# Méthodes Approchées
# Résolution de Problèmes d'Ordonnancement
# 4-IR

Le but des TP est de minimiser la durée totale d'un problème d'ordonnancement (de type Job-Shop). Le problème de Job-Shop a été présenté durant les séances de cours. Pour résoudre ce problème d'optimisation combinatoire, différentes méthodes approchées seront progressivement développées et testées. Sur la base d'un ensemble de jeux de données, une campagne d'évaluations expérimentales permettra de mettre en évidence les intérêts et limites des différentes méthodes.

MJ. Huguet - A. Bit-Monnot

Année Universitaire 2020-2021

# 1 Organisation des séances

L'organisation des TP est la suivante :

— Il y a 5 séances de TP et une dose raisonnable de travail personnel, le travail sera réalisé en monôme

— Page moodle : `https://moodle.insa-toulouse.fr/course/view.php?id=1354`

— Document de suivi d'avancement à remplir : `https://docs.google.com/spreadsheets/d/1usvMdsXt59wlqAidptBxxemqP3sTWBsp1sPdHP2U6Zo/edit?usp=sharing`

— Utilisez le serveur discord des 4-IR pour poser des questions (ou répondre à vos collègues)

— L'évaluation sera effectué sur la base d'un rendu individuel (à préciser sur la page moodle).

Pour développer les méthodes proposées, vous allez partir d'une base de code en Java. Des jeux de données sont fournies avec le code. Vous pouvez aussi consulter le site `http://jobshop.jjvh.nl/` pour des références bibliographiques sur le Job-Shop et des résultats sur les meilleures solutions connues (bornes inférieures, bornes supérieures, optimum, visualisation de solutions).

# 2 Découverte du code

## 2.1 Mise en place

L'objectif de cette étape est de récupérer le code existant, de vérifier que vous pouvez le compiler et de prendre connaissance de son organisation.

**A faire** **(1) Inscrivez-vous sur la page moodle** du cours si ce n'est pas déjà fait. **(2) Rendez vous sur la documentation et suivez les instructions** de la page d'accueil : `https://insa-4ir-meta-heuristiques.github.io/doc/`.

À la fin de cette étape, vous devriez avoir :

— votre propre dépôt git avec la base de code (créé par Github classroom).

— importé le projet dans IntelliJ (ou un autre IDE)

— testé que le programme compile et tourne sans problème

## 2.2 Exemple de problème de Job-Shop

Vous trouverez dans le dossier `instances/` un ensemble de jeux de données communément utilisées pour le problème de Job-Shop. Si l'on considère l'instance vue dans les exercices de cours, constituée de quatre jobs avec trois tâches chacun et utilisant 3 machines $(r_1, r_2, r_3)$ :

| | | | |
|---|---|---|---|
| $J_1$ | $r_1, 1$ | $r_2, 3$ | $r_3, 2$ |
| $J_2$ | $r_2, 8$ | $r_1, 5$ | $r_3, 10$ |
| $J_3$ | $r_1, 5$ | $r_3, 4$ | $r_2, 8$ |
| $J_4$ | $r_3, 4$ | $r_1, 10$ | $r_2, 6$ |

L'instance décrite ci-dessus est nommée `aaa2`, elle est donnée dans le fichier `instances/aaa2` :

```
# Fichier instances/aaa2
4 3 # 4 jobs and 3 tasks per job
0 1 1 3 2 2  # Job 1 : (machine duration) for each task
1 8 0 5 2 10 # Job 2 : (machine duration) for each task
0 5 2 4 1 8  # Job 3 : (machine duration) for each task
2 4 0 10 1 6 # Job 4 : (machine duration) for each task
```

La première ligne donne le nombre de jobs et le nombre de tâches par jobs. Le nombre de machines est égal au nombre de tâches. Chacune des lignes suivantes spécifie la machine (ici numérotées `0`, `1` et `2`) et la durée de chaque tâche. Par exemple la ligne `0 1 1 3 2 2` spécifie que pour le premier job :

— `0 1` : la première tâche s'exécute sur la machine `0` et dure `1` unité de temps

— `1 3` : la deuxième tâche s'exécute sur la machine `1` et dure `3` unités de temps

— `2 2` : la troisième tâche s'exécute sur la machine `2` et dure `2` unités de temps

### 2.3 Architecture du code fourni

**À faire.** Lire la documentation du projet, disponible ici : `https://insa-4ir-meta-heuristiques.github.io/doc/`

### 2.4 Manipulation des représentations

On va chercher ici à se familiariser avec la base de code fournie et et à prendre en main les différentes représentations de solution.

