Méthodes Approchées pour la Résolution de Problèmes d'Ordonnancement

Arthur Bit-Monnot, Marie-Jo Huguet

1 Étapes de mise en place

1.1 Discord

- rejoignez le serveur Discord pour ces TP : https://discord.gg/KyUbCCT.
- indiquez votre *Prénom Nom* comme pseudo, pour que l'on puisse vous identifier
- en cas de problème technique, vous pourrez vous adresser au chan *support-technique* de ce serveur.

1.2 Document de suivi

- inscrivez vous dans le document de suivi : https://docs.google.com/spreadsheets/d/1QAZlWaTCvrMlLLuwuVFmCD8Plr9QvOoPVpIR4g1PSBk/edit?usp=sharing
- à chaque étape franchie dans le TP, ajountez un `X` dans la case correspondante

1.3 Récuperation du code

- Récupérez la base de code sur Github : https://github.com/insa-4ir-meta-heuristiques/template-jobshop
- Suivez les instructions dans le README pour vous assurer que le code compile.
- Importez le projet dans votre IDE préféré. Tous doivent pouvoir supporter l'import de projet gradle et quelques liens sont donnés en bas du README.

2 Prise en main

2.1 Représentation d'un problème de JobShop

Vous trouverez dans le dossier `instances/` un ensemble d'instances communément utilisées pour le problème de jobshop. Si l'on considère l'instance vu dans les exercices, constituée de deux jobs avec trois tâches chacun :

J_1	$r_1, 3$	$r_2, 3$	$r_3, 2$
J_2	$r_2, 2$	$r_1, 2$	$r_3, 4$

L'instance est nommée `aaa1` décrite ci-dessus est donnée dans le fichier `instances/aaa1` :

```
# Fichier instances/aaa1
2 3 # 2 jobs and 3 tasks per job
0 3 1 3 2 2 # Job 1 : (machine duration) for each task
1 2 0 2 2 4 # Job 2 : (machine duration) for each task
```

La première ligne donne le nombre de jobs et le nombre de tâches par jobs. Le nombre de machine 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 3 1 3 2 2` spécifie que pour le premier job :

- `0 3`: la première tâche s'execute sur la machine `0` et dure `3` unités de temps
- `1 3`: la deuxième tâche s'execute sur la machine `1` et dure `3` unités de temps
- `2 2`: la troisième tâche s'execute sur la machine `2` et dure `2` unités de temps

2.2 Base de code

Il vous est fourni une base de code pour faciliter votre prise en main du problème. Vous trouverez dans la classe `jobshop.Main` un point d'entrée pour réaliser des tests de performance de méthodes heuristiques. Les autres points d'entrée du programme sont les tests déjà présents ou pouvant être ajoutés dans le dossier `src/test/java/`.

2.2.1 Problème et Solution

- classe `jobshop.Instance` qui contient la représentation d'un problème et une méthode pour parser un fichier de problème.
- classe `jobshop.Schedule` qui contient la représentation directe, associant à chaque tâche une date de début. La classe schedule sert notamment à la représentation d'une solution et toute autre représentation doit pouvoir être traduite dans un `Schedule`

2.2.2 Représentation

- une classe abstraite `jobshop.Encoding` dont les sous classes sont des représentations d'une solution au JobShop.
- une classe `jobshop.encoding.NumJobs` qui contient une implémentation de la représentation par numéro de job

2.2.3 Solveurs

Deux solveurs très simples basés sur une représentation par numéro de job. Les nouveaux solveurs doivent être ajoutés à la structure `jobshop.Main.solvers` pour être accessibles dans le programme principal.

- `basic` : méthode pour la génération d'une solution pour une représentation par numéro de job
- `random` : méthode de génération de solutions aléatoires par numéro de job

2.3 À faire : manipulation de représentations

Ouvrez la méthode `DebuggingMain.main()`.

