

Résoudre des problèmes NP-dur ?

Enumeration implicite et algorithmes d'approximation

ACT – Algorithmes et Complexité – M1 Info

Bilel Derbel
Département Informatique - Univ. Lille
(D'après le cours de Sophie Tison)

Problèmes NP-Durs

- Parler de problèmes NP, si ce ne sont pas des problèmes de décisions, n'a gère de sens !
 - Par contre on peut parler de problèmes NP-Durs

Un problème d'optimisation est NP-dur si l'existence d'un algorithme polynomial pour ce problème implique $P = NP$

- **Problèmes d'optimisation: on cherche une solution qui optimise (minimise ou maximise) une fonction objectif**
- Exemple du TP:
 - Le problème d'optimisation : comment ordonner les tâches pour minimiser l'attente?
 - Le problème de décision : peut-on avoir un ordonnancement telle que l'attente maximale soit inférieure à une certaine valeur
 - si il y a un algorithme polynomial pour le problème d'optimisation, il y en a un aussi pour celui de décision
 - Comme le problème de décision est NP-dur, si il y a un algorithme polynomial pour le problème d'optimisation, alors $P = NP$
 - Donc le problème d'optimisation est NP-dur

Problème NP-dur: que faire ?

- Il paraît raisonnable d'abandonner l'idée d'avoir un algorithme polynomial qui donne à coup sûr la solution optimale, à moins de prétendre prouver que $P = NP$ (très peu probable)!
 - **Abandonner l'idée d'avoir un algorithme polynomial, e.g.,**
 - Algorithmes pseudo-polynomiaux
 - Enumeration ‘intelligente’
 - ...
 - **Abandonner l'idée d'avoir un algorithme qui donne toujours la solution optimale !**
 - Algorithme d'approximation (avec garantie)
 - (Meta-)Heuristique

Algorithmes pseudo-polynomiaux

- Un algorithme **pseudo-polynomial** est un algorithme qui est polynomial si les entiers de la donnée (les valeurs numériques) sont codés en base 1
 - autrement dit, qui est polynomiale par rapport à la plus grande valeur numérique des données du problème en entrée, et non de la taille/longueur des données
- Un problème est faiblement NP-dur si il est NP-dur et qu'il existe un algorithme pseudo-polynomial pour le résoudre

Algorithmes pseudo-polynomiaux

- **Exemple: problème du sac à dos**

- Donnée :
 - **n** objets $\{1, \dots, i, \dots, n\}$, chacun de **poids p_i** et de **valeur v_i**
 - un sac de **capacité C**
- Problème d'optimisation: remplir le sac en maximisant sa valeur totale (on ne peut pas fractionner les objets)

- **Algorithme de programmation dynamique**

- Soit **OPT(i,X)** : la valeur optimale pour un sac de capacité X et en utilisant seulement les objets $\{1, \dots, i\}$
- **On cherche à calculer OPT(n,C)** :
 - Pour tout $X \leq C$ et $i \leq n$; $OPT(0,X) = OPT(i,0) = 0$
 - Si $p_i > X$ alors $OPT(i,X) = OPT(i-1,X)$
 - Si $p_i \leq X$ alors $OPT(i,X) = \max \{OPT(i-1,X), OPT(i-1,X-p_i) + v_i\}$
 - Complexité en $O(n*C)$

Le sac ne peut pas contenir l'objet

Si le sac peut contenir l'objet
Alors soit on ne le met, soit non !

Algorithmes pseudo-polynomiaux

- **Exemple: problème du sac à dos**

- Donnée :
 - **n** objets $\{1, \dots, i, \dots, n\}$, chacun de **poids p_i** et de **valeur v_i**
 - un sac de **capacité C**
- Problème d'optimisation: remplir le sac en maximisant sa valeur totale (on ne peut pas fractionner les objets)

Les pseudo-polynomiaux ne sont pas la panacée: ils restent coûteux et pour beaucoup de problèmes NP-durs, on n'en connaît pas !

- **Algorithme de programmation dynamique**

- Soit **OPT(i,X)** : la valeur optimale pour un sac de capacité X et en utilisant seulement les objets $\{1, \dots, i\}$
- **On cherche à calculer OPT(n,C)** :
 - Pour tout $X \leq C$ et $i \leq n$; $OPT(0,X) = OPT(i,0) = 0$
 - Si $p_i > X$ alors $OPT(i,X) = OPT(i-1,X)$
 - Si $p_i \leq X$ alors $OPT(i,X) = \max \{OPT(i-1,X), OPT(i-1,X-p_i) + v_i\}$
 - Complexité en $O(n*C)$

Le sac ne peut pas contenir l'objet



Si le sac peut contenir l'objet
Alors soit on ne le met, soit non !



Algorithmes par énumération ‘intelligente’...

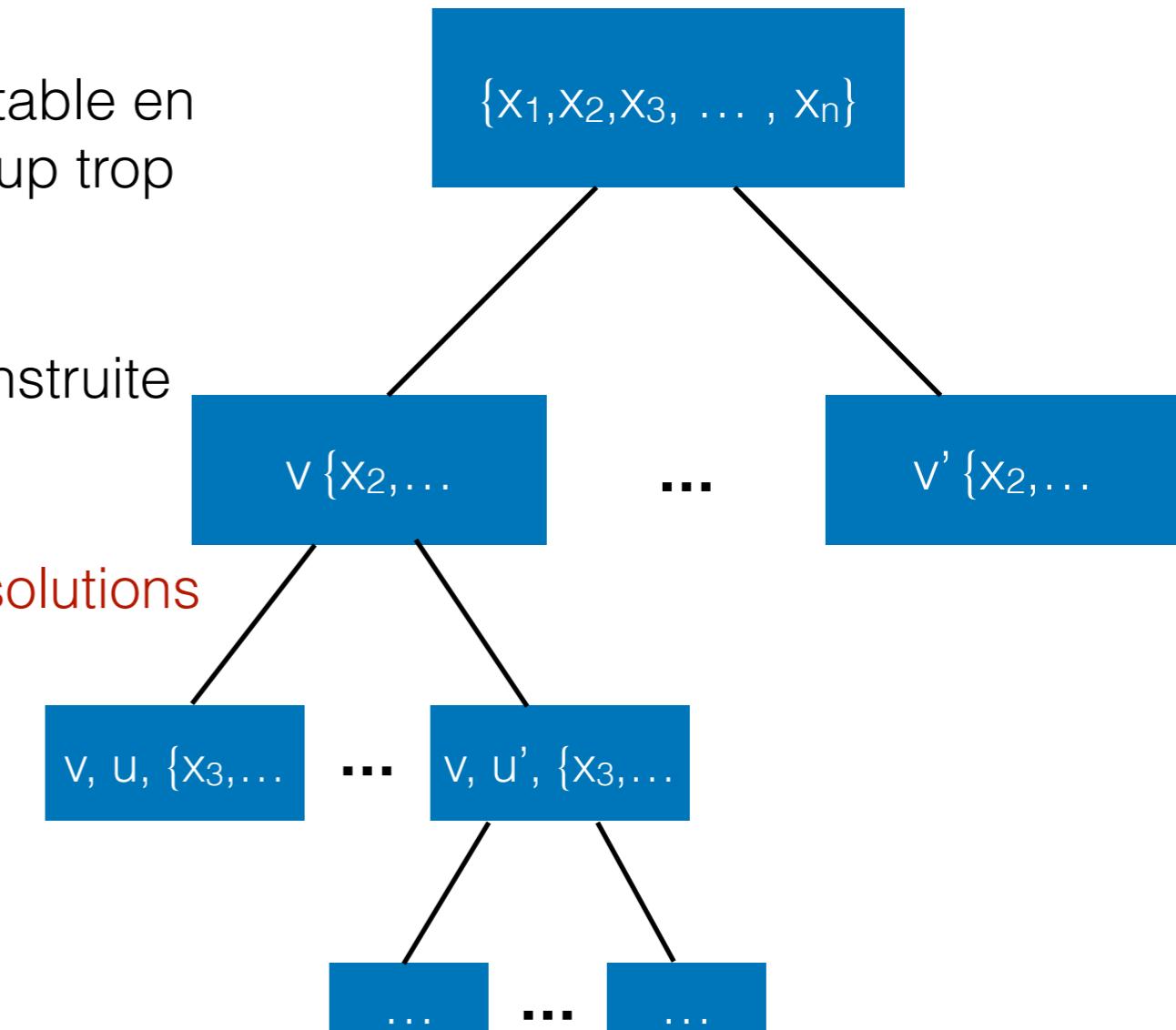
- (cf TP) La méthode du British Museum permet d'énumérer des solutions candidates ... intractable en général pour des problèmes NP-durs (beaucoup trop de solutions potentielles)

- (Alternative) une solution peut souvent être construite de façon incrémentale:

- on peut souvent représenter l'espace des solutions comme un arbre « de choix »

- Un noeud de l'arbre est une solution incomplète

- Explorer l'arbre en partant de rien et en rajoutant des composants de façon incrémentale



Algorithmes par énumération ‘intelligente’...

- Exploration récursive de l’arbre pour la décision
- Exploration récursive de l’arbre pour l’optimisation

Explorer(Solution sol) :

```
If sol est faisable then  
  If solution est complète then return Vrai,  
  Else  
    For chaque s successeur de sol do  
      Explorer(s);  
    Else  
      return Faux;
```

Solution sol^* : meilleure solution trouvée

Explorer(Solution sol, Solution sol^*) :

```
If sol est faisable then  
  If solution est complète then  
    If sol est meilleure que  $sol^*$  then  
       $sol^* := sol$  ;  
    Else  
      For chaque s successeur de sol do  
        Explorer(s,  $sol^*$ );  
  return meilleure solution trouvée ;
```

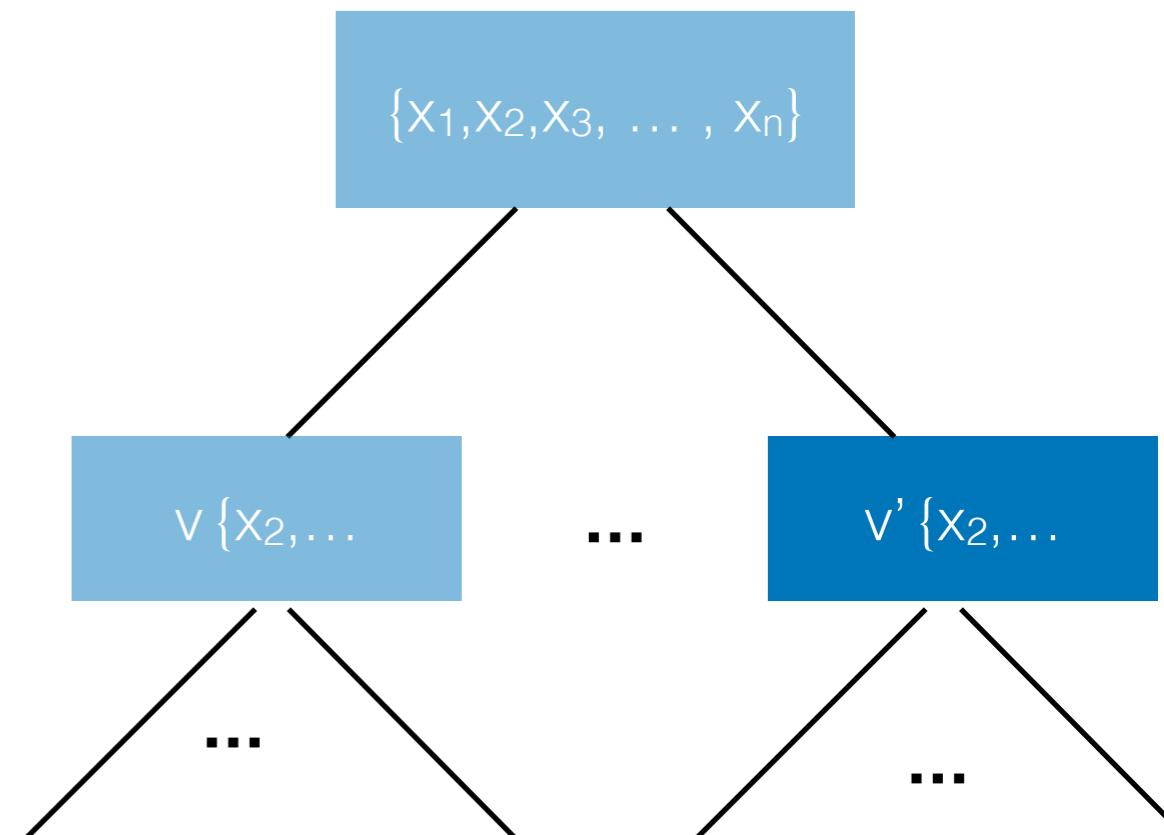
Algorithmes par énumération ‘intelligente’...

- Amélioration du parcours de l’arbre
 - Éviter d’explorer les sous-arbres qui mènent à cout sûr vers des solutions complètes non-optimales
 - Privilégier l’exploration des sous-arbres qui mèneraient vers la solution optimale
- Exemple du ‘Branch and Bound’ ou ‘Séparation et évaluation’
 - **Élagage** si on peut dire que toute solution qui complète la solution actuelle est moins bonne que la meilleure trouvée/connue
 - **Décomposer (Branch)** le problème et explorer en premier les choix qui paraissent le plus prometteurs

Branch and Bound (B&B) ou 'Séparation et Evaluation'

- **(Branch)** Séparation:

- Décomposer le problème en plusieurs sous-problèmes (branches) : diviser-pour-reigner



Branch and Bound (B&B) ou 'Séparation et Evaluation'

- **(Branch)** Séparation:

- Décomposer le problème en plusieurs sous-problèmes (branches) : diviser-pour-reigner

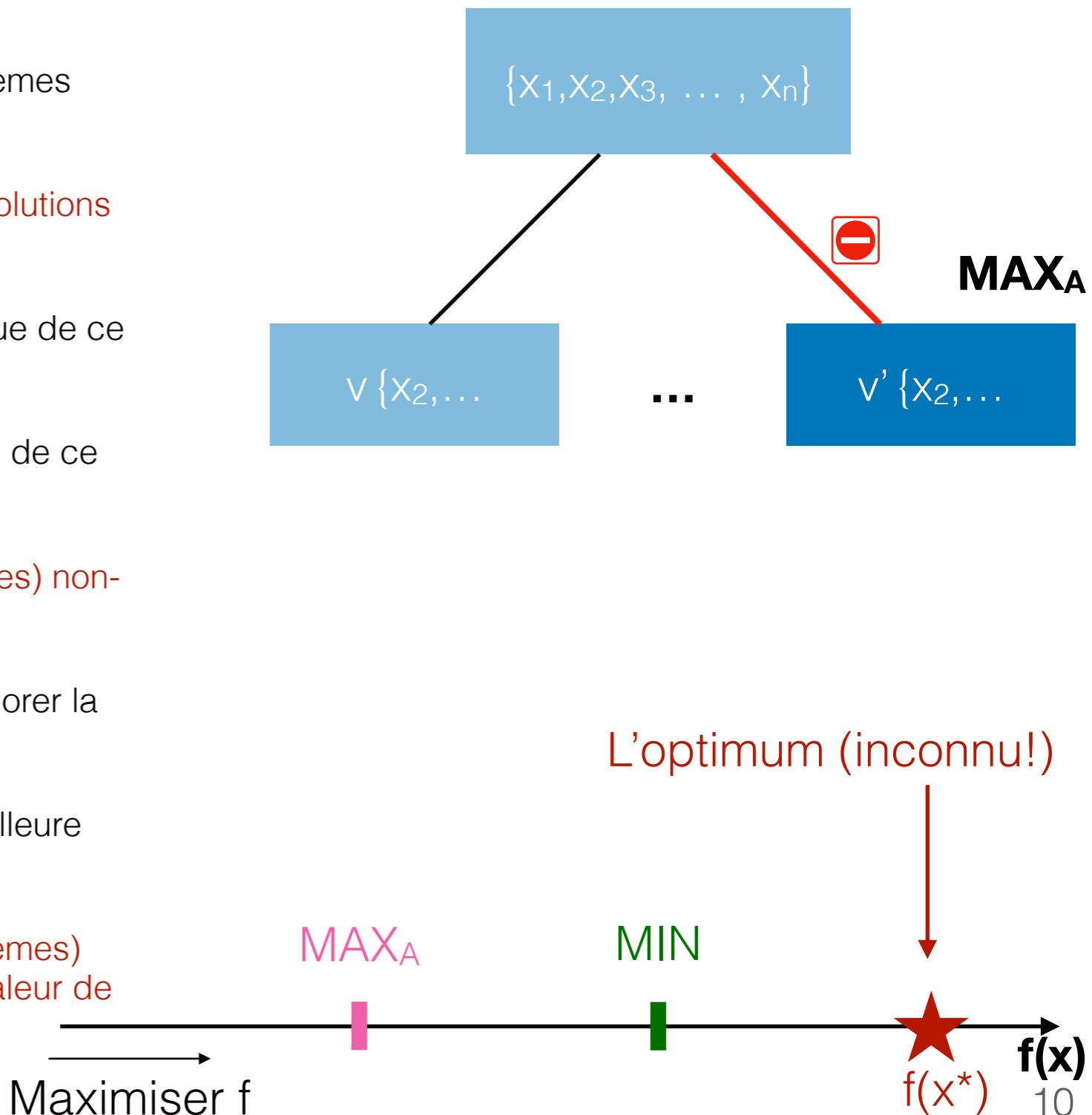
- **(Bound)** Évaluation: on évalue l'intérêt des nouvelles solutions partielles

- Borne supérieur (Upper Bound): toute solution issue de ce noeud aura un coût au plus de MAX
- Borne inférieur (Lower bound): toute solution issue de ce noeud aura un coût au moins de MIN

- **(pruning)** Élagage: éliminer les noeuds (sous-problèmes) non-optimaux

- (maximisation) Si $\text{MAX}_A \leq \text{MIN}_B$, on s'interdit d'explorer la branche A
- (maximisation) De même si $\text{MAX}_A \leq \text{MIN}$: la meilleure valeur connue (initialisation)

- **(Select)** Explorer en premier les branches (sous-problèmes) les plus prometteuses, e.g., celles ayant la meilleure valeur de MIN



Branch and Bound (B&B) ou 'Séparation et Evaluation'

- **(Branch)** Séparation:

- Décomposer le problème en plusieurs sous-problèmes (branches) : diviser-pour-reigner

- **(Bound)** Évaluation: on évalue l'intérêt des nouvelles solutions partielles

- Borne supérieur (Upper Bound): toute solution issue de ce noeud aura un coût au plus de MAX

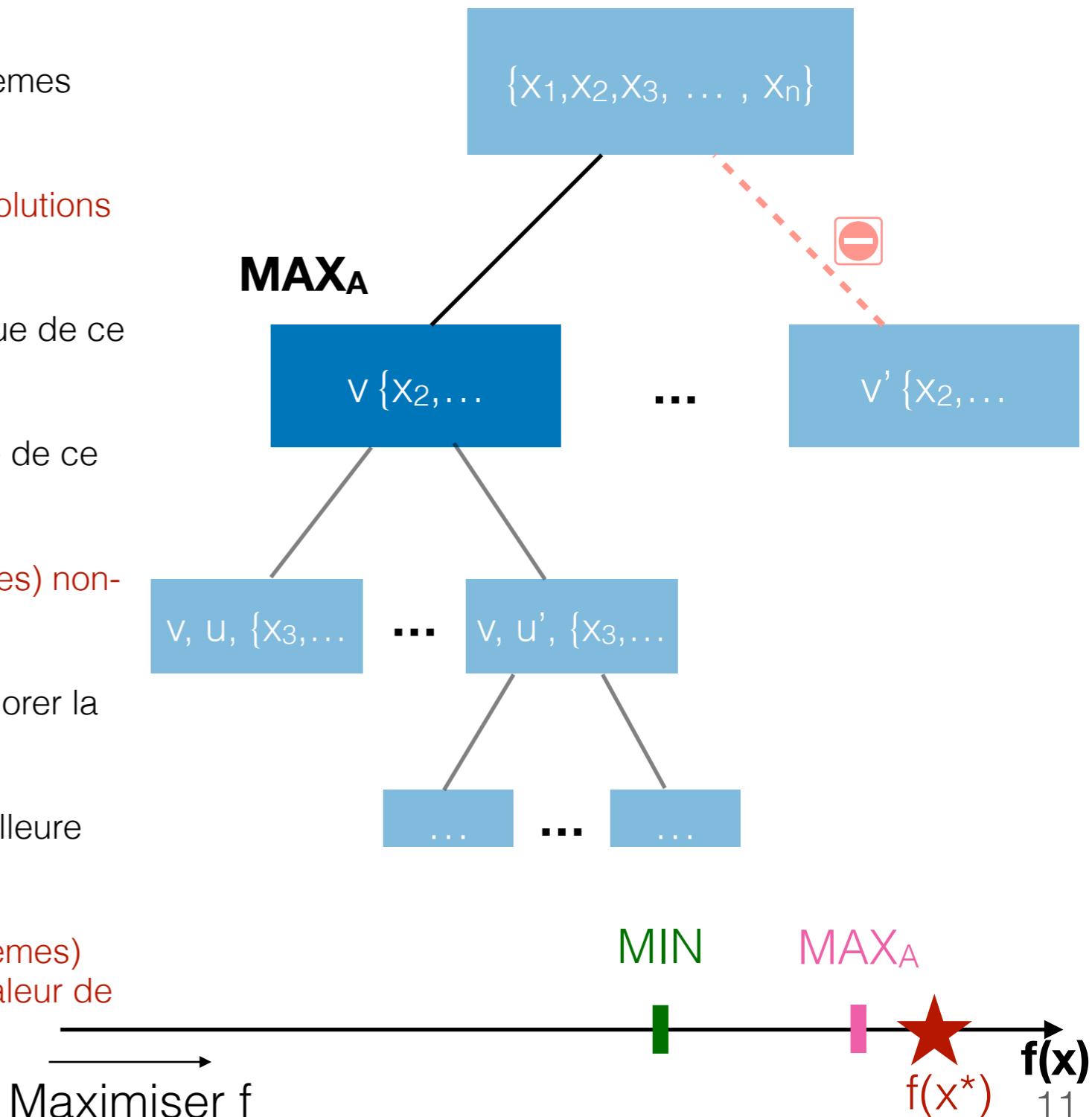
- Borne inférieur (Lower bound): toute solution issue de ce noeud aura un coût au moins de MIN