**À faire.** Ouvrez la méthode `MainTest.main()`.

— Pour la solution (proposée dans le code) de l'instance `aaa1` basée sur la représentation par numéro de job, calculez (à la main) les dates de début de chaque tâche

— Exécutez la méthode `MainTest.main()` et vérifiez que la solution affichée correspond bien à vos calculs.

— Construisez un `Schedule` représentant la même solution. On partira d'un `Schedule` sur lequel on spécifiera les dates de début de chaque tâche. Visualisez le diagramme de Gantt de cette solution (Gantt en mode texte).

— Reconstruisez à la main la même solution dans la représentation par ordre de passage sur les ressources.

— Modifiez la solution construite avec `ResourceOrder` pour obtenir la solution optimale (on sait qu'elle a un makespan de 11). Visualisez le diagramme de Gantt associé.

— Trouvez une représentation par ordre de passage sur les machines qui soit invalide, c'est à dire qui ne puisse pas être transformée en `Schedule`. On cherchera pour cela à créer des contraintes contradictoires entre l'ordre imposé entre les tâches d'un même job et l'ordre de passage sur les machines.

# 3 Heuristiques gloutonnes

L'objectif de cette partie est d'implémenter plusieurs méthodes gloutonnes pour calculer la durée totale d'un problème de Job-Shop en considérant qu'une solution est représentée par l'ordre de passage des tâches sur les machines (`ResourceOrder`).

## 3.1 Premières heuristiques

Pour le problème de jobshop, le principe général d'une méthode gloutonne est le suivant :

— **Initialisation** : Déterminer l'ensemble des tâches réalisables (initialement, les premières tâches de tous les jobs)

— **Boucle** : tant qu'il y a des tâches réalisables

1. Choisir une tâche dans cet ensemble et placer cette tâche sur la ressource qu'elle demande (à la première place libre dans la représentation par ordre de passage)
2. Mettre à jour l'ensemble des tâches réalisables

On dit qu'une tâche est *réalisable* si tous ses prédécesseurs ont été traités (précédences liées aux jobs). Il faut noter que chaque tâche doit apparaître exactement une fois dans l'ensemble des tâches réalisables et que toutes les tâches doivent être traitées.

Un paramètre déterminant dans la construction d'une heuristique gloutonne est la manière dont est choisie la prochaine tâche à exécuter parmi toutes les tâches réalisables. Cette décision est prise à partir de règles qui permettent de donner une priorité plus élevée à certaines tâches. Pour le JobShop, on peut notamment considérer les règles de priorité suivantes :

— SPT (Shortest Processing Time) : donne priorité à la tâche la plus courte ;

— LPT (Longest Processing Time) : donne priorité à la tâche la plus longue ;

— SRPT (Shortest Remaining Processing Time) : donne la priorité à la tâche appartenant au job ayant la plus petite durée restante ;

— LRPT (Longest Remaining Processing Time) : donne la priorité à la tâche appartenant au job ayant la plus grande durée

**A faire**

— Implémentation. Complétez la classe `GreedySolver` implémentant une recherche gloutonne basée sur `ResourceOrder` pour les priorités SPT et LRPT.

— Validation. Pour valider votre implémentation, vous pouvez utiliser l'instance `aaa3` qui produit des résultats déterministes pour chacune des priorités proposées. Vous trouverez dans les commentaires du fichier `instances/aaa3`, les makespans attendus pour chacune des priorités considérées.

— Evaluation. Testez et évaluez ces heuristiques sur les instances $ft06, ft10, ft20$ et sur les instances $la01, \ldots, la40$. Pour ces 43 instances, la valeur de la solution optimale est connue dans la littérature. Comparez vos résultats avec la métrique d'écart fournie.