- Pour la solutions en représentation par numéro de job donnée, calculez (à la main) les dates de début de chaque tâche
- implémentez la méthode `toString()` de la classe `Schedule` pour afficher les dates de début de chaque tâche dans un schedule.
- Vérifiez que ceci correspond bien aux calculs que vous aviez fait à la main.

Création d'une nouvelle représentation par ordre de passage sur les ressources :

- Créer une classe `jobshop.encodings.ResourceOrder` qui contient la représentation par ordre de passage sur ressources vue dans les exercices (section 3.2). Il s'agit ici d'une reprsentation sous forme de matrice où chaque ligne correspond à une machine, et sur cette ligne se trouvent les tâches qui s'exécutent dessus dans leur ordre de passage. Pour la representation d'une tâche dans la matrice, vous pouvez utiliser la classe `jobshop.encodings.Task` qui vous est fournie.

- Pour cette classe, implémentez la méthode `toSchedule()` qui permet d'extraire une représentation directe. Pour l'implémentation de cette méthode `toSchedule()`, il vous faut construire un schedule qui associe à chaque tâche une date de début (vous pouvez regardez l'implémentation pour `JobNums` pour en comprendre le principe). Pour construire ce schedule il faudra que vous mainteniez une liste des tâches qui ont été schédulé. Cette liste est initialement vide. À chaque itération de l'algorithme, on identifie les tâches executables. Une tâche est executable si

  - son prédécesseur sur le job a été schedulé (si c'est la tache (1, 3), il faut que les tâches (1,1) et (1,2) aient été schédulée)
  - son prédécesseur sur la ressource a été schedulé (l'ordre sur passage sur les ressources est précisément ce qui vous est donné par cette représentation).

- Ajouter des tests dans `src/test/java/jobshop` permettant de vérifier que vos méthodes fonctionnent bien pour les exemples traités en cours (instance `aaa1`). Vous pouvez pour cela vous inspirer et ajouter des cas de test à `EncodingTests`.

Changement de représentation *Optionnel* :

- pour les deux représentations `ResourceOrder` et `JobNums`, créez des méthodes permettant de créer cette représentation depuis un `Schedule`.
- utilisez la pour tester la conversion de `ResourceOrder` vers `JobNums` et vice-versa.

3 Heuristiques gloutonnes

On souhaite implémenter une méthode gloutonne pour calculer une solution représentée par l'ordre de passage des tâches sur les machines (`ResourceOrder`).

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

3.1 À faire

- Créer un nouveau solveur (`GreedySolver`) implémentant une recherche gloutonne basée sur `ResourceOrder` pour les priorités SPT et LRPT.
- Évaluer ces heuristiques sur les instances fournies et pour la métrique d'écart fournie. Pour les tests de performance, on privilégiera les instance `ft06`, `ft10`, `ft20` et les instances de Lawrence `la01` à `la40`.

3.2 Amélioration de l'heuristique gloutonne

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

4 Mises à jour du code

Le dépôt git a été mis à jour avec quelques additions : https://github.com/insa-4ir-meta-heuristiques/template-jobshop. On vous laisse juger de la meilleure manière de récupérer les mises à jour.

En particulier, une implémentation de la classe `ResourceOrder`, que vous avez fait dans la première partie du TP, vous est maintenant fournie. Elle contient deux méthodes qui ne vous étaient pas demandées :

- `ResourceOrder(Schedule s)` : ce constructeur permet de créer une solution représentée par un `ResourceOrder` à partir d'une solution représentée par un `Schedule`
- `copy()` : cette méthode permet de réaliser une copie complète d'une solution

De plus, si vous n'avez pas eu le temps d'implémenter la méthode `toSchedule()`, celle-ci vous est également fournie. Si vous l'avez implémentée, vérifiez que votre méthode renvoie `null` si la solution n'est pas réalisable.

La classe `Schedule` contient maintenant une méthode `Schedule.criticalPath()` et une méthode `Schedule.makespan()` qui vous seront utiles pour l'étape suivante du TP.

Il vous est également fourni une classe `DescentSolver` qui servira de template pour la méthode de descente.