- **(pruning)** Élagage: éliminer les noeuds (sous-problèmes) non-optimaux

- (maximisation) Si $\text{MAX}_A \leq \text{MIN}_B$, on s'interdit d'explorer la branche A

- (maximisation) De même si la $\text{MAX}_A \leq \text{MIN}$: la meilleure valeur connue (initialisation)

- **(Select)** Explorer en premier les branches (sous-problèmes) les plus prometteuses, e.g., celles ayant la meilleure valeur de MIN

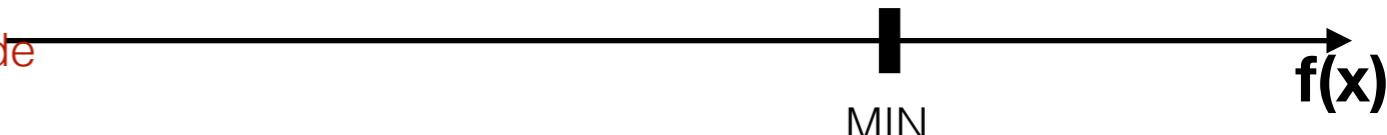


Branch and Bound (B&B) ou ‘Séparation et Evaluation’

- **Initialisation** : prendre les objets dans l'ordre décroissant des v_i/p_i
 - MIN = valeur du sac ainsi obtenu
- Représentation des solutions (partielles/incomplètes) par un arbre binaire (1 : on met l'objet; 0 : on ne le met pas)
- **Branch** : explorer l'objet avec le plus grand v_i/p_i restant
- **Bound** :
 - (Relaxation) remplir le sac avec les objets restants dans l'ordre des v_i/p_i , en fractionnant le dernier objet pour ne pas violer la contrainte
 - MAX : valeur du sac (virtuel) ainsi obtenu
- **Prune** :
 - Si la branche n'est pas faisable
 - Si $MAX \leq MIN$
- Attention à : la mise à jour du MIN, le choix des structures de données, la sélection des noeuds actif (BFS,DFS), etc

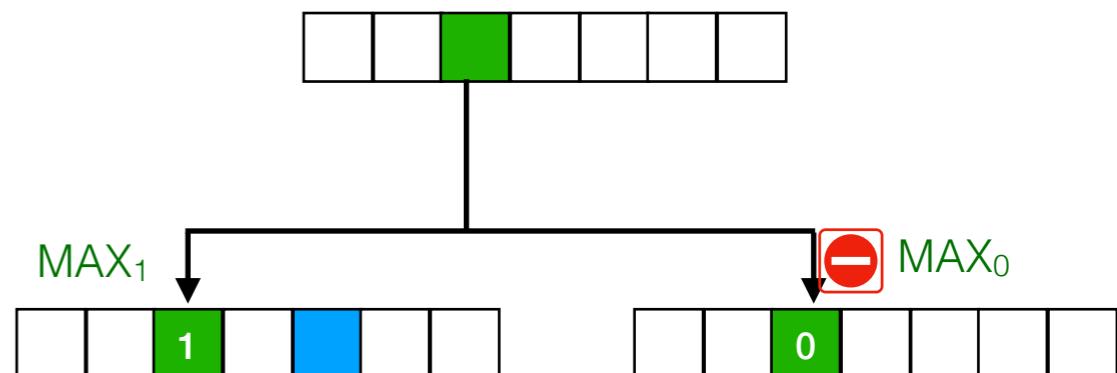
Branch and Bound (B&B) ou 'Séparation et Evaluation'

- **Initialisation** : prendre les objets dans l'ordre décroissant des v_i/p_i
 - MIN = valeur du sac ainsi obtenu
- Représentation des solutions (partielles/incomplètes) par un arbre binaire (1 : on met l'objet; 0 : on ne le met pas)
- **Branch** : explorer l'objet avec le plus grand v_i/p_i restant
- **Bound** :
 - (Relaxation) remplir le sac avec les objets restants dans l'ordre des v_i/p_i , en fractionnant le dernier objet pour ne pas violer la contrainte
 - MAX : valeur du sac (virtuel) ainsi obtenu
- **Prune** :
 - Si la branche n'est pas faisable
 - Si $MAX \leq MIN$
- Attention à : la mise à jour du MIN, le choix des structures de données, la sélection des noeuds actif (BFS,DFS), etc



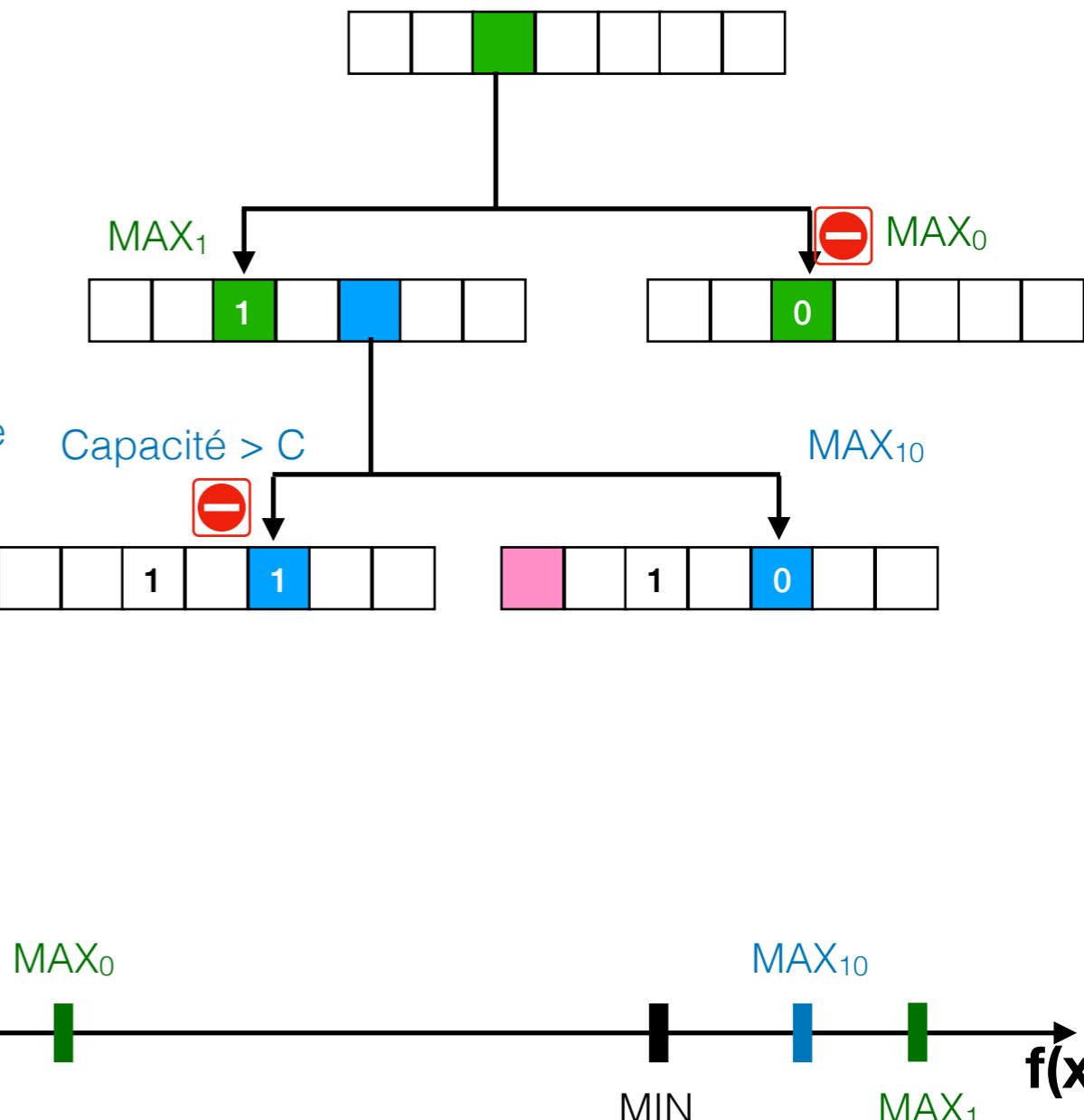
Branch and Bound (B&B) ou ‘Séparation et Evaluation’

- **Initialisation** : prendre les objets dans l'ordre décroissant des v_i/p_i
 - MIN = valeur du sac ainsi obtenu
- Représentation des solutions (partielles/incomplètes) par un arbre binaire (1 : on met l'objet; 0 : on ne le met pas)
- **Branch** : explorer l'objet avec le plus grand v_i/p_i restant
- **Bound** :
 - (Relaxation) remplir le sac avec les objets restants dans l'ordre des v_i/p_i , en fractionnant le dernier objet pour ne pas violer la contrainte
 - MAX : valeur du sac (virtuel) ainsi obtenu
- **Prune** :
 - Si la branche n'est pas faisable
 - Si $\text{MAX} \leq \text{MIN}$
 - Attention à : la mise à jour du MIN, le choix des structures de données, la sélection des noeuds actif (BFS,DFS), etc



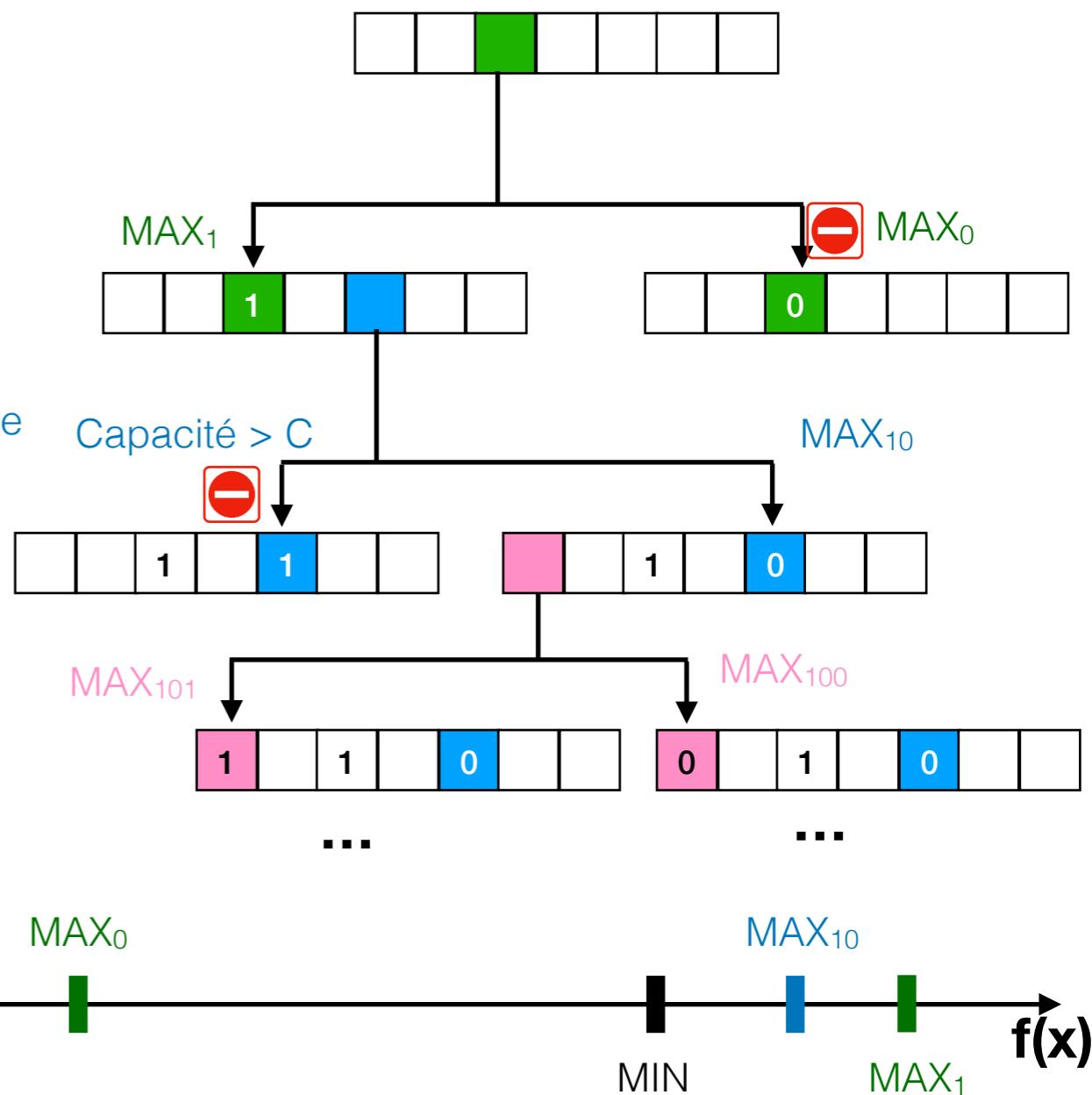
Branch and Bound (B&B) ou ‘Séparation et Evaluation’

- **Initialisation** : prendre les objets dans l'ordre décroissant des v_i/p_i
 - MIN = valeur du sac ainsi obtenu
- Représentation des solutions (partielles/incomplètes) par un arbre binaire (1 : on met l'objet; 0 : on ne le met pas)
- **Branch** : explorer l'objet avec le plus grand v_i/p_i restant
- **Bound** :
 - (Relaxation) remplir le sac avec les objets restants dans l'ordre des v_i/p_i , en fractionnant le dernier objet pour ne pas violer la contrainte
 - MAX : valeur du sac (virtuel) ainsi obtenu
- **Prune** :
 - Si la branche n'est pas faisable
 - Si $\text{MAX} \leq \text{MIN}$
 - Attention à : la mise à jour du MIN, le choix des structures de données, la sélection des noeuds actif (BFS,DFS), etc



Branch and Bound (B&B) ou ‘Séparation et Evaluation’

- **Initialisation** : prendre les objets dans l'ordre décroissant des v_i/p_i
 - MIN = valeur du sac ainsi obtenu
- Représentation des solutions (partielles/incomplètes) par un arbre binaire (1 : on met l'objet; 0 : on ne le met pas)
- **Branch** : explorer l'objet avec le plus grand v_i/p_i restant
- **Bound** :
 - (Relaxation) remplir le sac avec les objets restants dans l'ordre des v_i/p_i , en fractionnant le dernier objet pour ne pas violer la contrainte
 - MAX : valeur du sac (virtuel) ainsi obtenu
- **Prune** :
 - Si la branche n'est pas faisable
 - Si $\text{MAX} \leq \text{MIN}$
 - Attention à : la mise à jour du MIN, le choix des structures de données, la sélection des noeuds actif (BFS,DFS), etc

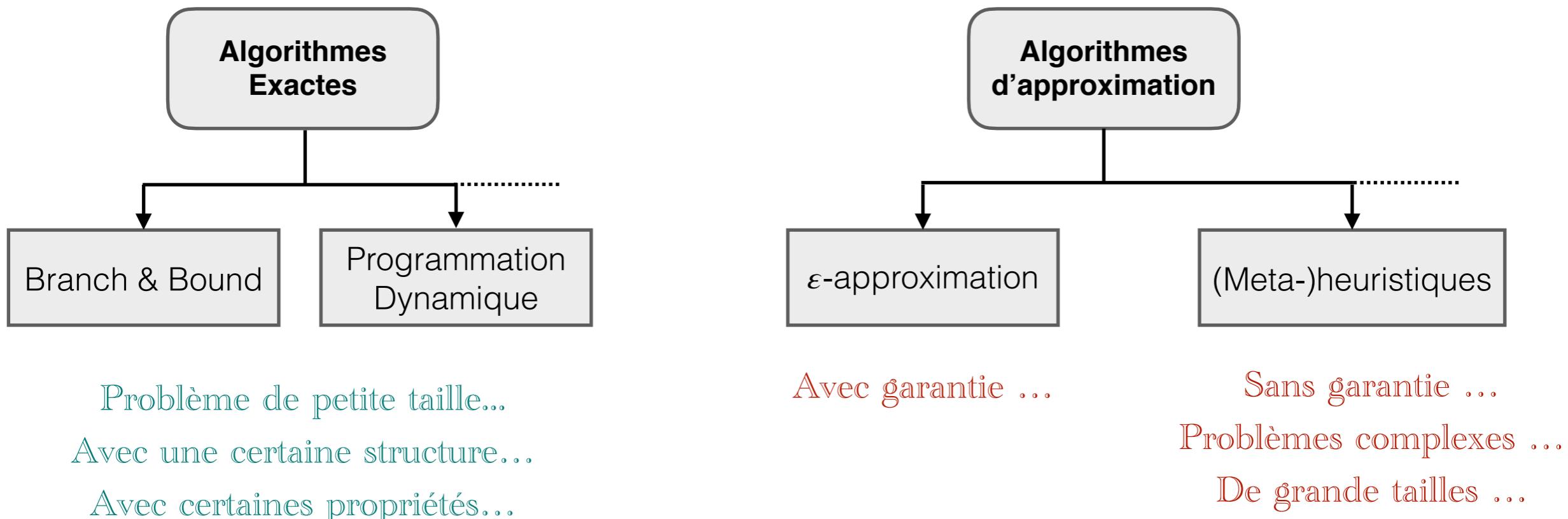


Problème NP-dur: que faire ?

- S'acharner à chercher un algorithme polynomial en espérant montrer $P = NP \dots$
- **Abandonner l'idée d'avoir un algorithme polynomial:**
 - Pour certains problèmes NP-durs, il existe des **algorithmes pseudo-polynomiaux**
 - Utiliser la spécificité des données; on peut parfois concevoir des algorithmes "efficaces à paramètres fixés" (algorithmes FPT, **fixed-parameter tractable**) ...
 - Utiliser des méthodes « améliorées » d'énumération des solutions: exploiter la symétrie des solutions, élaguer l'arbre des solutions (pruning), prioriser les branches les plus prometteuses - **branch and bound**, etc
 - Utiliser les **solveurs existants**, par exemple des solveurs SAT, de programmation linéaire en entiers, TSP, etc
- **Cela fonctionne-t-il vraiment pour des problèmes de grande taille, complexe, indépendamment de la nature/structure du problème, ... ?**

Problème NP-dur: que faire ?

- **Algorithmes exactes/complets** : On veut trouver un algorithme qui trouve toujours la solution optimale
- **Algorithmes d'approximation (heuristiques)** : On veut un algorithme qui fournit toujours une solution correcte, de ‘bonne’ qualité, mais pas forcément optimale, et qui soit efficace (polynomial)
 - Heuristiques : Art d’inventer, de faire des découvertes ; Du grec ancien *heurískô* (trouver), dont est aussi issu *eurêka*



Problème NP-dur: que faire ?

- **Algorithmes exactes/complets** : On veut trouver un algorithme qui trouve toujours la solution optimale
- **Algorithmes d'approximation (heuristiques)** : On veut un algorithme qui fournit toujours une solution correcte, de ‘bonne’ qualité, mais pas forcément optimale, et qui soit efficace (polynomial)
 - Heuristiques : Art d’inventer, de faire des découvertes ; Du grec ancien *heurískô* (trouver), dont est aussi issu *eurêka*
 - Plusieurs types d’approches des plus simples aux plus complexes
 - Qu'est ce qu'une bonne heuristique ?
 - Elle est de complexité raisonnable (idéalement polynomiale ; en tout cas efficace en pratique)
 - Robustesse : Elle fournit le plus souvent une solution proche de l’optimum ; la probabilité d’obtenir une solution de mauvaise qualité est faible ; etc
 - Simple et générique : peut éventuellement s’appliquer à plusieurs problèmes avec un minimum d’effort d’ingénierie !