## 3.2 Amélioration des heuristiques gloutonnes

On cherche maintenant à améliorer ces méthodes gloutonnes en limitant le choix de la prochaine tâche à celles pouvant commencer au plus tôt. Par exemple, si il y a trois tâches réalisables $(1, 2), (2, 1)$ et $(3, 3)$ dont la date de début au plus tôt sont respectivement 4, 6 et 4 ; alors le choix

de la prochaine tâche à faire sera réalisée entre les tâches $(1, 2)$ et $(3, 3)$ qui sont les deux pouvant commencer au plus tôt (c'est à dire à la date 4). Pour départager ces deux tâches, on utilisera l'une des règles de priorités (par exemple LRPT).

**A faire**

— Implémentation. Pour les priorités SPT et LRPT, implémentez la restriction aux tâches pouvant commencer au plus tôt. Ces nouvelles règles sont appelées EST_SPT et EST_LRPT.

— Evaluation. Testez et évaluez ces nouvelles heuristiques gloutonnes sur les mêmes instances (43 instances) que précédemment.

**A faire - Optionnel** Concevez et évaluez une version randomisée de ces heuristiques où une partie des choix est soumise à un tirage aléatoire.

# 4 Méthodes de descente

L'objectif de cette étape est de réaliser une méthode de descente. Cette méthode s'appuie sur la génération d'une solution initiale (par exemple avec une méthode gloutonne traitée à l'étape précédente) et sur l'exploration d'un voisinage.

## 4.1 Principe général

Le fonctionnement d'une méthode de descente s'appuie sur l'exploration successive d'un voisinage de solutions. Le principe général d'une méthode de descente est présenté dans l'algorithme 1. Il utilise les notations ci-dessous :

— $Pb$ : représente une instance d'un problème (ici le jobshop) ;

— $Makespan()$ : représente la valeur de la fonction objectif (ici le makespan, ou durée totale que l'on cherche à minimiser) ;

— $Neighboor()$ : représente le voisinage d'une solution

**Algorithm 1** Algorithme de Descente

{Initialisation}
$s \leftarrow Glouton(Pb)$ – générer une solution réalisable avec la méthode de votre choix ;
**repeat**
  {Sélectionner le meilleur voisin $s'$ sur tout le voisinage de $s$}
  $s' \leftarrow (s' \in Neighboor(s)$ tq $\forall s'' \in Neighboor(s), Makespan(s') \leq Makespan(s''))$
  **if** $Makespan(s') < Makespan(s)$ **then**
    $s \leftarrow s'$
  **end if**
**until** pas de voisin améliorant ou time out

La méthode de descente ne génère pas de cycle dans l'exploration des voisins successifs (elle produit une solution améliorante à chaque itération ou s'arrête). Elle se termine avec un optimum local (le voisinage ne permet pas de trouver de meilleure solution) ou lorsqu'un temps limite de calcul est atteint.

## 4.2 Voisinage proposé

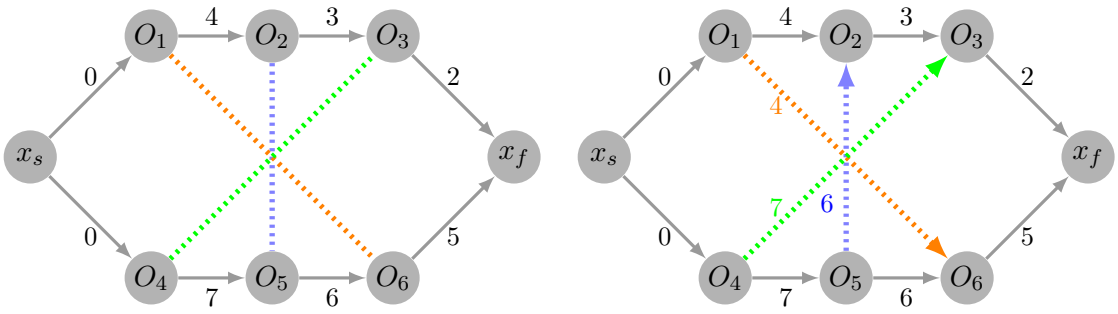
Le voisinage qui va être implémenté s'appuie sur la notion de chemin critique et sur la notion de blocs dans le chemin critique. Un bloc est composé de tâches utilisant la même ressource et consécutives dans le chemin critique.