5 Méthode de descente

5.1 Principe général

Le fonctionnement d'une méthode de descente s'appuie sur l'exploration successive d'un voisinage de solutions. En utilisant les notations ci-dessous :

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

le principe général d'une méthode de descente est le suivant :

- Initialisation : $s_{init} \leftarrow Glouton(Pb)$: // générer une solution réalisable avec la méthode de votre choix;
- Mémoriser la meilleure solution : $s^* \leftarrow s_{init}$
- Répéter // Exploration des voisinages successifs
  1. Choisir le meilleur voisin s' :
     - $s' \in Neighboor(s^*)$ tq $\forall s'' \in Neighboor(s^*), Obj(s') < Obj(s'')$
  2. si s' meilleur que s^* alors $s^* \leftarrow s'$
- Arrêt : pas d'amélioration de la solution 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.

5.2 Voisinage proposé

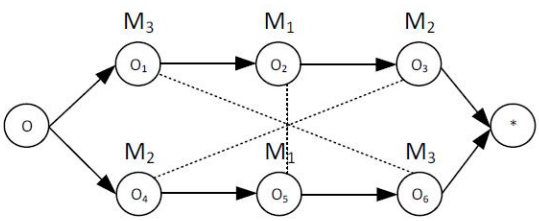
Le voisinage qui va être implémenté s'appuie sur la notion de chemin critique et sur la notion de blocs dans le chemin critique. Un bloc est composé de tâches utilisant la même ressource et consécutives dans le chemin critique.

Ces notions vont être illustrées à partir de l'exemple décrit dans la figure 1 et représenté par le graphe de la figure 2a. Une solution pour cet exemple est présentée dans la figure 2b.

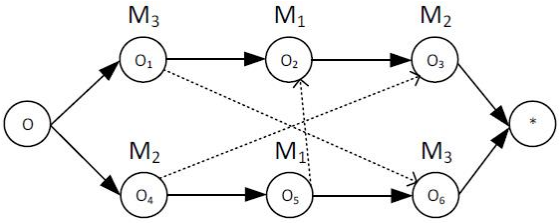
Job 1	$O_1 = (M_3 ; 4)$	$O_2 = (M_1 ; 3)$	$O_3 = (M_2 ; 2)$
Job 2	$O_4 = (M_2 ; 7)$	$O_5 = (M_1 ; 6)$	$O_6 = (M_3 ; 5)$

Figure 1: Exemple avec 2 jobs - 3 machines

Dans le graphe de la figure 2a, 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.



(a) Graphe du problème



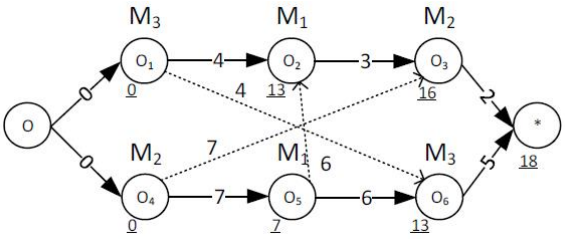
(b) Une solution

Dans la solution représentée par le graphe de la figure 2b, l'ordre de passage sur les machines a été décidé (arcs en pointillé).

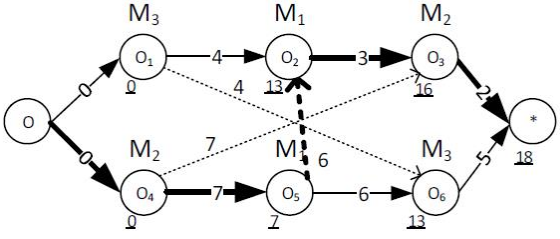
5.2.1 Chemin critique

Lorsqu'une solution réalisable est déterminée, il est possible de calculer les dates de début au plus tôt des différentes tâches du problème (à l'aide d'un algorithme de plus long chemin dans un graphe sans circuit). Le résultat de ce calcul est illustré dans la figure 3a. Ce calcul permet également de connaître la durée totale de l'ordonnancement (date associée à la tâche fictive représentant la fin (noté *). Lors du calcul du plus long chemin, la date de début au plus tôt d'un sommet est actualisée avec la valeur maximale issue des sommets prédécesseurs. La séquence de sommets ayant conduit à la valeur calculée pour le sommet (*) compose le chemin critique. Il est représenté en gras sur la figure 3b. 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