Paradigmes de recherche

- **(Une vue globale) De façon itérative, générer et évaluer des solutions potentielles (candidates)**
 - Problème de décision : evaluation = test si effectivement une solution ; Problème d'optimisation : evaluation = vérifier le coût de la solution (fonction objective)
- **Espace de recherche : ensemble de solutions (potentielles)**
 - Comment chercher de façon efficace = quel chemin prendre dans un espace de recherche pour atteindre une ‘bonne’ solution ?

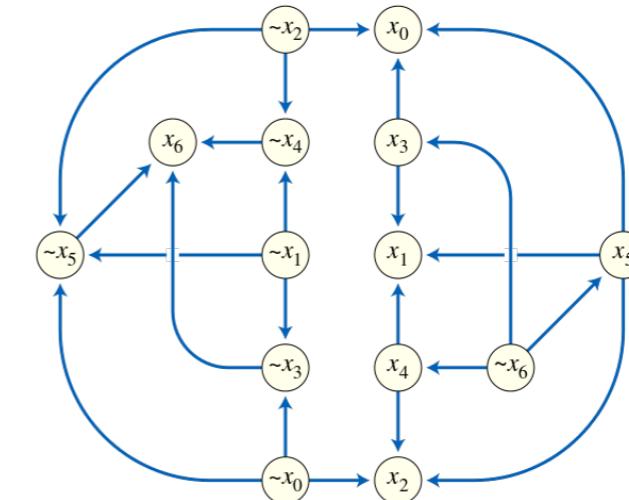
Paradigmes de recherche

- **(Une vue globale) De façon itérative, générer et évaluer des solutions potentielles (candidates)**

- Problème de décision : evaluation = test si effectivement une solution ; Problème d'optimisation : evaluation = vérifier le coût de la solution (fonction objective)

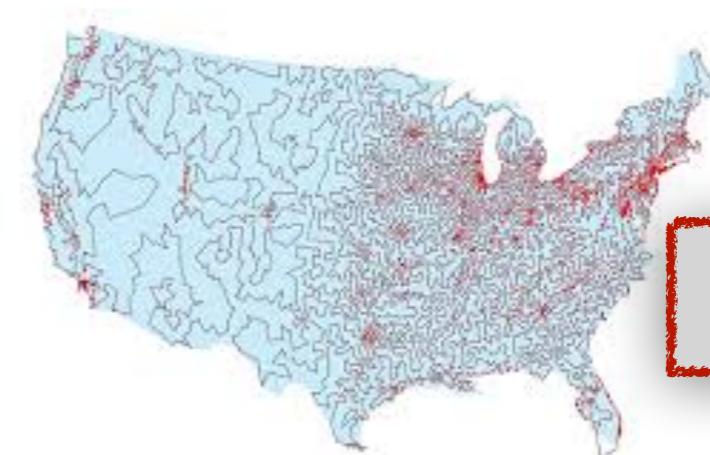
- **Espace de recherche : ensemble de solutions (potentielles)**

- Comment chercher de façon efficace = quel chemin prendre dans un espace de recherche pour atteindre une ‘bonne’ solution ?
- L'espace de recherche n'est pas toujours égal à l'espace des solutions réalisable, e.g., problème de décision, knapsack, etc
- Propriétés de l'espace de recherche
 - **Taille** : liée à la représentation que l'on se fait des solutions potentielles
 - **Structure** : liée au problème et à la façon avec laquelle on se ballade dans l'espace de recherche



SAT

Pour $n = 100$; $\sim 10^{30}$ solutions potentielles
1000 solutions / secondes : 15 billions d'années
(Big bang) pour évaluer moins de 1% de l'espace



TSP

Pour $n = 50$ villes; 10^{62} solutions
Il existe $\sim 10^{21}$ litres d'eau sur terre

Paradigmes de recherche

- **Recherche systématique / complète**

- 'Traverser/parcourir' l'espace de recherche de façon systématique
- Evaluate (implicitement) toute solution jusqu'à solution optimale trouvée (B&B, A*, ...).

- **Recherche Constructive (le retour des gloutons!)**

- Espace de recherche = solutions candidates partielles
- Etape de recherche = extension avec un ou plusieurs nouveaux composants

- **Recherche perturbative**

- Espace de recherche = ensemble complet de solutions candidates
- Une étape de recherche = modification d'une ou plusieurs composants d'une solution

- **Recherche Locale**

- Commencer avec une solution initiale
- 'itérativement' visiter une solution 'voisine'
- Autres ...

Paradigmes de recherche

- **Recherche systématique / complète**

- 'Traverser/parcourir' l'espace de recherche de façon systématique
- Evaluate (implicitement) toute solution jusqu'à solution optimale trouvée (B&B, A*, ...).

- **Recherche Constructive (le retour des gloutons!)**

- Espace de recherche = solutions candidates partielles
- Etape de recherche = extension avec un ou plusieurs nouveaux composants

- **Recherche perturbative**

- Espace de recherche = ensemble complet de solutions candidates
- Une étape de recherche = modification d'une ou plusieurs composants d'une solution

- **Recherche Locale**

- Commencer avec une solution initiale
- 'itérativement' visiter une solution 'voisine'
- Autres ...

Recherche constructive (le retour des gloutons)

- Deux principaux ‘composants’ :
 - Comment définir la notion de composant ?
 - Comment choisir le prochain composant à ajouter ?

$S = \emptyset$

Tant que S n'est pas une solution complète faire

$c :=$ choisir un nouveau **composant** ;

$S := S + c$

Fin Tant que

Recherche constructive (le retour des gloutons)

- Deux principaux ‘composants’ :
 - **Comment définir la notion de composant ?**
 - **Comment choisir le prochain composant à ajouter ?**
- Schéma typique :
 - Les composants C = les variables de décisions (pas toujours effectif)
 - Fonction heuristique de qualité/score d'une composante ($h : C \rightarrow \text{IR}$)
 - Choisir la composante ayant le meilleur qualité/score (selon h)
 - Qualité / score : stratégie statique (indépendant de la solution partielle) vs. dynamique (tient en compte les composants déjà ajoutés)
- Complémentaire aux autres types d'approches
 - c.f. B&B : utilisée pour construire une solution de départ (borne inf) raisonnablement bonne
 - Les propriétés étudiées dans ce cadre permettent d'avoir une connaissance spécifique aux problèmes et de mieux les résoudre (voir plus loin)

S = Vide

Tant que S n'est pas une solution complète **faire**

c := **choisir** un nouveau **composant** ;

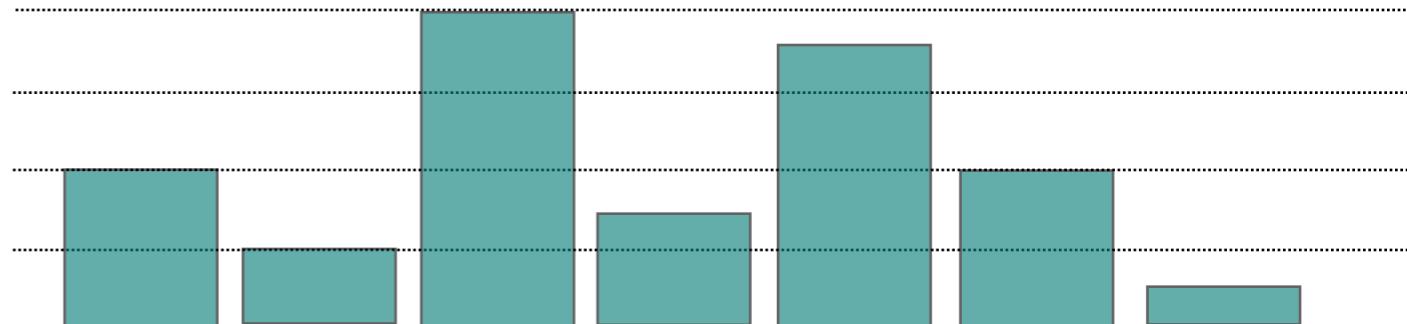
S := S + c

Fin Tant que

Exemple #1: Bin Packing

- Donnée:

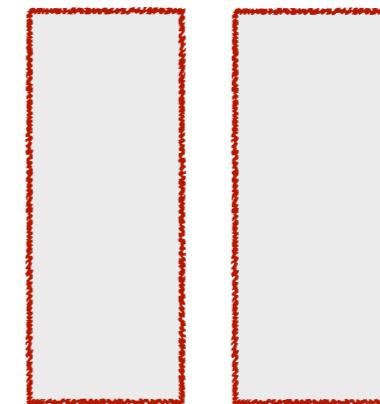
- n objets de poids p_1, p_2, \dots, p_n



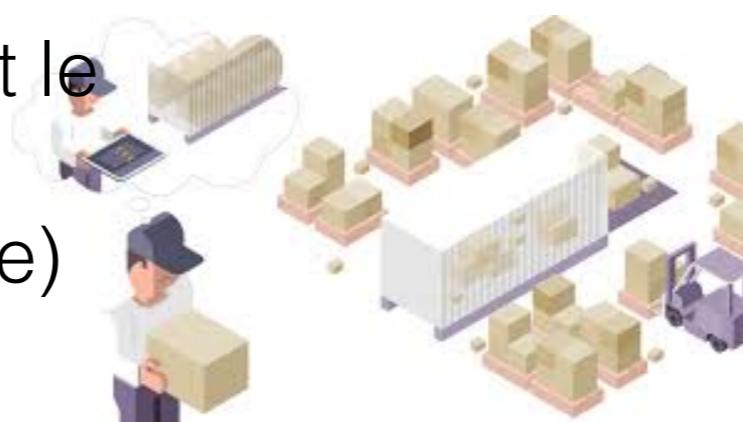
- Des sacs de capacité P chacun

- Problème:

- Mettre les objets dans le minimum de sacs

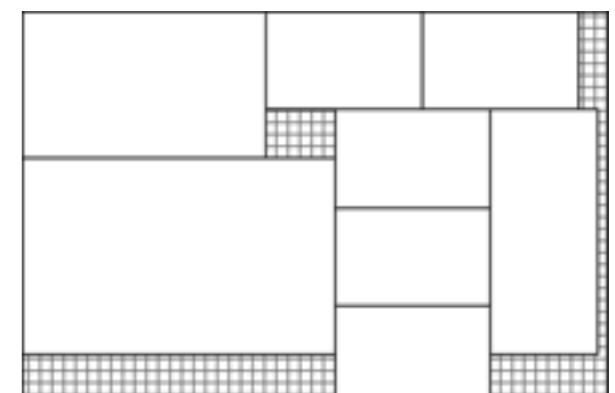


- Aff : $[1,n] \rightarrow [1,k]$ tel que k est le plus petit possible (et la contrainte de capacité satisfaite)



- **Bin Packing est NP-dur**

Logistique



Découpage

Exemple #1: Bin Packing

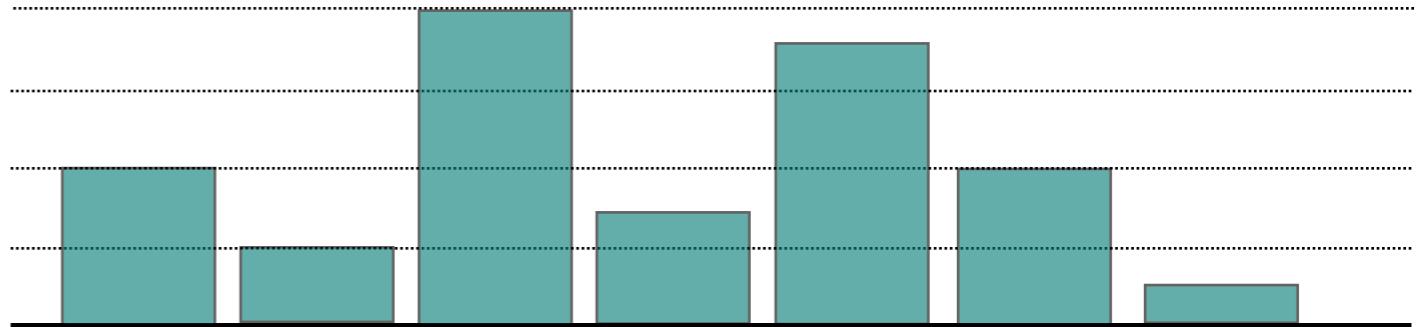
- **Un composant ?**

- l'affectation d'un objet à un sac

- **Fonction heuristique de choix ?**

- NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau

```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

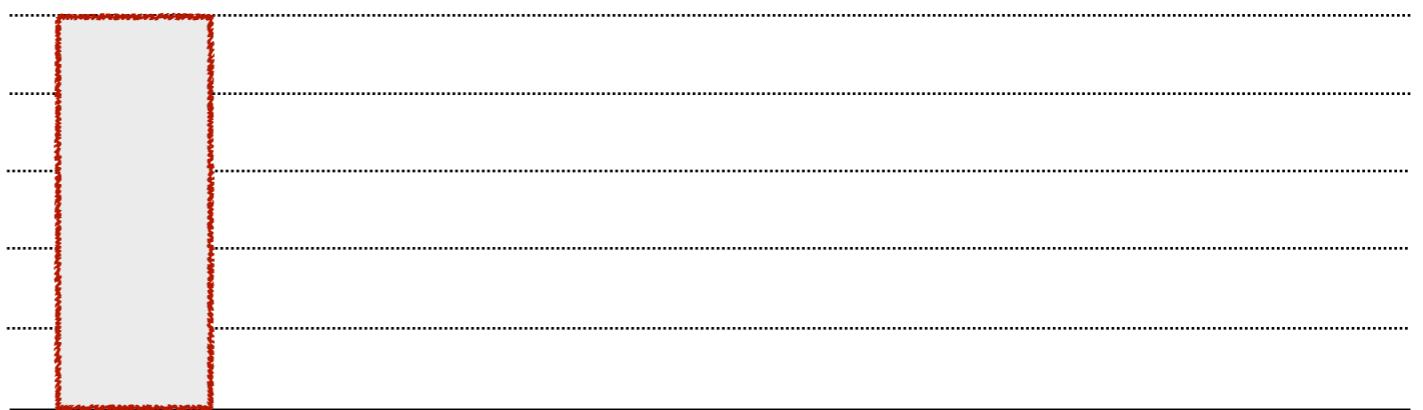
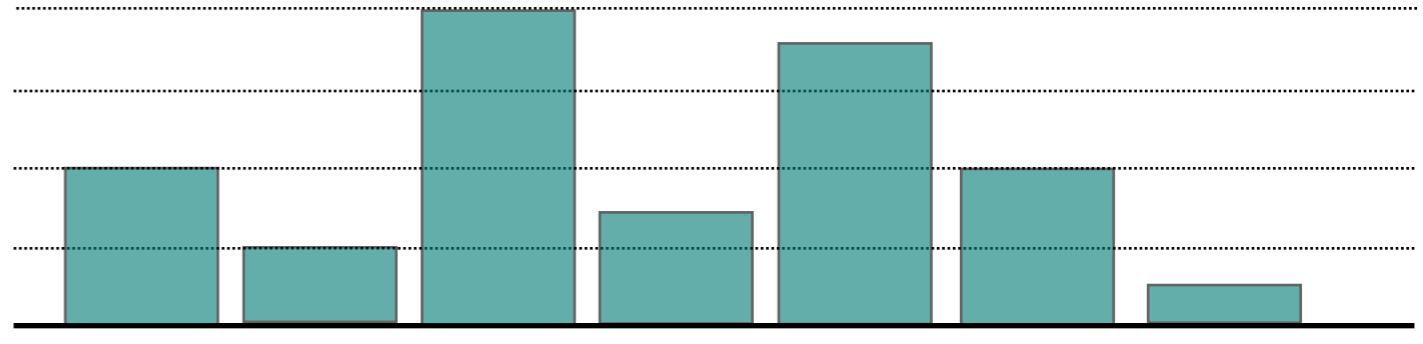
- **Un composant ?**

- l'affectation d'un objet à un sac

- **Fonction heuristique de choix ?**

- NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau

```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

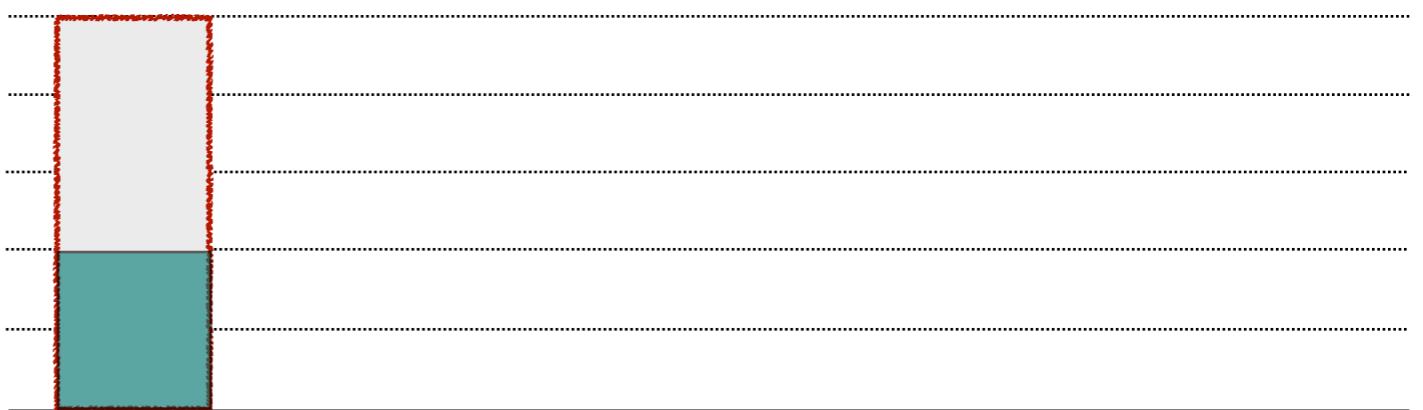
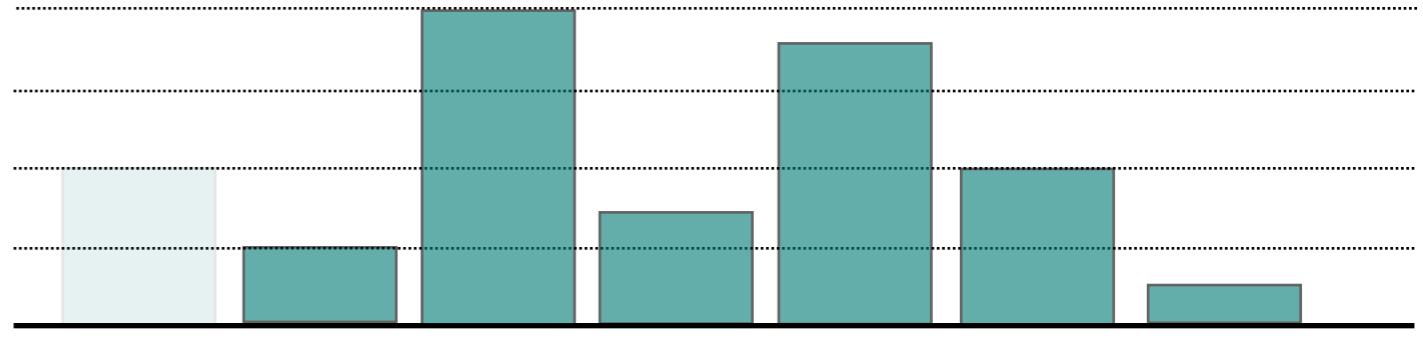
- **Un composant ?**

