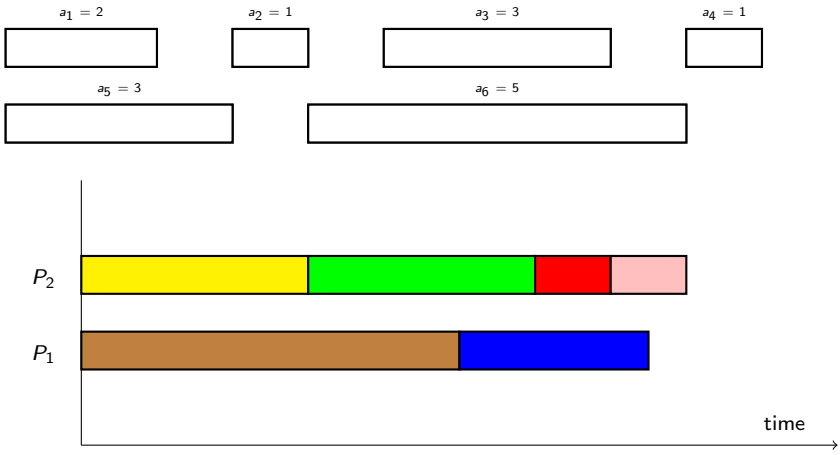
Example with 6 tasks: Offline



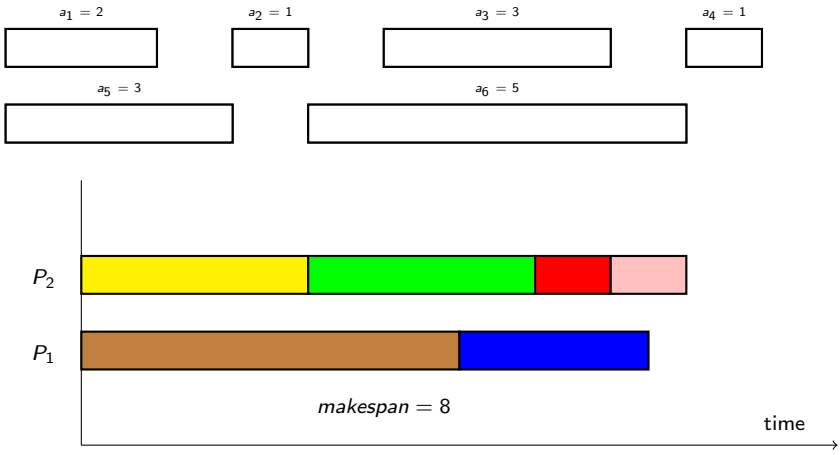
Example with 6 tasks: Offline



Example with 6 tasks: Offline



Example with 6 tasks: Offline



Theorem

Greedy-online is a $\frac{3}{2}$-approximation for INDEP(2)

Theorem

Greedy-offline is a $\frac{7}{6}$-approximation for INDEP(2)

Theorem

Greedy-online is a $\frac{3}{2}$-approximation for INDEP(2)

- Proof :
  - P_i finishes computing at time M_i (M stands for makespan)
  - Let us assume $M_1 \geq M_2$ ($M_{greedy} = M_1$)
  - Let T_j the last task to execute on P_1
  - Since the greedy algorithm put T_j on P_1, then $M_1 - a_j \leq M_2$

Theorem

Greedy-offline is a $\frac{7}{6}$-approximation for INDEP(2)

Theorem

Greedy-online is a $\frac{3}{2}$-approximation for INDEP(2)

- Proof :
  - P_i finishes computing at time M_i (M stands for makespan)
  - Let us assume $M_1 \geq M_2$ ($M_{greedy} = M_1$)
  - Let T_j the last task to execute on P_1
  - Since the greedy algorithm put T_j on P_1, then $M_1 - a_j \leq M_2$
  - We have $M_1 + M_2 = \sum_i a_i = S$
  - $M_{greedy} = M_1 = \frac{1}{2}(M_1 + (M_1 - a_j) + a_j) \leq \frac{1}{2}(M_1 + M_2 + a_j) = \frac{1}{2}(S + a_j)$

Theorem

Greedy-offline is a $\frac{7}{6}$-approximation for INDEP(2)

Theorem

Greedy-online is a $\frac{3}{2}$-approximation for INDEP(2)

- Proof :
  - P_i finishes computing at time M_i (M stands for makespan)
  - Let us assume $M_1 \geq M_2$ ($M_{greedy} = M_1$)
  - Let T_j the last task to execute on P_1
  - Since the greedy algorithm put T_j on P_1, then $M_1 - a_j \leq M_2$
  - We have $M_1 + M_2 = \sum_i a_i = S$
  - $M_{greedy} = M_1 = \frac{1}{2}(M_1 + (M_1 - a_j) + a_j) \leq \frac{1}{2}(M_1 + M_2 + a_j) = \frac{1}{2}(S + a_j)$
  - but $M_{opt} \geq S/2$ (ideal lower bound on optimal)
  - and $M_{opt} \geq a_j$ (at least one task is executed)
  - Therefore : $M_{greedy} \leq \frac{1}{2}(2M_{opt} + M_{opt}) = \frac{3}{2}M_{opt}$ □

Theorem

Greedy-offline is a $\frac{7}{6}$-approximation for INDEP(2)

Theorem

Greedy-online is a $\frac{3}{2}$-approximation for INDEP(2)