5.2.2 Blocs dans le chemin critique et voisinage associé

Un bloc dans le chemin critique est une séquence de tâches consécutives utilisant la même ressource. Sur l'exemple de la figure 4, il y a un bloc composé de trois tâches : O_9, O_1, O_6. Ces trois tâches utilisent toutes la machine M_3 et sont consécutives le long du chemin critique. 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.

Attention : il y a une erreur dans le chemin critique de la figure 4. Il est composé des tâches $\{O_4, O_8, O_9, O_1, O_6\}$. Ce chemin critique comporte 2 blocs : le bloc $\{O_4, O_8\}$ associé à la

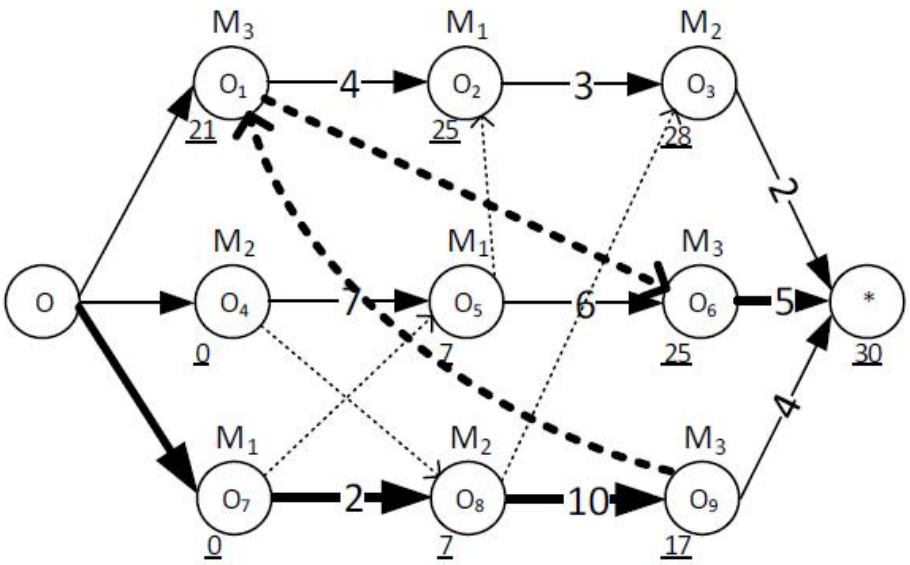


Figure 4: Exemple : Bloc de 3 tâches sur le chemin critique. Un bloc est ici la séquence O_9, O_1, O_6 qui se suivent sur le chemin critique et s'exécutent sur la machine M_3.

ressource M_2 et le bloc $\{O_9, O_1, O_6\}$ associé à la ressource M_3.

Le voisinage à implémenter consiste à permuter les deux tâches en début de bloc et les deux tâches en fin de bloc. Pour le bloc $\{O_9, O_1, O_6\}$ associé à la ressource M_3 de la figure 4, le voisinage généré consiste donc à permuter O_9 et O_1 et à permuter O_1 et O_6 (il y a deux voisins pour ce bloc).

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.

5.3 A faire

- Lire le code fourni (et les commentaires) dans la classe `DescentSolver`.
- Écrire la méthode `blocksOfCriticalPath` pour extraire la liste des blocs d'un chemin critique
- Écrire la méthode `neighbors` pour générer le voisinage d'un bloc, c'est à dire l'ensemble des permutations (utiliser 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`.
- Écrire la méthode `applyOn` de la classe `Swap`.
- Implémenter la méthode de descente dans `solve` (retournant un `Schedule`).
- Tester votre méthode de descente sur les instances fournies.