- l'affectation d'un objet à un sac

- **Fonction heuristique de choix ?**

- NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau

```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

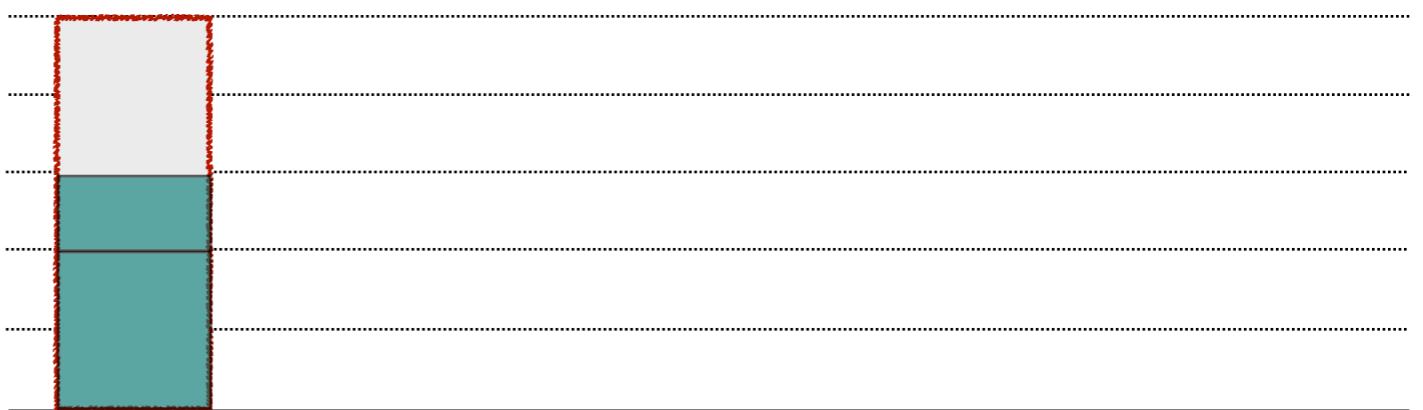
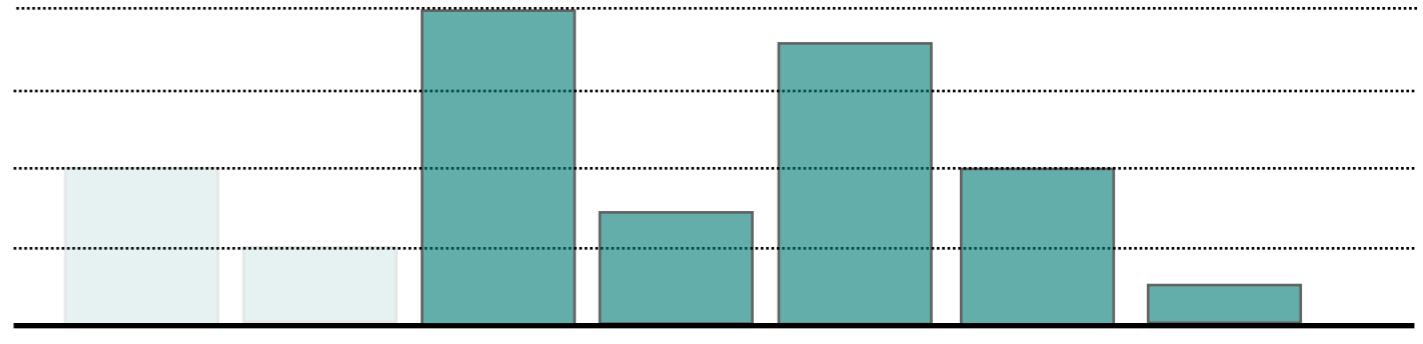
- **Un composant ?**

- l'affectation d'un objet à un sac

- **Fonction heuristique de choix ?**

- NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau

```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

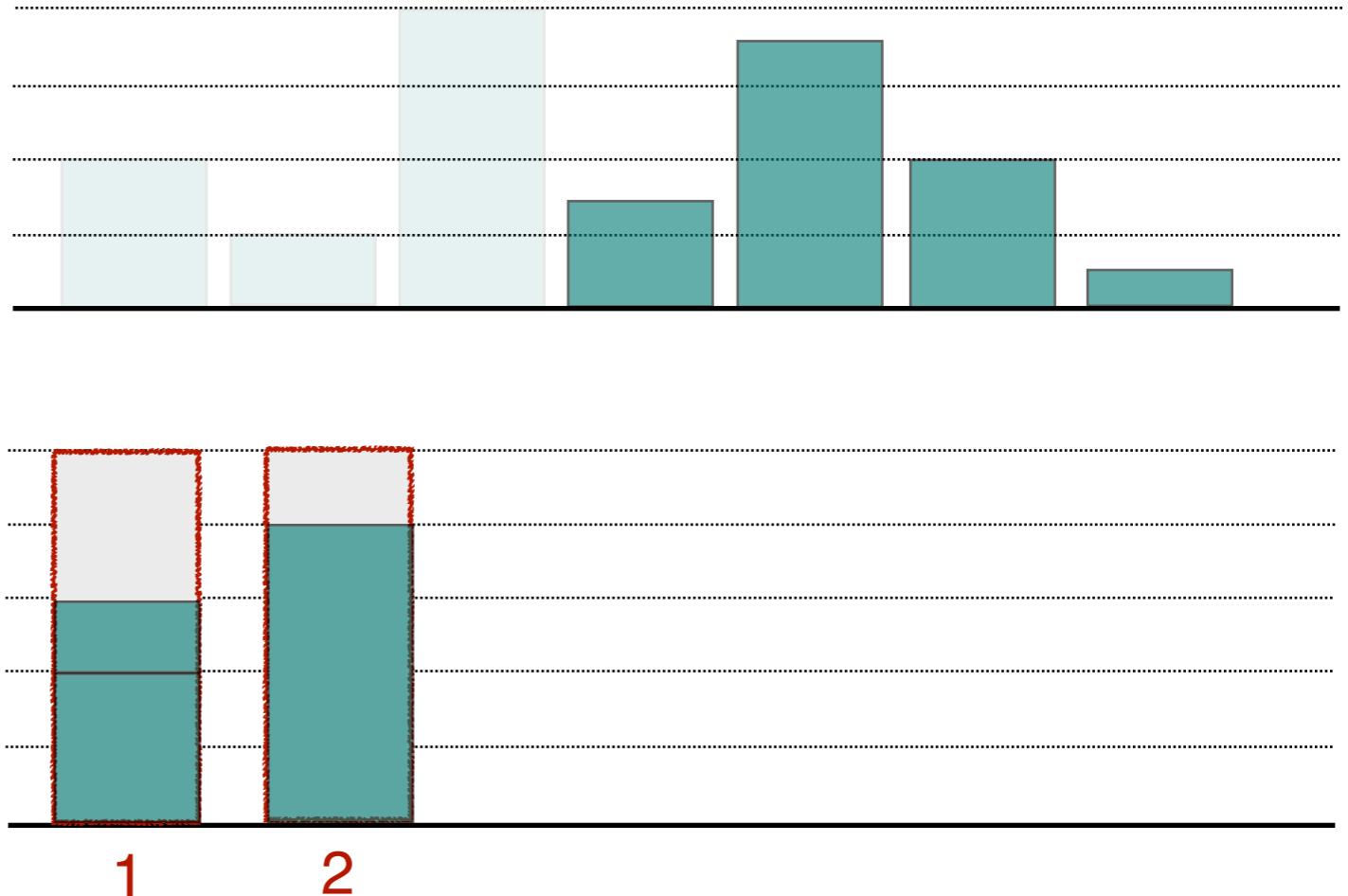
- **Un composant ?**

- l'affectation d'un objet à un sac

- **Fonction heuristique de choix ?**

- NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau

```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

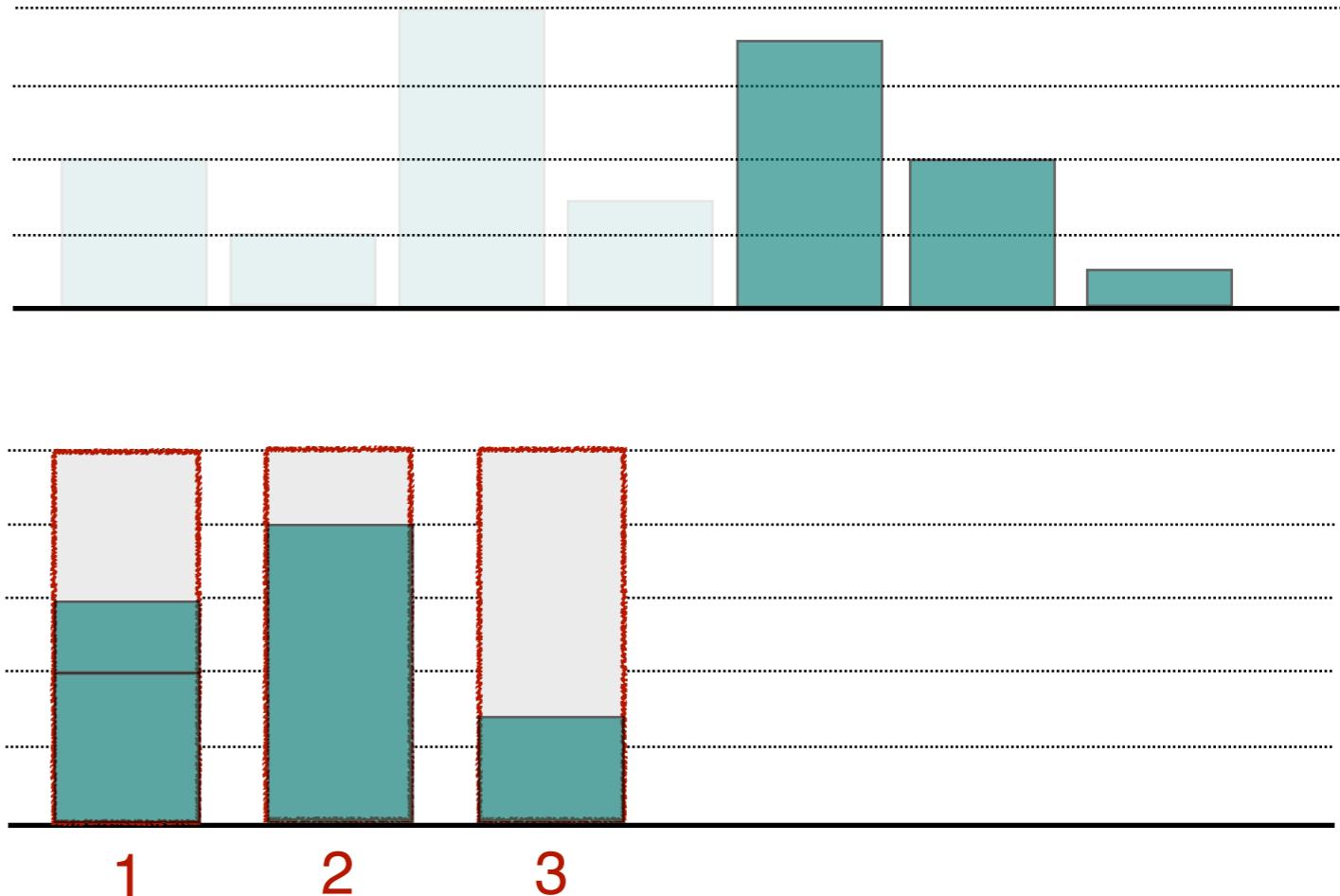
- **Un composant ?**

- l'affectation d'un objet à un sac

- **Fonction heuristique de choix ?**

- NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau

```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

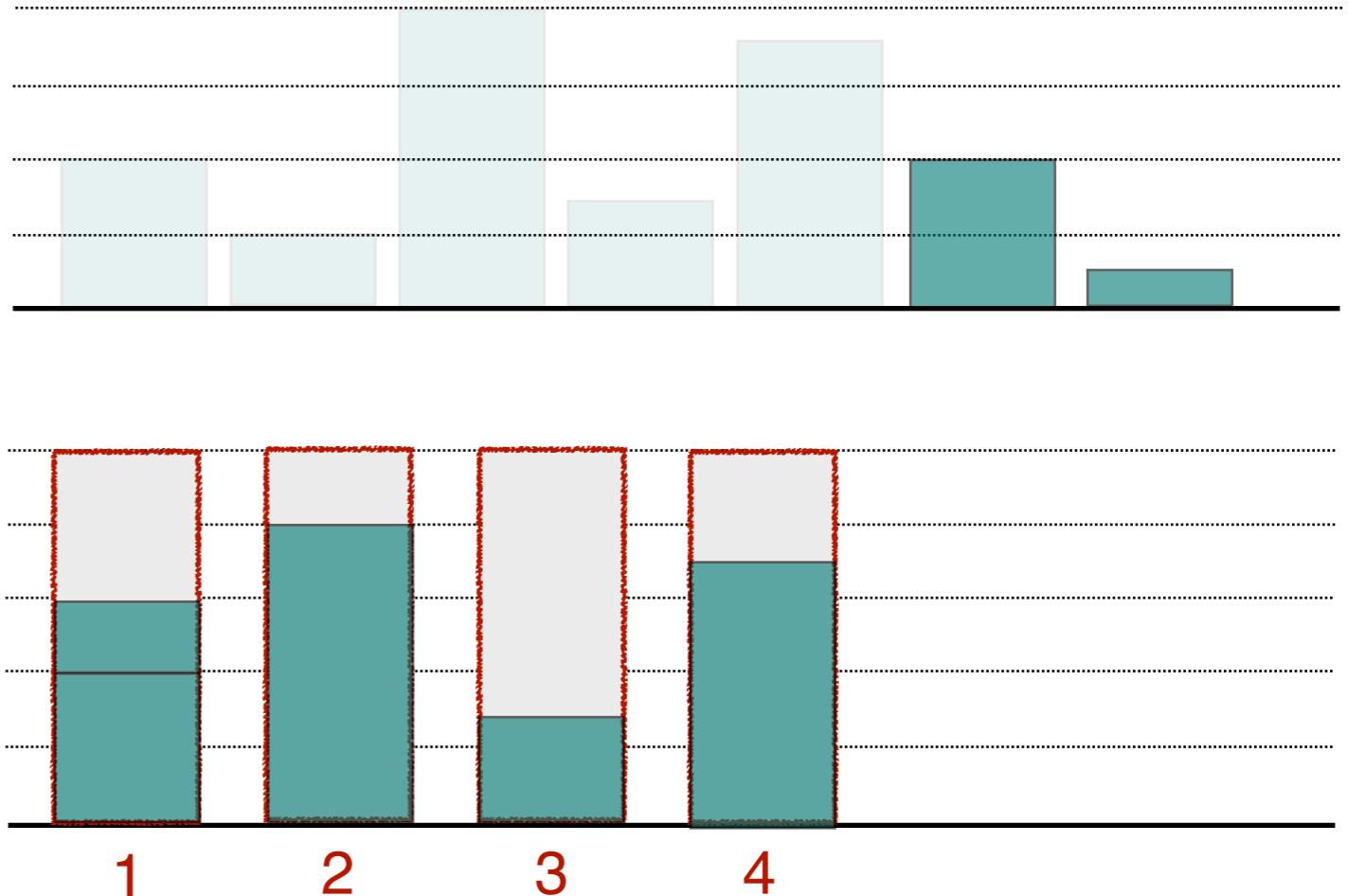
- **Un composant ?**

- l'affectation d'un objet à un sac

- **Fonction heuristique de choix ?**

- NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau

```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

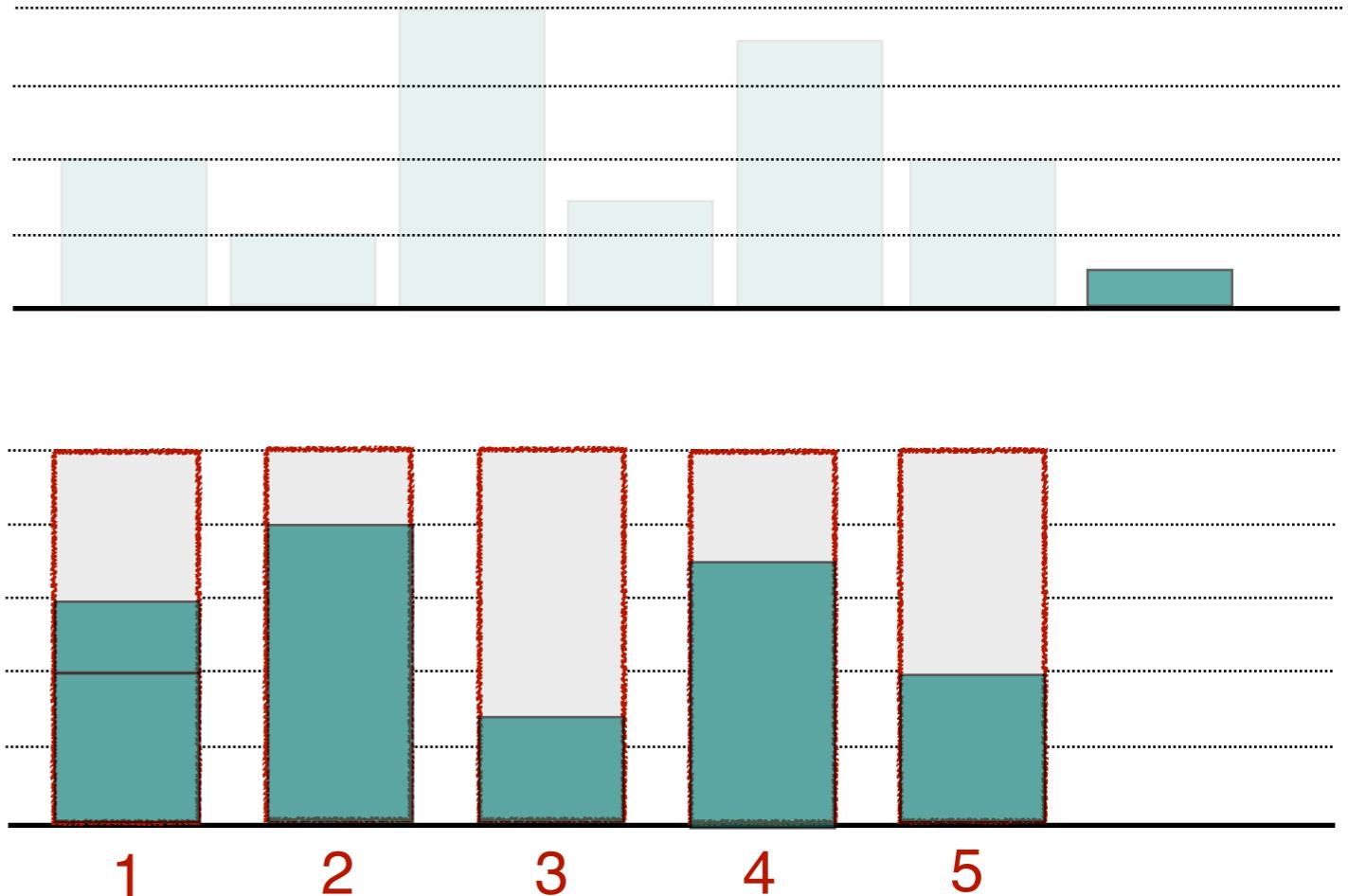
- **Un composant ?**

- l'affectation d'un objet à un sac

- **Fonction heuristique de choix ?**

- NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau

```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

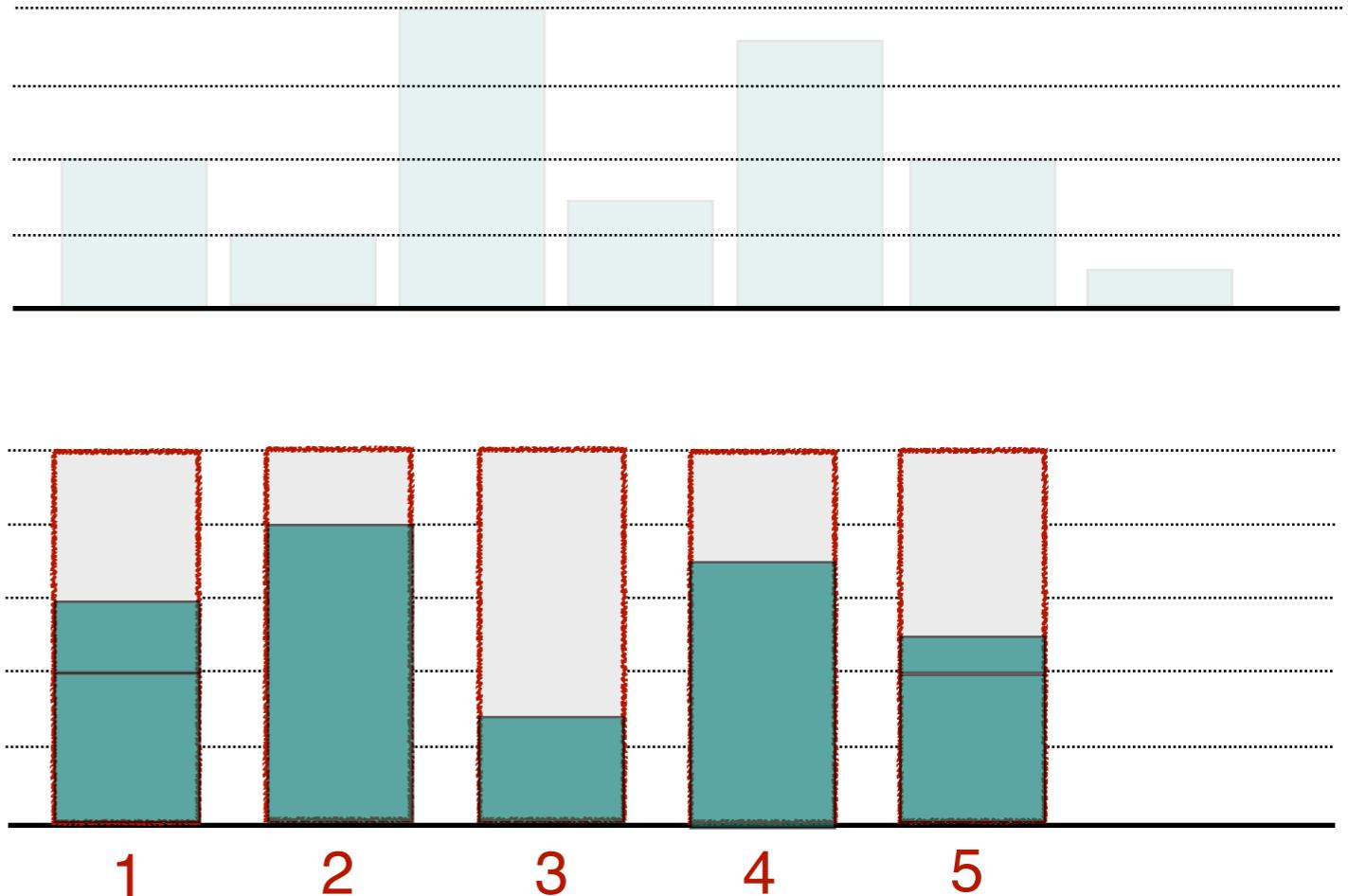
- **Un composant ?**

- l'affectation d'un objet à un sac

- **Fonction heuristique de choix ?**

- NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau

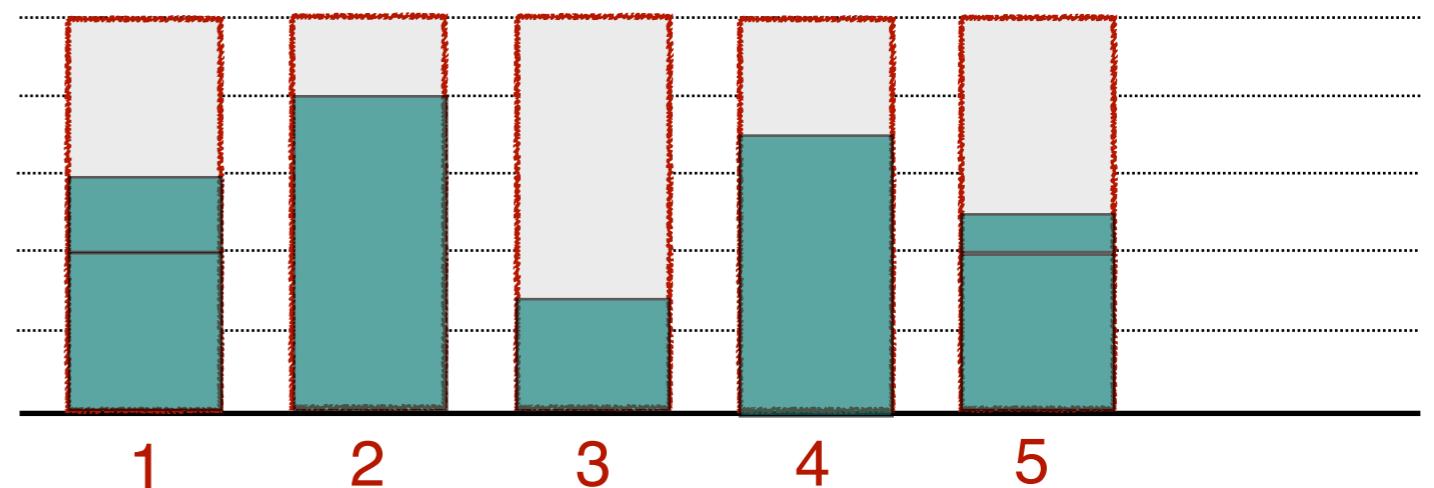
```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

- **Un composant ?**
 - l'affectation d'un objet à un sac
- **Fonction heuristique de choix ?**
 - NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau
 - L'algorithme est en $O(n)$!
 - Quelle garantie a-t-on par rapport à la qualité de la solution trouvée ?
 - Clairement pas optimale ! De combien ?

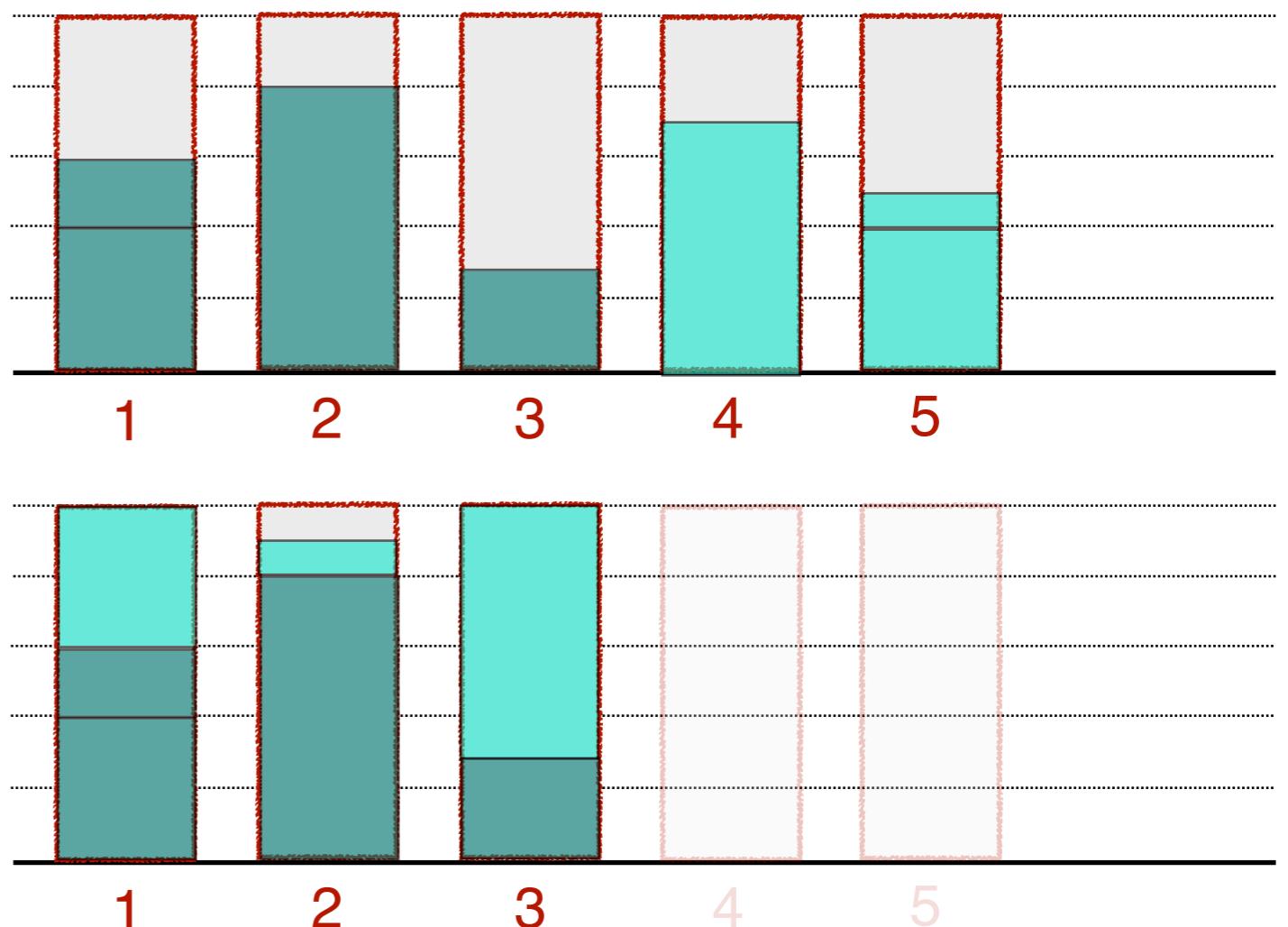
```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

- **Un composant ?**
 - l'affectation d'un objet à un sac
- **Fonction heuristique de choix ?**
 - NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau
 - L'algorithme est en $O(n)$!
 - Quelle garantie a-t-on par rapport à la qualité de la solution trouvée ?
 - Clairement pas optimale ! De combien ?

```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```



Exemple #1: Bin Packing

- **Un composant ?**
 - l'affectation d'un objet à un sac
 - **Fonction heuristique de choix ?**
 - NextFit (ou Heuristique de la ménagère?) : affecter un objet au sac courant sinon en créer un nouveau
- L'algorithme est en $O(n)$!
 - Quelle garantie a-t-on par rapport à la qualité de la solution trouvée ?
 - Clairement pas optimale ! De combien ?

```
ns = 1; //le premier sac
c = P; //place restante dans le sac courant
for i = 1 à n do
    if (pi ≤ c) then
        //l'objet rentre dans sac courant
        aff(i) = ns;
        // la capacité restante diminue
        c = c - pi;
    else
        // on le met dans un nouveau sac
        ns++;
        aff(i) = ns;
        c = P - pi;
    end else
end for
```

Soit k le nombre de sacs utilisés par l'heuristique:

- le poids des objets des sacs 1 et 2 > P
- le poids des objets des sacs 3 et 4 > P
-
- le poids des objets des sacs $2*i - 1$ et $2*i$ > P
- le poids des objets des sacs $2*(k/2) - 1$ et $2*(k/2)$ > P

=> la somme $S(p_1..p_n)$ des poids des objets > $P*k/2$

Or l'optimale ne peut pas utiliser moins de $S(p_1..p_n)/P$ sacs

Soit k^* le nombre sac de la solution optimal

$$k^* \geq S(p_1..p_n)/P > P*k/2 / P = k/2 \Rightarrow k^* \geq k/2 + 1 \Rightarrow k/k^* \leq 2$$

$\frac{k(\text{NextFit})}{k(\text{OPT})} \leq 2 \text{ (en fait = ?)}$

Ratio de garantie

- f : la fonction objectif à optimiser
- I une instance du problème d'optimisation (une donnée particulière du problème)
 - $\text{OPT}(I)$: la solution optimale (inconnue)
 - $A(I)$: la solution produite par un algorithme A (connue)

- **Si f est à maximiser, le ratio de garantie de A est :**

$$\sup_{I \text{ instance}} \frac{f(\text{OPT}(I))}{f(A(I))}$$

- **Si f est à minimiser, le ratio de garantie de A est :**

$$\sup_{I \text{ instance}} \frac{f(A(I))}{f(\text{OPT}(I))}$$

Ratio de garantie

- **Le ratio « exprime » le pire des cas ...**
 - Il est toujours supérieur ou égal à 1
 - S'il est égal à 1 alors l'algorithme est exacte
 - Plus il est proche de 1, meilleure est la garantie de l'approximation
- **Le ratio de garantie n'est pas toujours fini pour un algorithme et un problème donné !**
- **Peut-on trouver un algorithme (polynomiale) avec un ratio $(1+\varepsilon)$ pour un problème ? Pour quelle valeur de ε ?**
 - Cela dépend du problème \Rightarrow Autres classes de complexité (non abordées ici)
 - Pour certains problèmes, on peut montrer que quel que soit le ratio de garantie cible, on peut trouver un algorithme polynomial qui offre cette garantie (attention en pratique à la dépendance entre complexité et garantie !)
 - Pour certains problèmes, on peut montrer que, pour un ratio de garantie particulier cible, on ne peut pas trouver d'algorithme polynomial qui offre ce ratio de garantie, sauf si $P = NP$
 - Exemple : Si il existait une heuristique polynomiale de garantie strictement inférieure à $3/2$ pour Bin Packing, on aurait $P=NP$ (Exercice: reduction vers le problème Partition)
 - (Pire encore) Pour certains problèmes, on peut montrer que, quel que soit un ratio de garantie constant, on ne peut pas trouver d'algorithme polynomial qui offre ce ratio de garantie, sauf si $P = NP$ (exemple TSP, voir plus loin)

Heuristiques : restons optimiste !

- (Bin Packing) Peut on faire mieux : un océan d'heuristiques ?
 - Heuristique 1 (NextFit)
 - Heuristique 2 (FirstFit) : Pour chaque objet, on regarde si il rentre dans un des sacs créés: si oui, on le met dans le premier qui convient; sinon, on crée un nouveau sac et on y met l'objet
 - Heuristique 3 (BestFit) : Pour chaque objet, si il rentre dans un des sacs créés, on le met dans celui qui est le plus rempli parmi ceux qui conviennent. Sinon, on crée un nouveau sac et on y met l'objet
 - Heuristique 4 (WorstFit) : Pour chaque objet, si il rentre dans un des sacs créés, on le met dans celui qui est le moins rempli. Sinon, on crée un nouveau sac et on y met l'objet
 - Chaque heuristique peut se décliner en deux version :
 - « on-line » : on range les objets au fur et à mesure de leur arrivée
 - « off-line » : on peut alors trier les objets par poids décroissants avant de les ranger

Exemple #2: TSP

- Donnée:

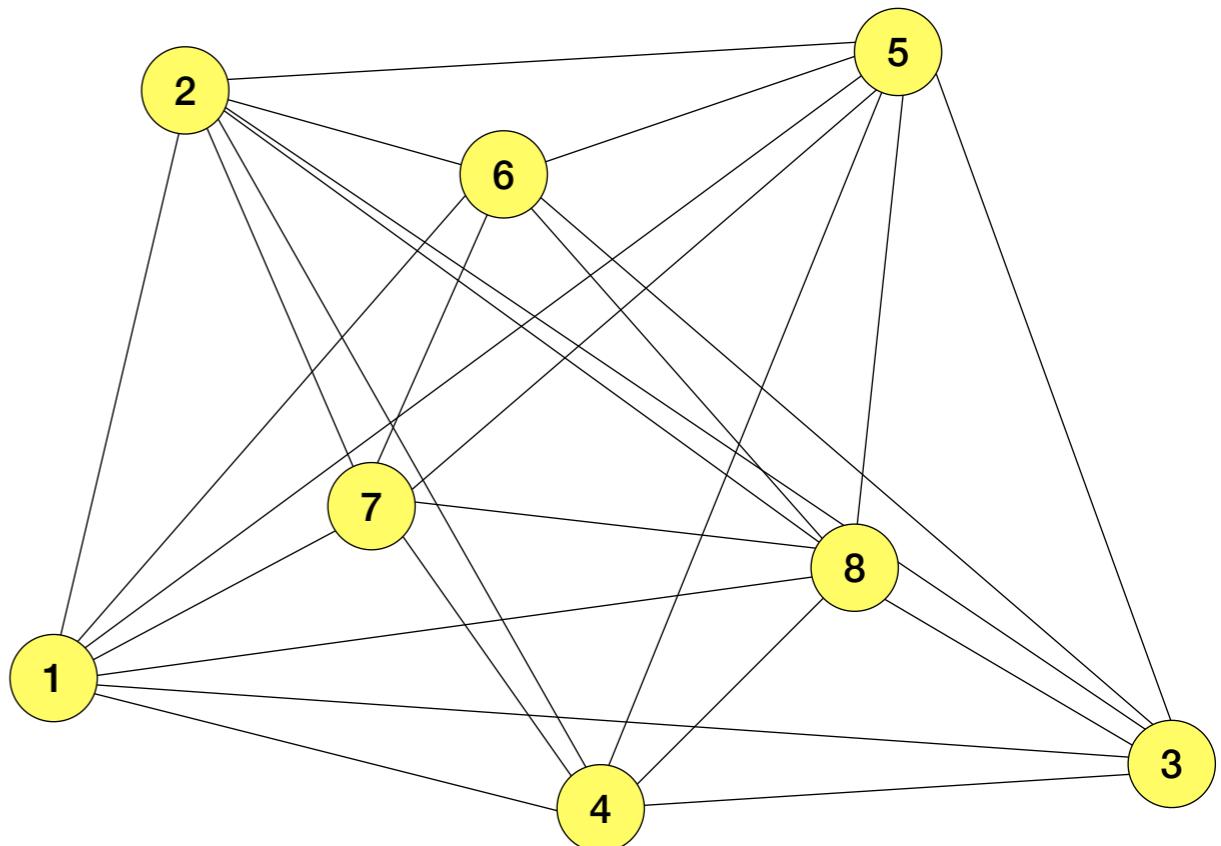
- Un graphe $G=(V,A,\omega)$ à **n** sommets
- $\omega : A \rightarrow \text{IR}$ poids sur chaque arête
- Le plus souvent, on considère un graphe complet (et ω symétrique, géométrique, etc)

- Problème:

- Trouver un cycle Hamiltonian de poids minimal

- une tournée de poids minimale (qui passe par chaque sommet une et une seule fois)

- **TSP est NP-dur**



Exemple #2: TSP

Énumération – Force Brute

- Énumérer les tournées possible ?
 - $\sim (n-1) \cdot (n-2) \cdot \dots \cdot 3 \cdot 2 \cdot 1 = (n-1)!$
 - Une tournée est une **permutation** des sommets
 - Clairement Intractable !

Nombre de villes	Nombre de tours possible	Temps d'énumération
3	1	1 msec
4	3	3 msec
5	12	12 msec
6	60	60 msec
7	360	360 msec
8	2,520	2,5 sec
9	20,160	20 sec
10	181,440	3 mn
11	1,814,400	0.5 h
12	19,958,400	5.5 h
13	239,500,800	2.8 jrs
14	3,113,510,400	36 jrs
15	43,589,145,600	1.3 jrs
16	653,837,184,000	20 ans

Exemple #2: TSP programmation dynamique

- $S = \{1, 2, \dots, n\}$: les villes
 - On note $DIST[i, j]$ la distance entre la ville i et la ville j
 - On peut commencer par n'importe quelle ville et on y retourne !
 - On commence par la ville 1
 - On visite une ville $i \in \{2, 3, \dots, n\}$
 - On recommence avec les autres villes $\{2, 3, \dots, i-1, i+1, \dots\}$ de façon récursive !
 - **Comment choisir la ville suivante i ?**

Exemple #2: TSP programmation dynamique

- Soit i une ville et A un ensemble de ville
- Soit $L(i,A)$ la longueur d'un **plus court chemin** qui :
 - (1) commence en i , (2) visite les villes de A exactement une fois, et (3) se termine à la ville 1
- Propriété:
 - $A = \emptyset \rightarrow L(i,A) = DIST[i,1]$
 - $A \subseteq S \setminus \{1,i\} \rightarrow L(i,A) = \min\{DIST[i,j] + L(j,A \setminus \{j\}) \mid j \in A\}$
- **$L(1,\{2,\dots,n\})$ est la longueur d'une tournée optimale !**

Exemple #2: TSP programmation dynamique

- On commence par $A = \emptyset$ et on l'augmente itérativement...