### 4.2.1 Exemple illustratif

Ces notions de chemin critique et de bloc vont être illustrées à partir de l'exemple décrit dans la table 1 et représenté par le graphe de la figure 1a. Une solution pour cet exemple est présentée dans la figure 1b.

| | | | |
|---|---|---|---|
| $J_1$ | $O_1 = (M_3, 4)$ | $O_2 = (M_1, 3)$ | $O_3 = (M_2, 2)$ |
| $J_2$ | $O_4 = (M_2, 7)$ | $O_5 = (M_1, 6)$ | $O_6 = (M_3, 5)$ |

TABLE 1 – Exemple illustratif




(a) Graphe du problème

(b) Une solution

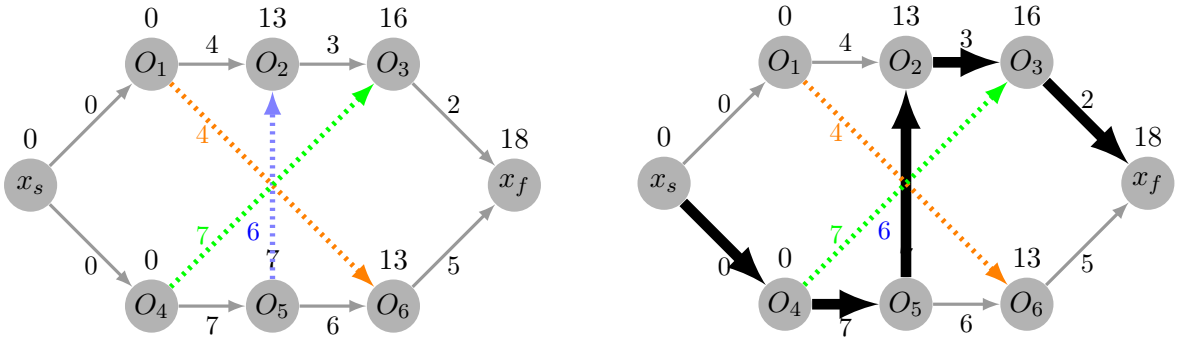
FIGURE 1 – Graphe de l'exemple illustratif

Dans le graphe de la figure 1a, les arcs en trait plein représentent les contraintes de succession liées aux jobs et les arêtes en pointillé représentent les choix d'ordre de passage sur les machines. Dans la solution représentée par le graphe de la figure 1b, l'ordre de passage sur les machines a été décidé (arcs en pointillé).

### 4.2.2 Chemin critique

Lorsqu'une solution réalisable est déterminée, il est possible de calculer les dates de début au plus tôt des différentes tâches du problème (cf. Graphes 3MIC - Chapitre plus court chemin - Exercice de TD sur le PERT). La date de début au plus tôt d'un sommet est le maximum parmis les dates de début au plus tôt de tous ses prédécesseurs plus le coût de l'arc. Le résultat de ce calcul est illustré dans la figure 2a. Ce calcul permet également de connaître le makespan, la date de début associée à la tâche fictive représentant la fin (noté $x_f$). La séquence de sommets ayant conduit à la valeur calculée pour le sommet ($x_f$) compose le chemin critique. Il est représenté en gras sur la figure 2b. Modifier la date de début d'une tâche du chemin critique entraîne un changement de la durée totale.

Le calcul d'un chemin critique est fourni dans la classe `Schedule` : méthode `criticalPath()` qui retourne une liste de tâches. La méthode `makespan()` permet de connaître la durée totale d'une solution.




(a) Evaluation de la solution

(b) Chemin critique associé

FIGURE 2 – Evaluation pour l'exemple illustratif

### 4.2.3 Blocs dans le chemin critique et voisinage associé

Un bloc dans le chemin critique est une séquence de tâches consécutives utilisant la même ressource. Cette notion de bloc dans le chemin critique a été proposée en 1986 par (Grabowski et.al). Sur l'exemple de la figure 2b il y a un seul bloc composé des tâches $(O_5, O_2)$.