- Proof :
  - P_i finishes computing at time M_i (M stands for makespan)
  - Let us assume $M_1 \geq M_2$ ($M_{greedy} = M_1$)
  - Let T_j the last task to execute on P_1
  - Since the greedy algorithm put T_j on P_1, then $M_1 - a_j \leq M_2$
  - We have $M_1 + M_2 = \sum_i a_i = S$
  - $M_{greedy} = M_1 = \frac{1}{2}(M_1 + (M_1 - a_j) + a_j) \leq \frac{1}{2}(M_1 + M_2 + a_j) = \frac{1}{2}(S + a_j)$
  - but $M_{opt} \geq S/2$ (ideal lower bound on optimal)
  - and $M_{opt} \geq a_j$ (at least one task is executed)
  - Therefore : $M_{greedy} \leq \frac{1}{2}(2M_{opt} + M_{opt}) = \frac{3}{2}M_{opt}$ □

.. and the bound is tight :

- a_i's $= \{1,1,2\}$
- $M_{greedy} = 3$; $M_{opt} = 2$
- $ratio = \frac{3}{2}$

Theorem

Greedy-offline is a $\frac{7}{6}$-approximation for INDEP(2)

PTAS and FPTAS for INDEP(2)

Theorem

There is a PTAS ($(1+\epsilon)$-approximation) for $\mathrm{INDEP}(2)$

- Proof sketch :
  - Classify tasks as either "small" or "large"
    - Very common technique
  - Replace all small tasks by same-size tasks
  - Compute an optimal schedule of the modified problem in p-time (not polynomial in $1/\epsilon$)
  - Show that the cost is $\leq 1+\epsilon$ away from the optimal cost
  - The proof is a couple of pages, but not terribly difficult

Theorem

There is an FPTAS ($(1+\epsilon)$-approx pol. in $1/\epsilon$) for $\mathrm{INDEP}(2)$

We know a lot about $\mathrm{INDEP}(2)$

- $\mathrm{INDEP}(2)$ is NP-complete
- We have simple greedy algorithms with guarantees on result quality
- We have a simple PTAS
- We even have a (less simple) FPTAS
- $\mathrm{INDEP}(2)$ is basically "solved"

- Sadly, not many scheduling problems are this well understood...

Plan

Graham Notation

Many parameter can change in a scheduling problem. Graham has then proposed the following classification : $\langle \alpha | \beta | \gamma \rangle$ [Brucker-Book]

Graham Notation

Many parameter can change in a scheduling problem. Graham has then proposed the following classification : $\langle \alpha | \beta | \gamma \rangle$ [Brucker-Book]

- α is the processor environment (a few examples):
  - $\emptyset$: single processor;
  - P: identical processors;
  - Q: uniform processors;
  - R: unrelated processors;

Graham Notation

Many parameter can change in a scheduling problem. Graham has then proposed the following classification : $\langle\alpha|\beta|\gamma\rangle$ [Brucker-Book]

- α is the processor environment (a few examples):
  - $\emptyset$: single processor;
  - P: identical processors;
  - Q: uniform processors;
  - R: unrelated processors;
- β describe task and resource characteristics (a few examples):
  - $pmtn$: preemption;
  - $prec$, $tree$ or $chains$: general precedence constraints, tree constraints, set of chain constraints and independent tasks otherwise;
  - r_j: tasks have release dates;
  - $p_j = p$ or $\underline{p} \leqslant p_j \leqslant \overline{p}$: all task have processing time equal to p, or comprised between $\underline{p}$ and $\overline{p}$, or have arbitrary processing times otherwise;
  - $\tilde{d}$: deadlines;

Graham Notation

Many parameter can change in a scheduling problem. Graham has then proposed the following classification : $\langle\alpha|\beta|\gamma\rangle$ [Brucker-Book]

- α is the processor environment (a few examples):
  - $\emptyset$: single processor;
  - P: identical processors;
  - Q: uniform processors;
  - R: unrelated processors;
- β describe task and resource characteristics (a few examples):
  - $pmtn$: preemption;
  - $prec$, $tree$ or $chains$: general precedence constraints, tree constraints, set of chain constraints and independent tasks otherwise;
  - r_j: tasks have release dates;
  - $p_j = p$ or $\underline{p} \leqslant p_j \leqslant \overline{p}$: all task have processing time equal to p, or comprised between $\underline{p}$ and $\overline{p}$, or have arbitrary processing times otherwise;
  - $\tilde{d}$: deadlines;
- γ denotes the optimization criterion (a few examples):
  - $C_{\max}$: makespan;
  - $\sum C_i$: average completion time;
  - $\sum w_i C_i$: weighted A.C.T;
  - $L_{\max}$: maximum lateness $(\max C_i - d_i)$;
  - ...

Graham Notation: Exercise

Understand the following problems and propose a practical situation to illustrate them:

- $\langle P|prec|C_{\max}\rangle$
- $\langle P|q_j, prec|C_{\max}\rangle$
- $\langle P|q_j|F_{\max}\rangle$
- $\langle 1|r_j; pmtn|S_{\max}\rangle$
- $\langle 1|r_j; pmtn, d_i|L_{\max}\rangle$

Inspiration

Les slides de ce cours ont été bien inspirés par ceux d'Arnaud Legrand (comme d'hab), mais aussi de cours d'Henri Casanova/Yves Robert/Anne Benoit/Frédéric Vivien/Loris Marchal.