```
for i = 1 to n do L(i,  $\emptyset$ ) = DIST[i,1] endfor
for k = 1 to n-1 do
    forall A  $\subseteq$  S \ {1} with |A| = k do
        for i = 1 to n do
            if i  $\notin$  A then
                L(i,A) = min{DIST[i,j] + L(j,A\{j}) | j  $\in$  A}
            endif
        endfor
    endforall
endfor
return L(1,{2,...,n})
```

Exemple #2: TSP

programmation dynamique

Nombre de villes	Force brute		Programmation dynamique	
	Nombre de tours possible $(n-1)!/2$	Temps	Taille des tables L (i,A) n^{22^n}	Temps
3	1	1 msec	72	72 msec
4	3	3 msec	256	0.4 sec
5	12	12 msec	800	0.8 sec
6	60	60 msec	2304	2.3 sec
7	360	360 msec	6272	6.3 sec
8	2,520	2,5 sec	16,384	16 sec
9	20,160	20 sec	41,472	41 sec
10	181,440	3 mn	102,400	102 sec
11	1,814,400	0.5 h	247,808	4.1 mn
12	19,958,400	5.5 h	589,824	9.8 mn
13	239,500,800	2.8 jrs	1,384,448	23 mn
14	3,113,510,400	36 jrs	3,211,264	54 mn
15	43,589,145,600	1.3 jrs	7,372,800	2 h
16	653,837,184,000	20 ans	16,777,216	4,7 h

Exemple #2: TSP

Énumération – ‘intelligente’...

- Branch & ?

Exemple #2: TSP

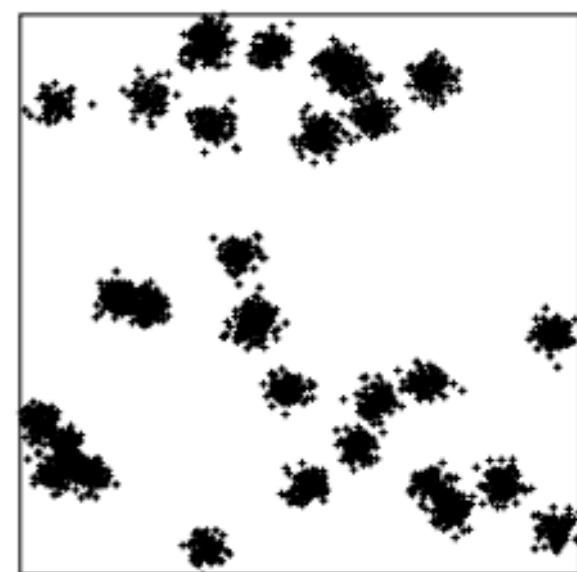
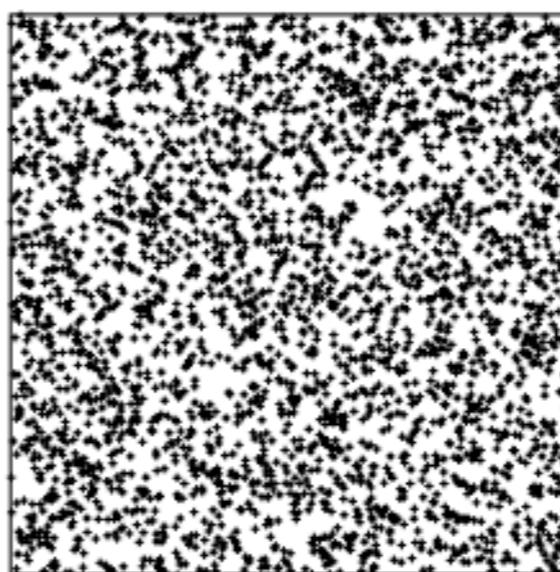
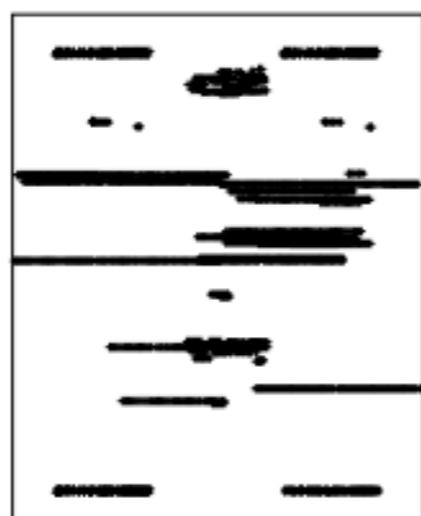
Énumération – ‘intelligente’...

- **Branch & cut : TSP concorde solver :**

- (LP-)relaxation pour les bornes inf ; cutting planes + branchement
- Heuristiques pour les bornes sup

<http://www.math.uwaterloo.ca/tsp/concorde>

Instance	# noeuds de calcul parallèle	Temps CPU
att532	7	109 sec
Rat783	1	37 sec
Pcb1173	19	468 sec
Fl1577	7	6705 sec
d2105	169	3108 h
Pr2392	1	116 sec
rl5934	205	163 h
usa13509	9539	4 ans
d15112	164569	22 ans
s24978	167263	84 ans



Exemple #2: TSP

Heuristiques – Constructives

- **Basées sur le pré-calcul (polynomial) de structures d'arbre**
 - Arbre de poids minimum, Christofides' heuristics, Fast recursive partitioning heuristic
- **Basées sur la construction incrémentale de fragments**
 - Nearest Neighbor, Double-Ended Nearest Neighbor, Multiple Fragment
- **Basées sur la construction incrémentale de tournées partielles**
 - Nearest Addition; Farthest Addition ; Random Addition ; Nearest Insertion ; Farthest Insertion ; Random Insertion

Exemple #2: TSP

Heuristiques – Constructives

- **Basées sur le pré-calcul (polynomial) de structures d'arbre**
 - Arbre de poids minimum, Christofides' heuristics, Fast recursive partitioning heuristic
- **Basées sur la construction incrémentale de fragments**
 - Nearest Neighbor, Double-Ended Nearest Neighbor, Multiple Fragment
- **Basées sur la construction incrémentale de tournées partielles**
 - Nearest Addition; Farthest Addition ; Random Addition ; Nearest Insertion ; Farthest Insertion ; Random Insertion

Exemple #2: TSP

Heuristiques – arbre !

- Soit T_{\min} une tournée de poids/longueur minimale

- Enlever une arête arbitraire $e \in T_{\min}$ de poids $p(e)$
- Soit $C = T_{\min} \setminus \{e\}$
 - C est un arbre : $p(C) = p(T_{\min}) - p(e) < p(T_{\min})$

- Soit A_{\min} un arbre recouvrant de point minimum

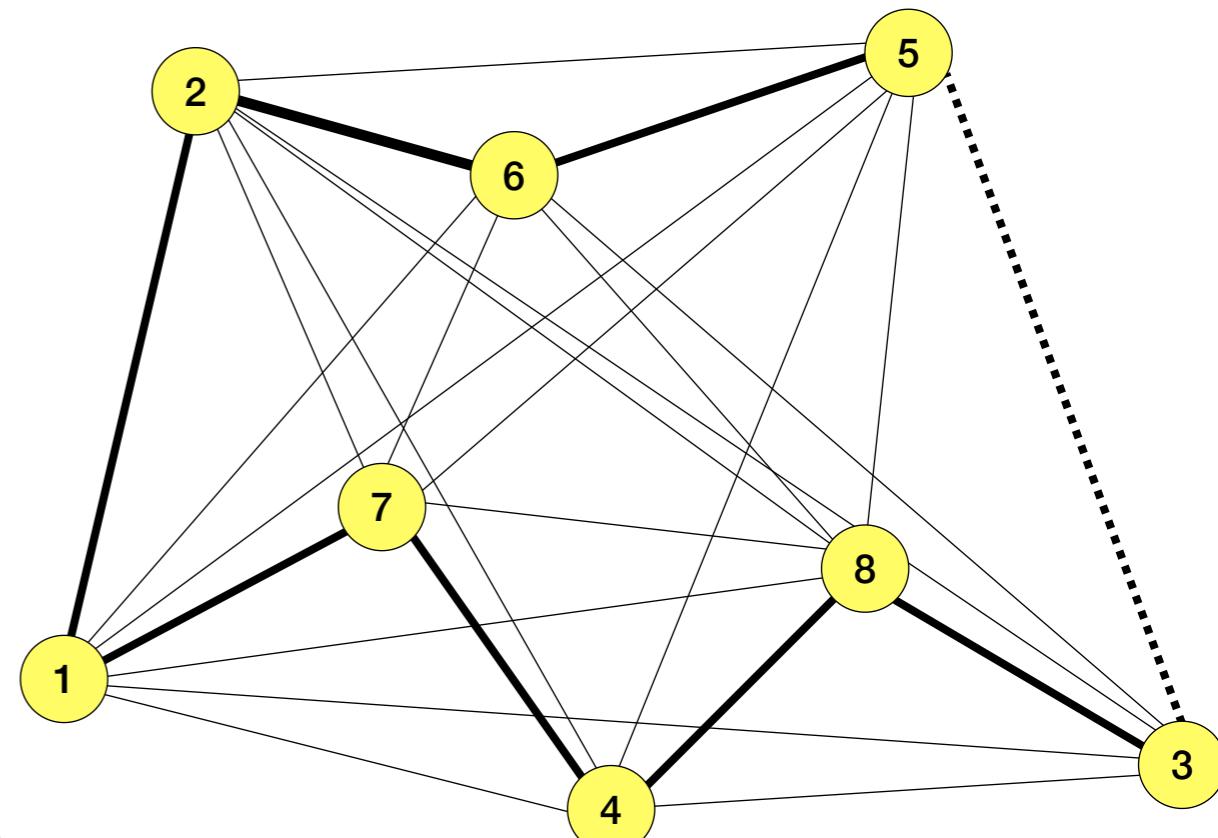
- On a donc : $p(A_{\min}) \leq p(C) < p(T_{\min})$

- Idée d'un algorithme constructif avec pré-calcul (?)

- Pré-calculer un arbre recouvrant de poids minimum A_{\min}

- Le problème du calcul d'arbre recouvrant de poids minimum est dans P ! Algorithme glouton en $O(n^2)$ (Exercice?)

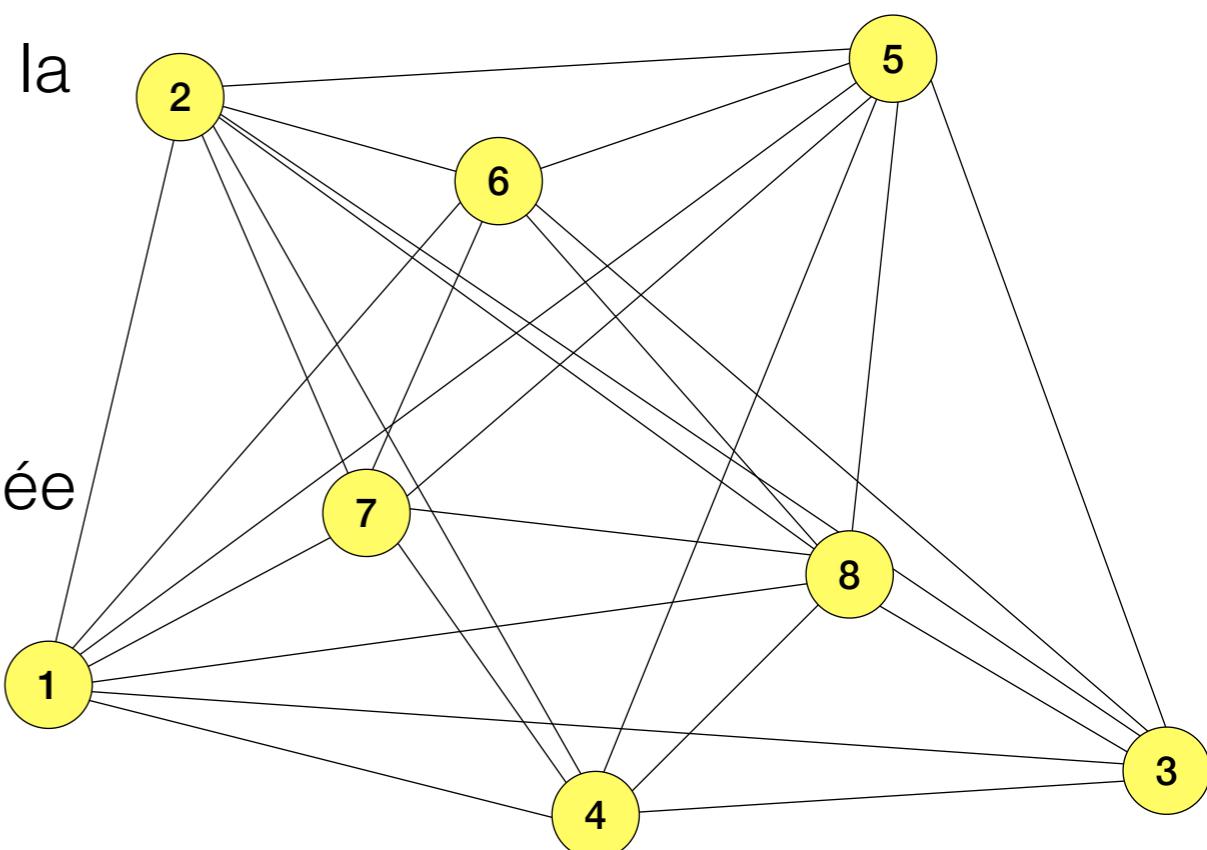
- Construire une tournée de façon gloutonne en corigeant l'arbre pour avoir un cycle Hamiltonien !



Exemple #2: TSP

Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum
- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours
- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$
- Corriger (de façon gloutonne) L par une tournée L' valide !
 - soit $a \rightarrow b \rightarrow c$ trois sommets successifs
 - les remplacer par $a \rightarrow c$ si b est non isolé !
- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**



Exemple #2: TSP

Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

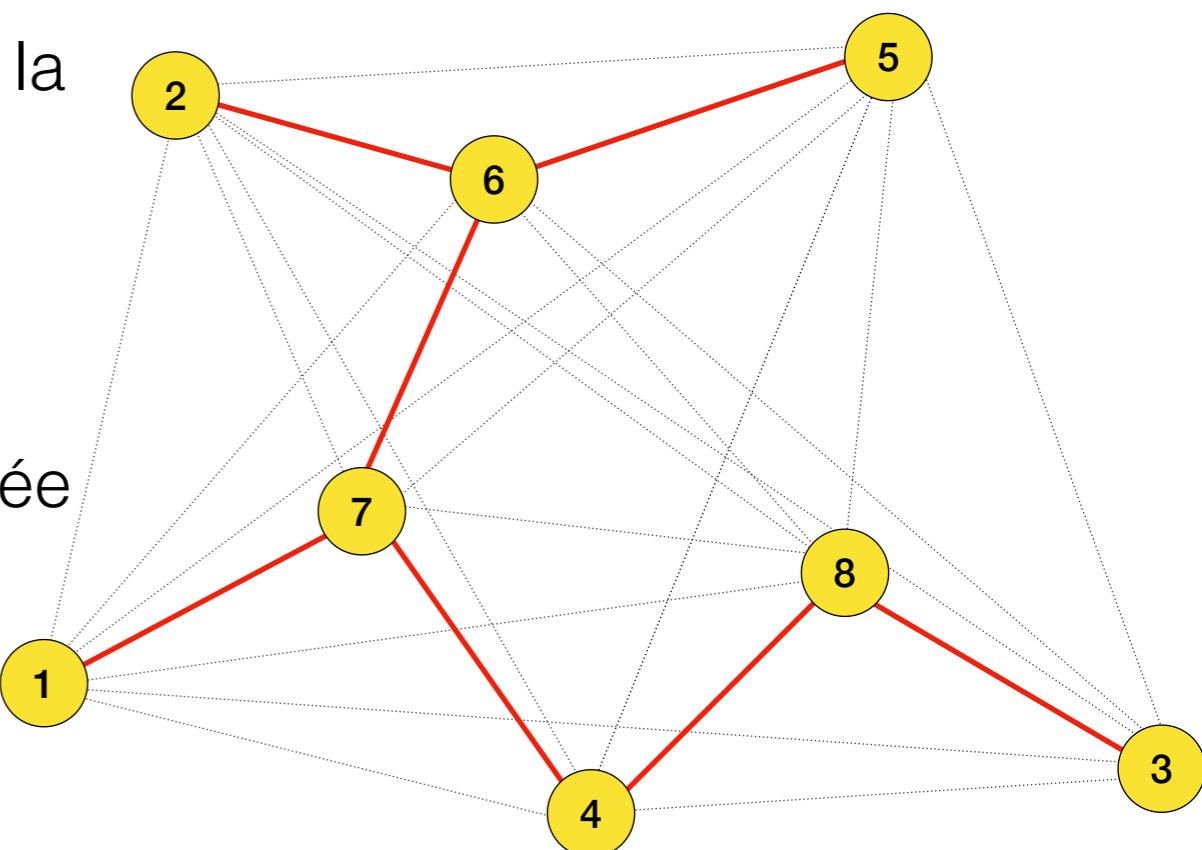
- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- Corriger (de façon gloutonne) L par une tournée L' valide !

- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**



Exemple #2: TSP

Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

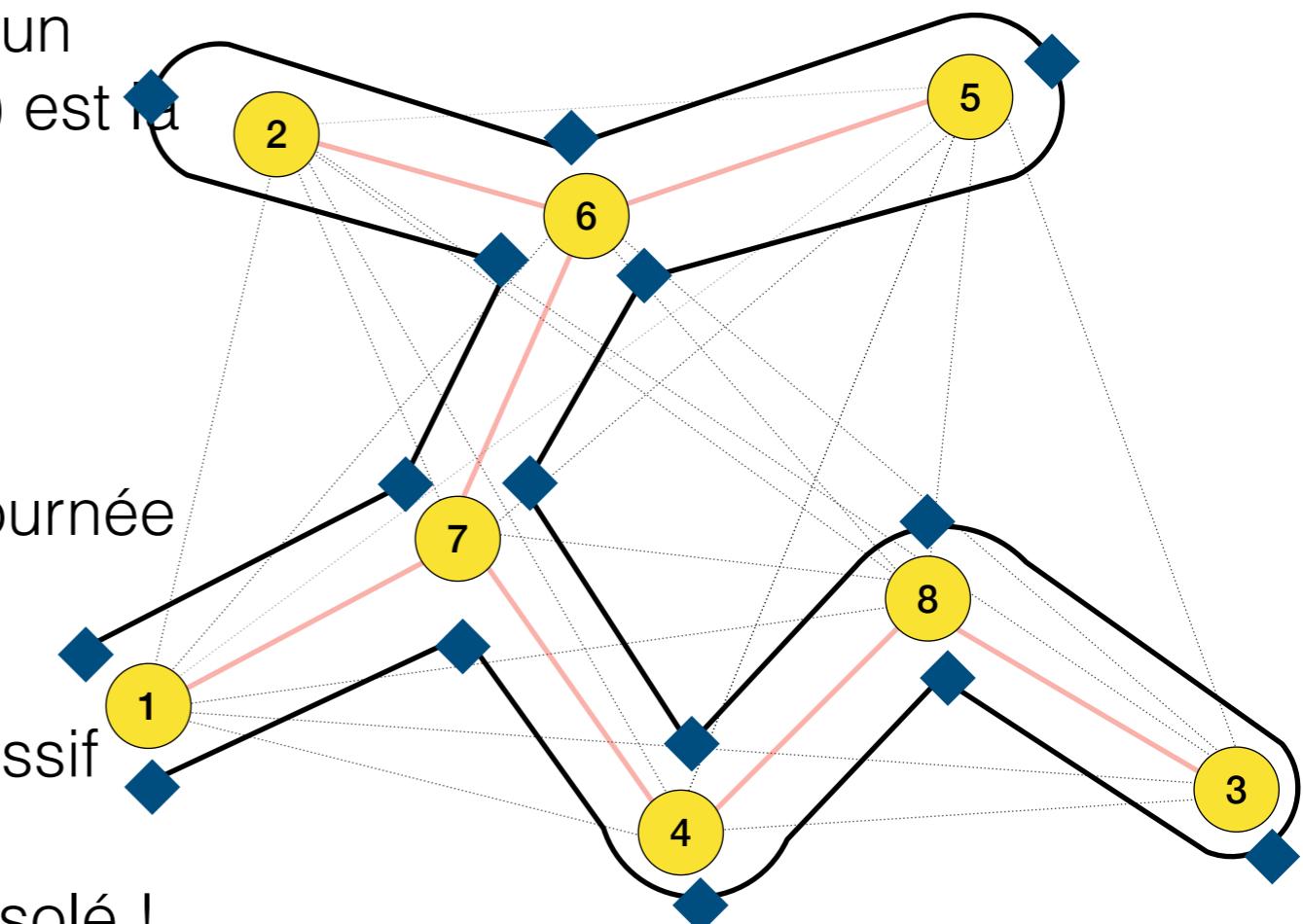
- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- Corriger (de façon gloutonne) L par une tournée L' valide !

- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**



Exemple #2: TSP

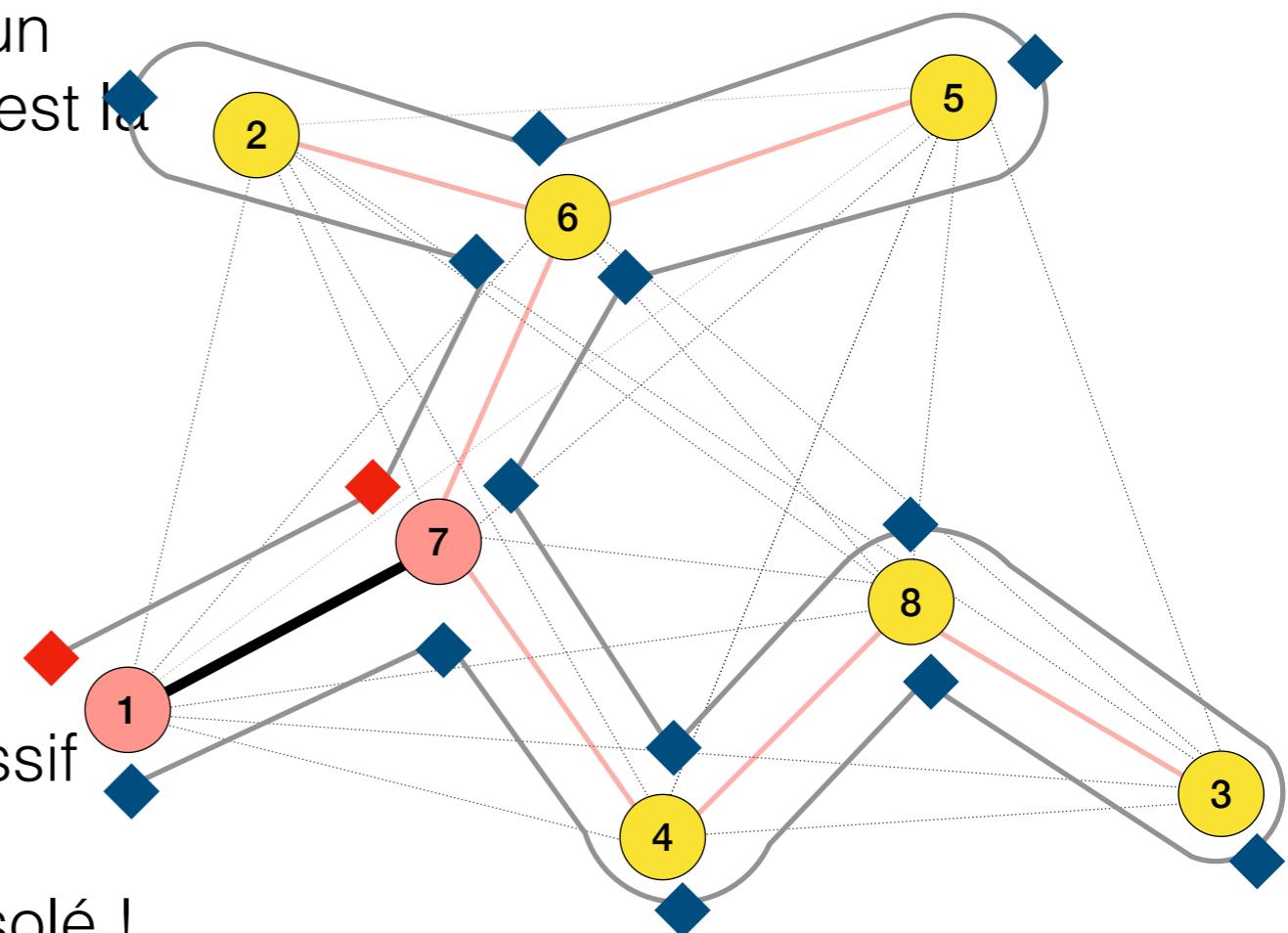
Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- **Corriger (de façon gloutonne)** L par une tournée L' valide !



- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**

Exemple #2: TSP

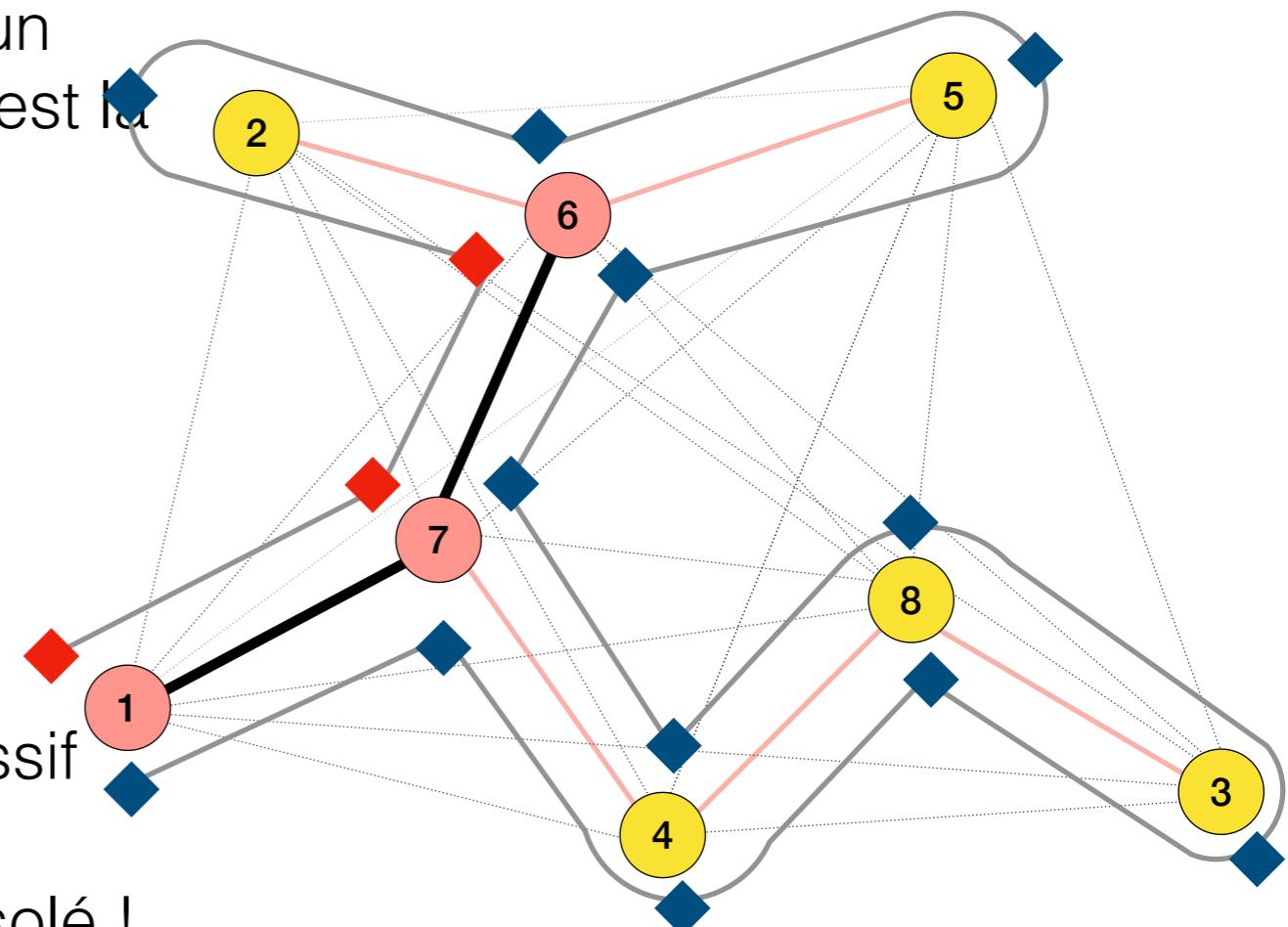
Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- **Corriger (de façon gloutonne)** L par une tournée L' valide !



- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**

Exemple #2: TSP

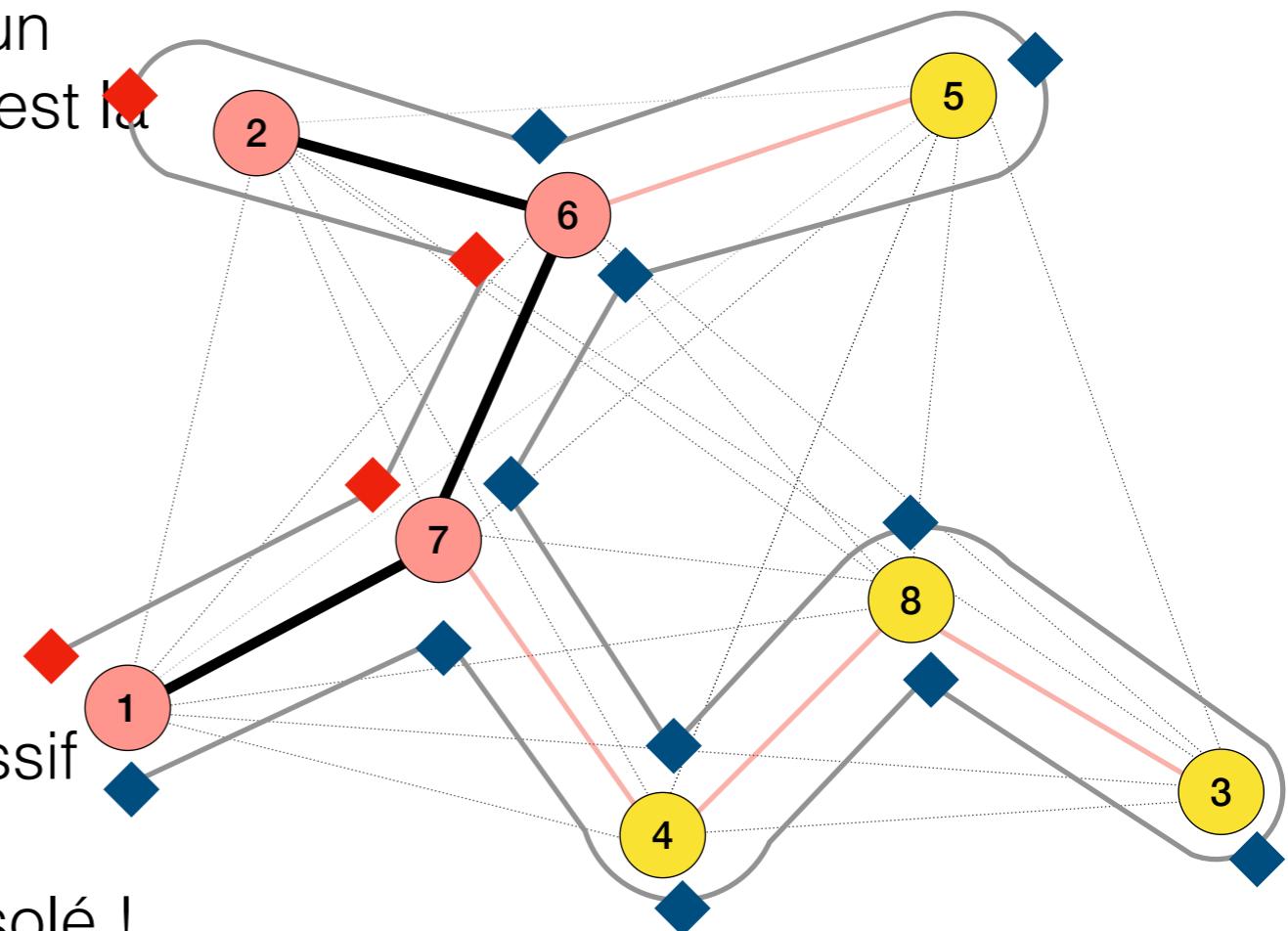
Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- **Corriger (de façon gloutonne)** L par une tournée L' valide !



- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**

Exemple #2: TSP

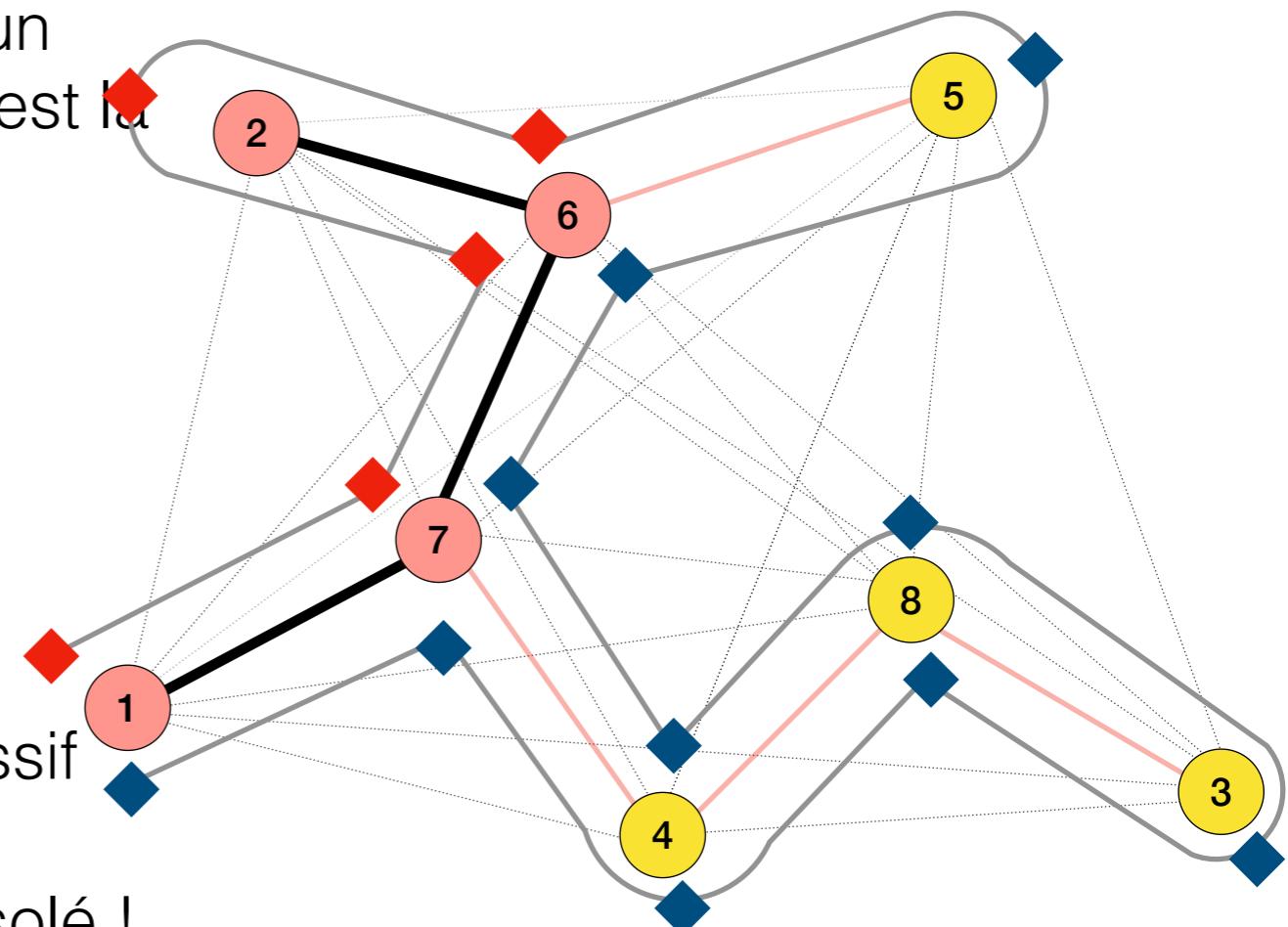
Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- **Corriger (de façon gloutonne)** L par une tournée L' valide !



- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**

Exemple #2: TSP

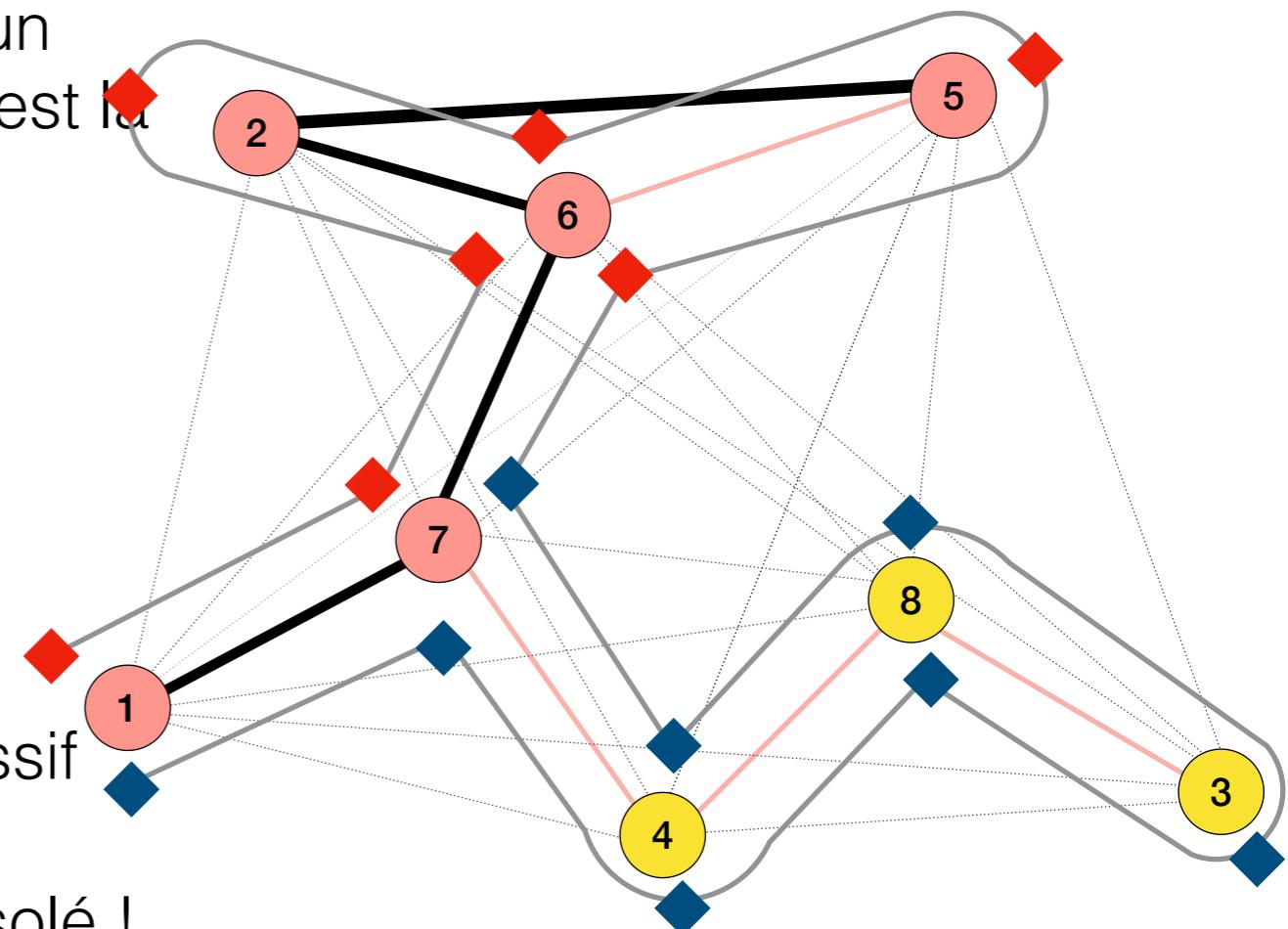
Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- **Corriger (de façon gloutonne)** L par une tournée L' valide !



- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**

Exemple #2: TSP

Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

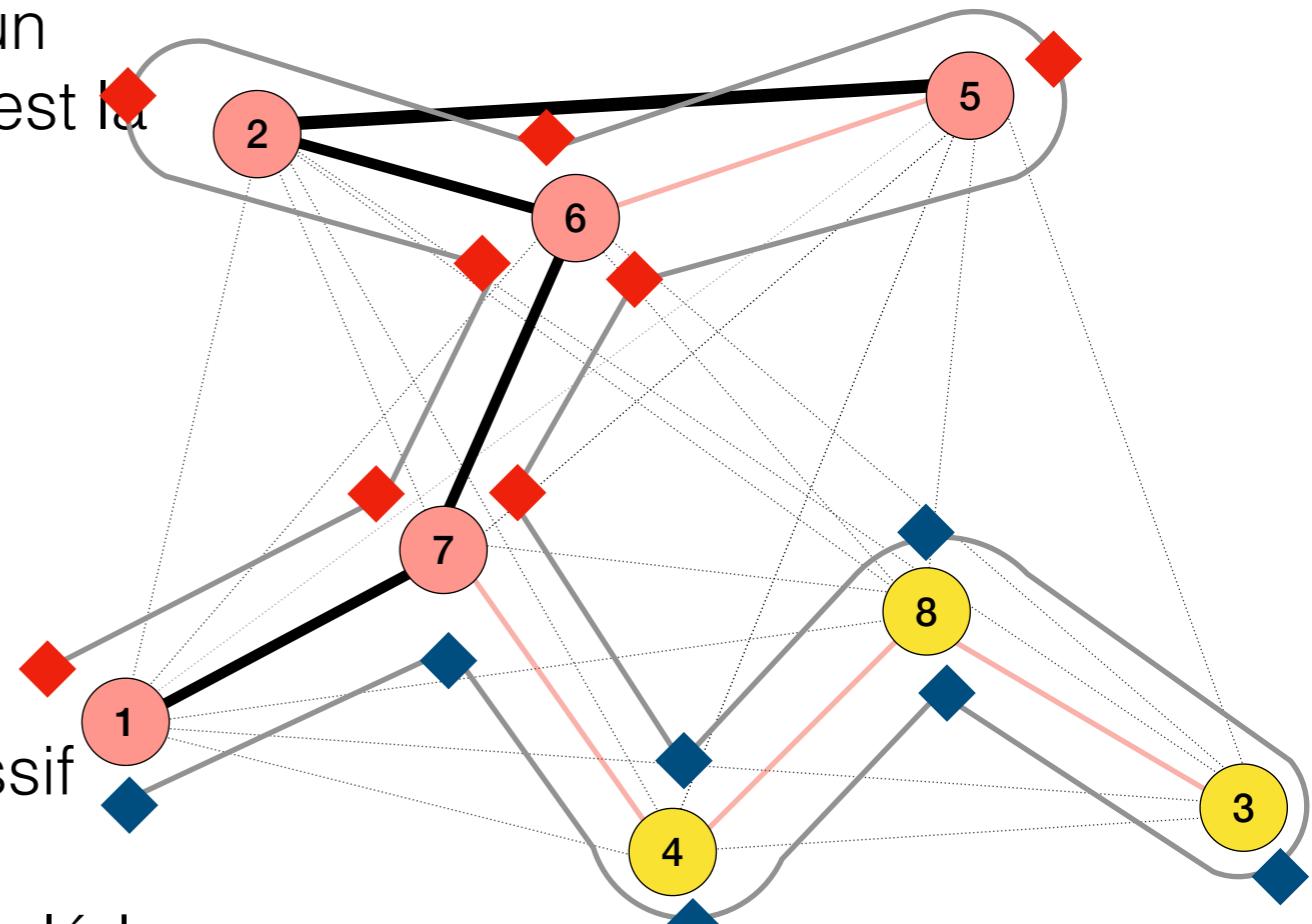
- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- **Corriger (de façon gloutonne)** L par une tournée L' valide !

- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**



Exemple #2: TSP

Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

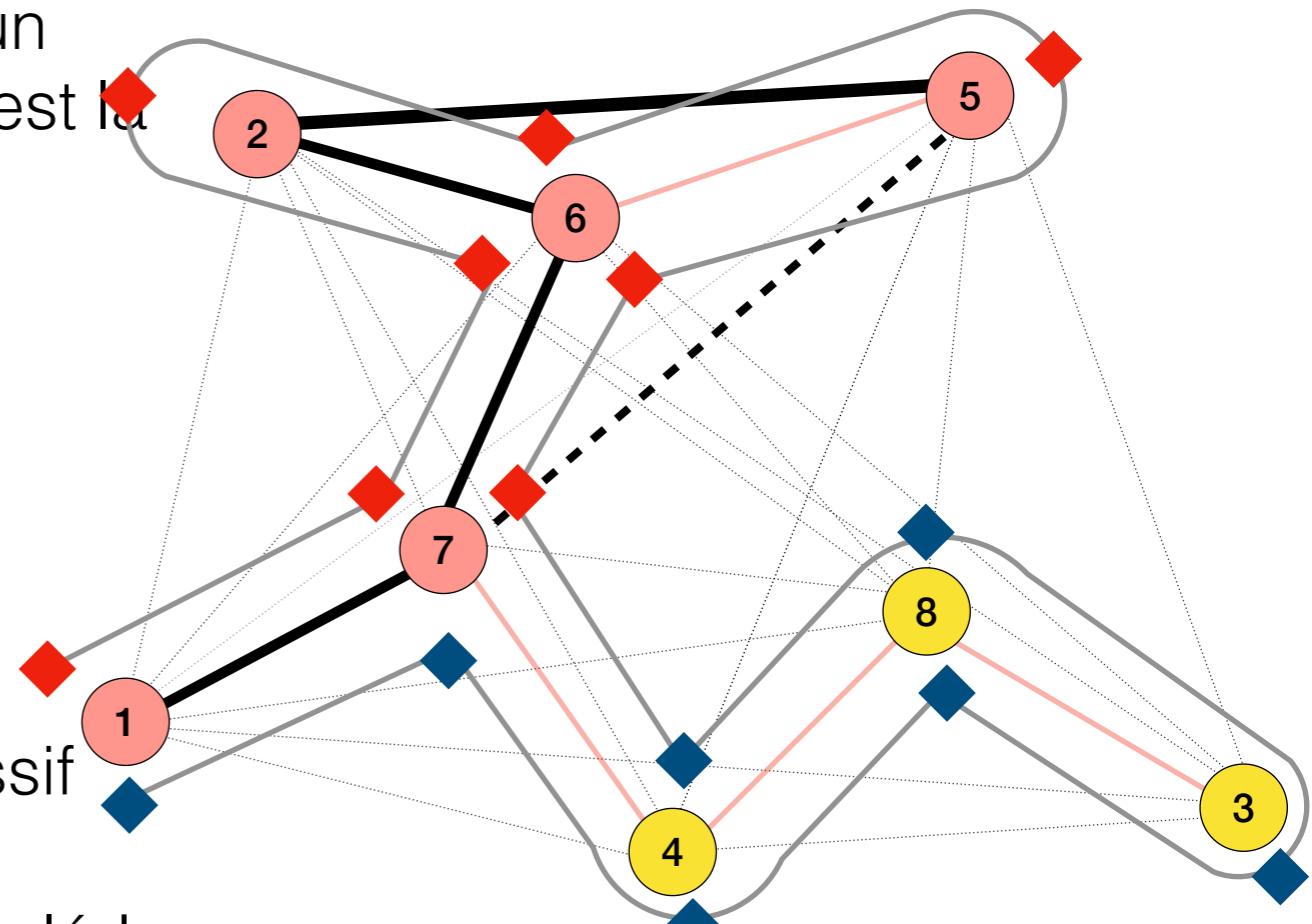
- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- **Corriger (de façon gloutonne)** L par une tournée L' valide !

- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**



Exemple #2: TSP

Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

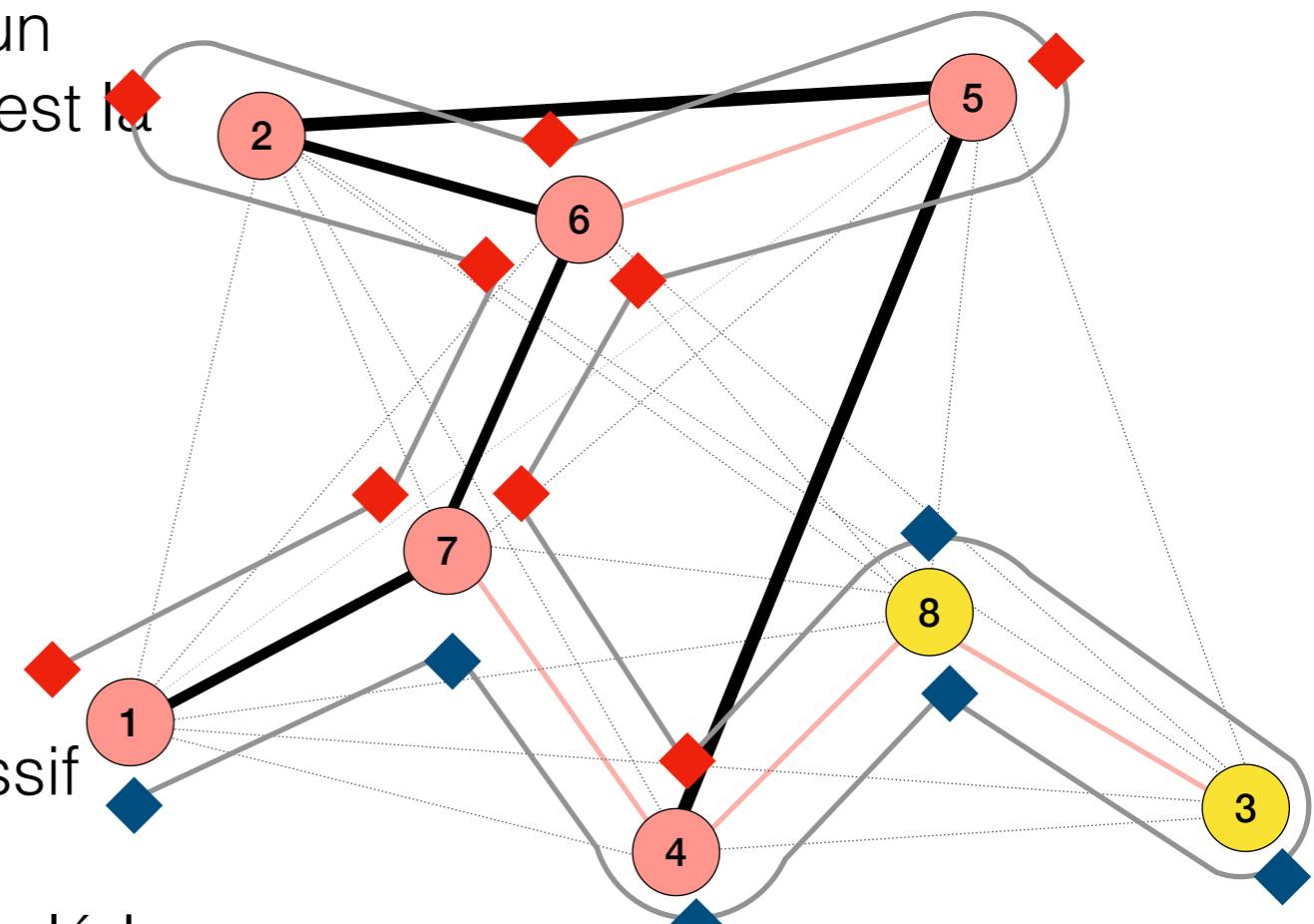
- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- **Corriger (de façon gloutonne)** L par une tournée L' valide !

- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**



Exemple #2: TSP

Heuristiques – arbre !

- Soit A_{\min} un arbre recouvrant de poids minimum

- Traverser A_{\min} en profondeur et construire un chemin $L=\{1,2,\dots,2n-1\}$ où pour tout i , $v(i)$ est la ville visitée à l'étape i du parcours

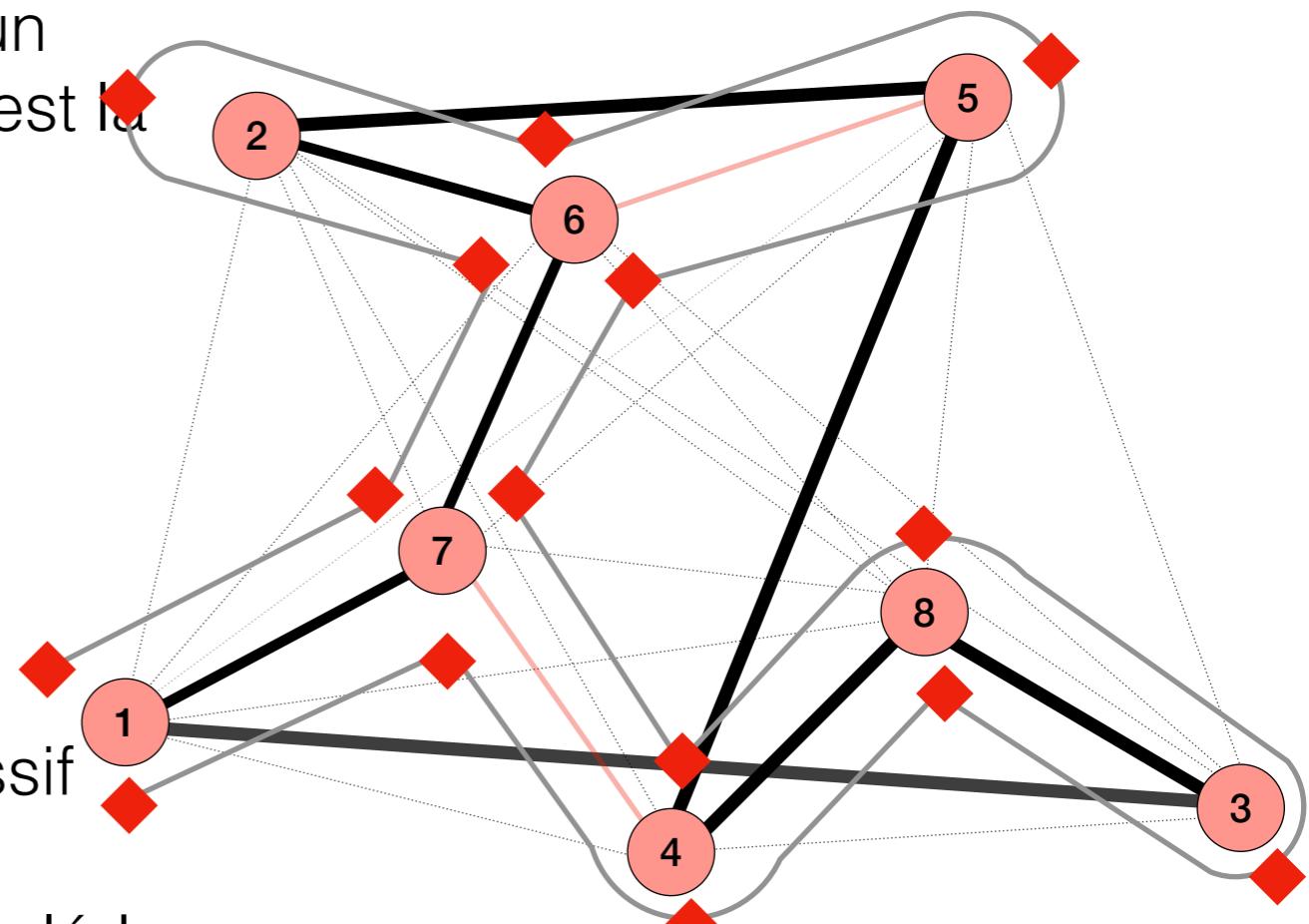
- $p(L) = 2 p(A_{\min}) < 2 p(T_{\min})$

- **Corriger (de façon gloutonne)** L par une tournée L' valide !

- soit $a \rightarrow b \rightarrow c$ trois sommets successifs