Nous allons considérer un exemple un peu plus grand, composé de 3 jobs et 3 machines. Une solution et le chemin critique correspondant sont représentés sur la figure 3 dans laquelle chaque ligne correspond à un job et les couleurs correspondent aux différentes machines (machine $M_1$ en bleu, machine $M_2$ en vert et machine $M_3$ en orange). Sur cet exemple, le chemin critique est formé des tâches $\{O_4, O_8, O_9, O_1, O_6\}$. Il comporte 2 blocs : le bloc $\{O_4, O_8\}$ associé à la ressource $M_2$ (vert) et le bloc $\{O_9, O_1, O_6\}$ associé à la ressource $M_3$ (orange).

En pratique, il peut y avoir plusieurs blocs le long d'un chemin critique. On peut noter qu'un bloc doit comporter au moins deux tâches.

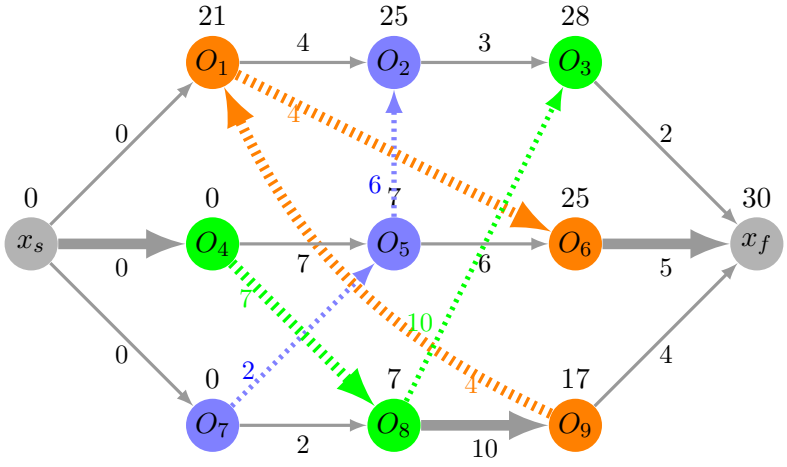



FIGURE 3 – Exemple comportant deux blocs de tâches sur le chemin critique

Le voisinage à implémenter consiste à considérer chaque bloc et pour chacun à permuter les deux tâches en début de bloc et les deux tâches en fin de bloc.

— Pour le bloc $\{O_4, O_8\}$, le voisinage consiste à permuter $O_4$ et $O_8$. Il n'y a qu'un seul voisin.

— Pour le bloc $\{O_9, O_1, O_6\}$, le voisinage contient deux voisins : permutation de $O_9$ et $O_1$ et permutation de $O_1$ et $O_6$.

De manière générale, lorsqu'un bloc comporte deux tâches, il ne permet de générer qu'une seule solution voisine. Tout bloc ayant au moins trois tâches permet de générer deux solutions voisines.

Lorsqu'une solution voisine est générée, elle peut ne pas être réalisable et sera dans ce cas rejetée.

Ce voisinage est appelé voisinage de Nowicki et Smutnicki (1996). Les auteurs ont montré que la permutation de tâches internes à un bloc ne permettait pas d'améliorer le chemin critique. Ils ont également proposé d'autres améliorations pour les blocs situés en début ou en fin de chemin critique.

Une synthèse de plusieurs voisinages basés sur la notion de bloc est proposée dans la section 4.1.4 de la thèse de Damien Lamy (2017) : `https://tel.archives-ouvertes.fr/tel-01758932`.

### A faire

— Implémentation.

  — Lisez le code fourni (et les commentaires) dans classe `Nowicki`.

  — Implémentez la méthode `blocksOfCriticalPath` pour extraire la liste des blocs d'un chemin critique

  — Implémentez la méthode `neighbors` pour générer le voisinage d'un bloc, c'est à dire l'ensemble des permutations (utilisez pour cela la classe `Swap` fournie). Le voisinage complet d'une solution correspond à l'ensemble des voisins, il sera construit dans la méthode `solve`.

  — Les solutions voisines générées doivent être réalisables. Comment pouvez-vous détecter cette situation dans votre implémentation ?