6 Méthodes exhaustives

6.1 Espace de recherche

La taille de l'espace de recherche dépend de la représentation choisie :

- avec la représentation par numéro de job (`NumJobs`), cette taille correspond au nombre de permutations du numéro de jobs avec répétition. Elle est égale à $\frac{(n \times m)!}{m!^n}$.
- avec la représentation par ordre de passage sur les ressources (`ResourceOrder`), cette taille correspond à toutes les permutations de tâches sur les ressources. Elle est égale à $(n!)^m$ où n est le nombre de jobs et m le nombre de machines (en supposant que les tâches d'un job passent par toutes les machines). Parmi ces permutations, certaines correspondent à des solutions non réalisables.
- avec la représentation par date de début de chaque tâche (`Schedule`), cette taille dépend des valeurs possibles des dates de début et du nombre de tâches. Elle est égale à $(D_{max})^{n \times m}$ où D_{max} représente la durée maximale d'un ordonnancement (dans le pire cas, c'est la somme des durée de toutes les tâches). Parmi ces solutions, certaines ne sont pas réalisables.

6.2 A Faire (pour le rapport)

- Calculer le nombre de solutions de l'espace de recherche associé à chaque représentation pour l'instance `ft06`
- En supposant que la génération et l'évaluation d'une solution prend une nano-seconde (pour chacune des représentations), quel serait le temps nécessaire pour explorer l'espace de recherche associé à chaque représentation ?
- Quelles conclusions en tirez-vous sur les représentations de solutions ?
- Quelles conclusions en tirez-vous sur les méthodes exhaustives ?

7 Méthode Tabou

7.1 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 le suivant :

- $s_{init} \leftarrow Glouton(Pb)$: // générer une solution initiale réalisable
- $s^* \leftarrow s_{init}$ // Mémoriser la meilleure solution
- $s \leftarrow s_{init}$ // solution courante
- $sTaboo \leftarrow \{s\}$ // solutions tabou
- $k \leftarrow 0$ // compteur itérations
- Répéter // Exploration des voisinages successifs
  1. $k \leftarrow k + 1$
  2. Choisir le meilleur voisin s' non tabou :
     - $s' \in Neighboor(s)$ tq $\forall s'' \in Neighboor(s), Obj(s') < Obj(s'')$ et $s' \notin sTaboo$
  3. Ajouter s' à $sTaboo$
  4. $s \leftarrow s'$
  5. si s' meilleur que s^* alors $s^* \leftarrow s'$
- Arrêt : $k \geq maxIter$ ou time out

7.2 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.

7.3 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 4, 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 dans laquelle on notera pour chaque interdiction la numéro d'itérations à partir de laquelle le mouvement est autorisé : $k_{OK} \leftarrow k + dureeTaboo$ où k est l'itération ayant produit l'interdiction. Sur l'exemple précédent, si la permutation (O_6, O_1) est interdite à l'itération k, on obtient la matrice représentée dans le tableau 1

	O_1	...	...	O_6	...	O_9
O_1						
...						
...						
O_6	k_{OK}					
...						
O_9						

Table 1: Exemple de représentation des mouvements tabou

Avec cette représentation, vérifier à une itération k' qu'une permutation est autorisée (ie. le voisin n'est pas tabou) est simple : il suffit de tester si $k' \geq k_{OK}$ pour la case associée à cette permutation (initialement toutes les cases de la matrice ont la valeur 0).

7.4 A Faire

- Ajouter 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 ...).
- Ecrire 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).
- Ajouter la structure permettant de gérer les solutions Tabou (toute solution Tabou est rejetée). Une solution reste Tabou pendant un nombre $dureeTaboo$ d'itérations.
- Modifier votre méthode `solve` pour accepter une solution tabou lorsqu'elle améliore la meilleure solution déjà trouvée.
- Tester votre méthode de Tabou sur les instances fournies en faisant varier les paramètres de la méthode ($maxIter$, $dureeTaboo$) et éventuellement la limite de temps de calcul.