- les remplacer par $a \rightarrow c$ si b est non isolé !

- **Par l'inégalité triangulaire, $p(L') \leq p(L) < 2 p(T_{\min})$**



Exemple #2: TSP

Heuristiques – fragment !

- **NN (Nearest Neighbor) heuristic**

Illustration (tableau)

- Commencer avec un sommet initial (aléatoirement)
- À chaque étape, prendre l'arête de poids minimum vers un sommet non encore visité
- Étendre (v_1, \dots, v_k) avec un sommet u non visité tel que $d(v_k, u)$ soit minimale
- Compléter pour obtenir un cycle hamiltonien (fermer la tournée)

Exemple #2: TSP Heuristiques – fragment !

- **NN (Nearest Neighbor) heuristic**

- Commencer avec un sommet initial (aléatoirement)
- À chaque étape, prendre l'arête de poids minimum vers un sommet non encore visité
- Étendre (v_1, \dots, v_k) avec un sommet u non visité tel que $d(v_k, u)$ soit minimale
- Compléter pour obtenir un cycle hamiltonien (fermer la tournée)

- Pour les instances de taille n vérifiant l'inégalité triangulaire, NN a un facteur de garantie $1/2 \cdot (\log(n) + 1)$

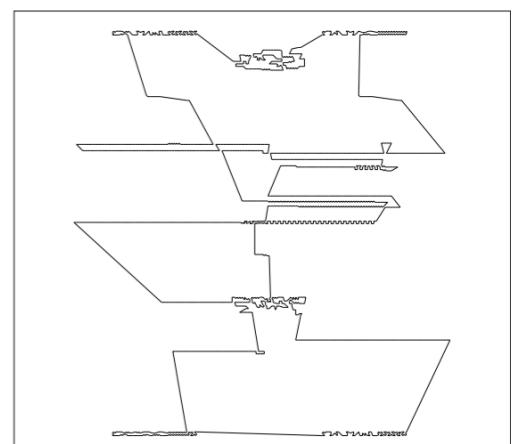
- En pratique (instances de la TSPLIB), les tournées NN sont typiquement 20–35% plus mauvaises que l'optimale

- Typiquement, les tournées NN sont proches de l'optimale mais contiennent quelques arêtes longues

pcb1173



fl1577



Exemple #2: TSP

Heuristiques – insertion !

- **Itérativement étendre une tournée (partielle) en insérant un sommet choisi de façon heuristique tel que la longueur de la tournée augmente le moins possible**
- Plusieurs heuristiques pour choisir le sommet u à insérer dans la tournée partielle p
 - Nearest insertion (u : le plus proche à un sommet de p)
 - Farthest insertion (u : celui dont la distance à un sommet de p est maximale)
 - Cheapest insertion (u : celui qui augmente le moins possible la longueur de p)
 - random insertion (u : aléatoire)
- Nearest / cheapest insertion → **facteur de garantie 2** pour les instances TSP vérifiant l'inégalité triangulaires
- En pratique, farthest / random insertion est plus efficace;
 - Typiquement, qualité 13–15% au delà de l'optimale pour les instances TSPLIB

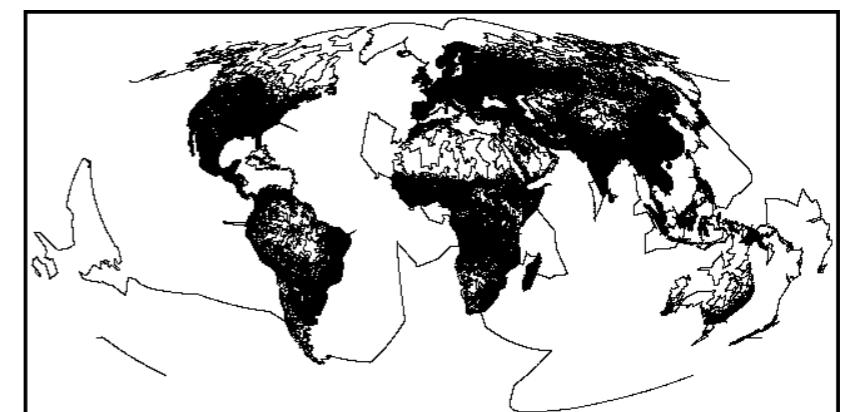
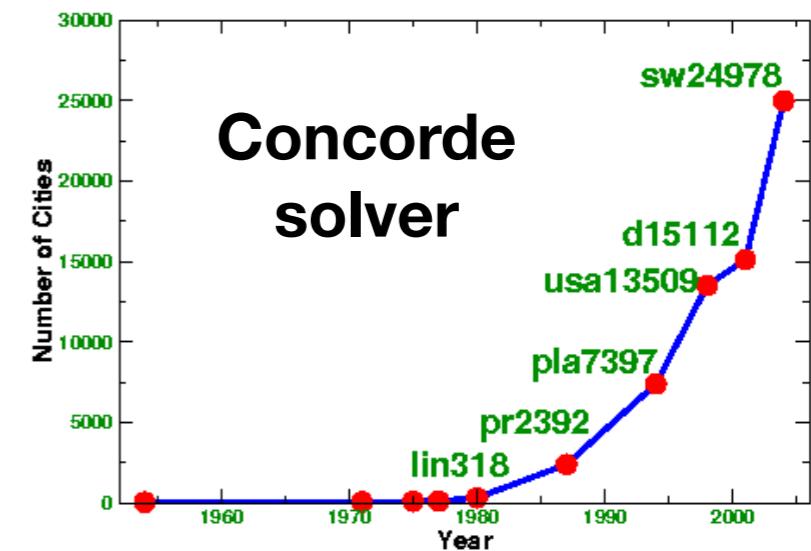
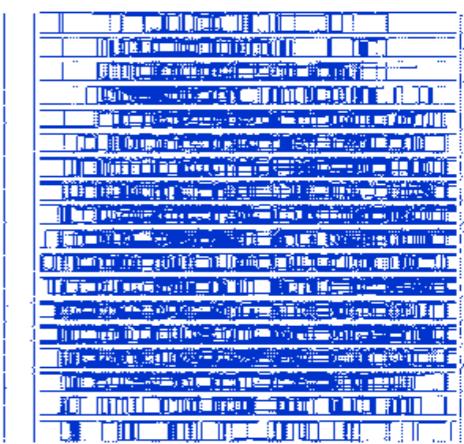
Exemple #2: TSP

- Autres algorithmes puissants (heuristiques et exactes) sur des instances de plus en plus grandes:
 - TSPlib:
 - $n \leq 1000 \rightarrow \sim 1\text{mn}$
 - $1000 \leq n \leq 2392 \rightarrow \text{quelques secondes à 1 h}$
 - Plus grandes instances résolues par concorde (exacte):
 - Sweden : $n = 24,978$
 - VLCI : $n = 85,900 \rightarrow 286.2 \text{ CPU jours !}$
 - World TSP : $n = 1,904,711$ villes !
 - Non-résolue (encore) de façon exacte
 - Meilleur record (Juin 2020) détenue par une heuristique

Sw24978



pla85900



Problème NP-dur: que faire ?

Bilan

- **Algorithmes exactes (programmation dynamique, énumération implicite intelligente...)**
 - Petite taille, structures d'instances / propriétés particulières, etc
 - Attention à la complexité ! Temps de calcul/mémoire ...
- **Algorithmes d'approximations (heuristiques constructives)**
 - Souvent simples et efficaces (linéaires, ou presque)
 - Le ratio de garantie ... garantit une certaine qualité mais n'est qu'une indication correspondant au pire des cas!
 - Certains problèmes NP-durs n'ont pas d'algorithmes polynomiaux d'approximation avec garantie, sauf si $P=NP$
 - Heuristiques (génériques) sans garanties ... (à suivre aux prochains cours)