  — Implémentez les méthodes `applyOn` et `undoApplyOn` de la classe `Swap`.

  — Implémentez la méthode de descente dans `DescentSolver`.

— Evaluation.

  — Testez et évaluez votre méthode de descente sur les 43 instances déjà considérées.

  — En complément des métriques d'évaluation proposées, prenez également en compte le nombre de voisins explorés (ie. la taille du voisinage exploré à chaque itération).

  — Visualisez l'évolution de la fonction d'évaluation (makespan) et fonction des itérations de la méthode de descente.

  — Comparez les résultats pour différentes solutions initiales.

**A faire - Optionnel** Concevez, implémentez et évaluez une méthode de descente multi-start. Vous pouvez utiliser pour cela les différentes heuristiques gloutonnes pour avoir plusieurs solutions initiales ou vous pouvez utiliser une heuristique randomisée.

# 5 Méthode Tabou

L'objectif de cette étape est de développer une métaheuristique de type recherche Tabou afin de sortir des optima locaux. La méthode Tabou va s'appuyer sur le voisinage déjà implémenté lors de l'étape précédente. Elle permet d'explorer des solutions non améliorantes et utilise différentes techniques pour interdire de revenir vers des solutions déjà visitées.

## 5.1 Principe général

La méthode Tabou est une méta-heuristique basée sur l'exploration de voisinage et permettant de sortir des optima locaux en acceptant des solutions non améliorantes. Afin d'éviter de boucler sur des solutions déjà visitées, elle garde en mémoire (pendant un certain temps), la liste des solutions déjà visitées afin d'éviter de les considérer de nouveau.

Le principe général d'une méthode tabou est présenté dans l'algorithme 2.

**Algorithm 2** Algorithme de recherche Tabou

{Initialisation}
$s \leftarrow Glouton(Pb)$ – générer une solution réalisable avec la méthode de votre choix ;
$s^* \leftarrow s$ – mémoriser la meilleure solution
{Mise à jour des solutions Tabou}
$sTaboo \leftarrow \{s\}$
$k \leftarrow 0$ – compteur itérations
**repeat**
  {Exploration des voisins successifs}
  $k \leftarrow k + 1$
  {Choisir le meilleur voisin $s'$ non tabou}
  $s' \leftarrow (s' \in Neighboor(s) \setminus sTaboo$ tq $\forall s'' \in Neighboor(s) \setminus sTaboo, Makespan(s') \leq Makespan(s'')$
  {Mise à jour des solutions Tabou}
  $sTaboo \leftarrow sTaboo \cup \{s'\}$
  {Mémoriser si meilleure solution}
  **if** $Makespan(s') < Makespan(s^*)$ **then**
    $s^* \leftarrow s'$
  **end if**
  {Poursuivre avec nouvelle solution}
  $s \leftarrow s'$
**until** $k \geq maxIter$ ou time out

## 5.2 Voisinage

Le voisinage utilisé est le même que celui implémenté pour la méthode de descente. La seule différence réside dans l'exploration de ce voisinage. La solution voisine sélectionnée est la meilleure de tout le voisinage même si elle n'est pas améliorante.

## 5.3 Solutions Tabou

Pour gérer l'ensemble des solutions tabou, il est nécessaire de définir une structure de données permettant de vérifier si une solution a déjà été visitée ou non. Pour des raisons d'efficacité de

cette vérification, on mémorise généralement les changements (mouvements) interdits dans l'exploration d'un voisinage plutôt que les solutions déjà visitées. Ces interdictions limitent plus fortement l'exploration de l'espace de recherche qu'interdire uniquement les solutions déjà visitées.

Sur l'exemple de jobshop de la figure 3, deux voisins sont générés : un voisin avec la permutation $(O_9, O_1)$ et un voisin avec la permutation $(O_1, O_6)$. La méthode tabou, sélectionne le meilleur de ces deux voisins, par exemple celui obtenu avec la permutation $(O_1, O_6)$ et on va interdire la permutation inverse, c'est à dire $(O_6, O_1)$.

Pour éviter de déconnecter la solution courante de la solution optimale (ou pour des raisons d'espace mémoire), la méthode tabou utilise un paramètre, $dureeTaboo$, correspondant à la durée des interdictions. Au bout de cette durée, une solution tabou peut de nouveau être visitée.

Une proposition pour représenter l'ensemble $sTaboo$ pour le jobshop est d'utiliser une matrice carrée (nombre de tâches $\times$ nombre de tâches) dans laquelle on notera pour chaque interdiction le numéro d'itérations jusqu'auquel la permutation inverse est interdite. Par exemple, si la permutation $(O_1, O_6)$ a été effectuée à l'itération $k$, la case de la matrice $sTaboo(6, 1)$ reçoit la valeur $k + dureeTaboo$ pour interdire la permutation $(O_6, O_1)$ jusqu'à dépasser ce nombre d'itérations.

### A Faire

- Implémentation.
  - Ajoutez une classe `TabooSolver` en vous inspirant des autres solveurs déjà réalisés (pour la gestion du voisinage, vous pouvez simplement recopier ce qui a été codé dans `DescentSolver` même si ce n'est pas le plus élégant ...).
  - Implémentez la méthode `solve` de `TabooSolver` en supposant tout d'abord qu'il n'y a pas de solutions Tabou (arrêt au bout d'un nombre maximum d'itérations $maxIter$ ou limite de temps de calcul atteinte).
  - Ajoutez la structure permettant de gérer les solutions Tabou (toute solution Tabou est rejetée). Une solution reste Tabou pendant un nombre $dureeTaboo$ d'itérations.
  - Modifiez votre méthode `solve` pour accepter une solution tabou lorsqu'elle améliore la meilleure solution déjà trouvée.
- Evaluation.
  - Testez votre méthode de recherche Tabou sur les 43 instances précédentes.
  - Pour l'évaluation, faites varier les paramètres de la méthode ($maxIter$, $dureeTaboo$) et éventuellement la limite de temps de calcul.
  - En complément des métriques d'évaluation proposées, prenez également en compte le nombre de voisins explorés (ie. la taille du voisinage exploré à chaque itération).
  - Visualisez l'évolution de la fonction d'évaluation (makespan) et fonction des itérations de la méthode Tabou et mettez en évidence l'apparition de cycles.
  - Comparez les résultats pour différentes solutions initiales.

**A faire - Optionnel** Proposez et évaluez des mécanismes complémentaires pour sortir des optima locaux : détection de cycles, plus d'amélioration de la meilleure solution pendant un nombre d'itérations, retour sur la meilleure solution courante, ...

# 6 Evaluation expérimentale

L'objectif de cette partie est de construire une évaluation expérimentale des différents algorithmes implémentés et de produire une analyse des résultats obtenus.

**Protocole d'évaluation** Pour automatiser le lancement de l'évaluation expérimentale, il faut définir :

1. les jeux de données : utilisez les 80 instances de Taillard. Attention la valeur de la solution optimale n'est pas connue pour toutes ces instances.
2. les méthodes : on souhaite évaluer les différentes méthodes développées précédemment. Il y a deux objectifs :
   - déterminer la meilleure méthode dans chaque famille de méthode (meilleure variante de l'heuristique gloutonne, meilleur paramétrage de méthode de descente, la meilleure variante de la méthode tabou)
   - comparer entre elles chacune de ces familles de méthodes
3. les métriques d'évaluation : la valeur du makespan, l'écart à la meilleur solution connue, le nombre d'itérations et autres caractéristiques significatives (par exemple nombre de voisins explorés).

**Collecter et analyser les résultats.** Faire cette analyse comparative vous amènera à évaluer un grand nombre de solveurs (pour explorer les différents paramétrages possibles) et sur un grand nombre d'instances (pour avoir des résultats statistiquement significatifs). Réaliser ce genre d'analyse efficacement nécessite un minimum d'automatisation pour (1) faire tourner les tests, (2) collecter les résultats et (3) en produire une analyse et une visualisation. Vous êtes libre de vous organiser comme vous le souhaitez pour ceci. Néanmoins vous trouverez quelques pistes pour automatiser et faciliter l'analyse des résultats dans la documentation : `https://insa-4ir-meta-heuristiques.github.io/doc/benchmarking.html`