Méthodes Approchées
Résolution de Problèmes d'Ordonnancement
4-IR

Durant ces TP vous allez implémenter et évaluer des méthodes d'optimisation pour minimiser la durée d'un problème d'ordonnancement. Le problème d'ordonnancement proposé est le problème de Job-Shop (présenté en cours). Pour le résoudre, il faut planifier la réalisation de différentes tâches tout en respectant des contraintes de séquencement et des contraintes de ressources. Le critère d'optimisation est la minimisation de la durée totale pour réaliser toutes les tâches. Ce critère est aussi appelé *makespan*.

Pour résoudre ce problème d'optimisation combinatoire, différentes méthodes approchées seront progressivement développées et testées pour s'assurer qu'elles fonctionnent correctement.

Un protocole d'évaluation expérimentale devra ensuite être défini pour permettre d'analyser les intérêts, performances et limites de chaque méthode.

MJ. Huguet - A. Bit-Monnot

Année Universitaire 2024-2025

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 effectué en binôme
- Page moodle : `https://moodle.insa-toulouse.fr/course/view.php?id=1354` comportant le planning des séances de TP.
- L'évaluation se base sur un mini-compte rendu des TP à remettre sur moodle (1 rapport par binôme) et sur une évaluation orale.

2 Découverte du code et du problème de Job Shop

2.1 Mise en place (avant TP1)

Pour développer les méthodes proposées, vous allez partir d'une base de code en Java et d'un ensemble de jeux de données.

A faire **Rendez vous sur la documentation :** `https://insa-4ir-meta-heuristiques.github.io/jobshop/` et **Suivez les instructions** de la page d'accueil.

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 Problème de Job-Shop (avant TP1)

Le problème de Job Shop a été présenté en cours. Ses caractéristiques sont rappelées ci-après : un job est une séquence de tâches, chaque tâche ne nécessite qu'une seule machine et une fois démarrée ne peut être interrompue, les machines ne peuvent réaliser qu'une seule tâche à la fois).

Des jeux de données pour le problème de Job Shop sont fournis avec le code dans le dossier `instance`. 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).

Notations. Par la suite, on note n le nombre de jobs et m le nombre de tâches par job. Le nombre de ressources est égal au nombre de tâches. On peut remarquer une spécificité des jeux de données proposés qui sont dits "rectangulaires" : pour chaque job toutes les machines sont utilisées une et une seule fois.

Trois jeux de données "jouet" sont également proposés pour vous permettre d'exécuter à la main vos algorithmes : (`aaa1`, `aaa2` et `aaa3`). **Attention : les instances "jouet" ne sont pas à considérer pour la campagne d'évaluation expérimentale car ils sont de taille trop réduite.**

2.3 Architecture du code et Manipulation des représentations de solution

À faire. Documentation : `https://insa-4ir-meta-heuristiques.github.io/jobshop/`

- Lisez les pages `Entry Points`, `Instances` et `Encodings`

L'instance "jouet" nommée `aaa1` est constituée de deux jobs avec trois tâches chacun et utilisant 3 machines (r_0, r_1, r_2). Elle 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 0
1 2 0 2 2 4 # Job 1
```

J_0	$r_0, 3$	$r_1, 3$	$r_2, 2$
J_1	$r_1, 2$	$r_0, 2$	$r_2, 4$

À faire. Ouvrez la classe `jobshop.encodings.ManualEncodingTests` (dans le dossier de tests). Il s'agit d'une classe de tests dont plusieurs méthodes sont actuellement désactivées (annotation `@Ignore`). Cette classe de tests dispose d'une méthode `setUp()`,appelée avant chaque test, qui charge l'instance `aaa1` et génère une solution pour cette instance. Chacun des points ci-dessous correspond à une méthode de test de la classe `ManualEncodingTests` qu'il vous faut (1) activer en enlevant l'annotation `@Ignore`, et (2) compléter pour qu'elle réponde aux attentes.

- méthode `testManualSchedule()`. Activez et exécutez la méthode pour visualiser la solution proposée de l'instance `aaa1`.
  - Quelle est la représentation utilisée pour renvoyer le résultat (ie. une solution) ?
- En restant dans la méthode `[testManualSchedule]` : 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).
  - Quelle est la méthode de résolution appelée et comment fonctionne-t-elle ?
- méthode `[testManualResourceOrder]`. Reconstruisez à la main la même solution dans la représentation par ordre de passage sur les ressources `ResourceOrder`.
- méthode `[testOptimalResourceOrder]`. 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.
- méthode `[testInvalidResourceOrder]` 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.
  - Pourquoi avoir deux représentations de solution (intérêts, limites) ?

À la fin de cette étape, vous devez avoir compris à quoi servent les classes du dossier `encodings`, en particulier `Schedule` et `ResourceOrder`. Vous connaissez également les différentes méthodes de ces classes (c'est important pour la suite pour vous évitez de coder une méthode qui est déjà fournie ...)

2.4 Bilan

Vous avez pris connaissance du code, de la documentation et des jeux de données. Vous êtes prêt pour implémenter des méthodes de résolution pour le Job-Shop.

3 Heuristiques gloutonnes

L'objectif de cette section est d'implémenter plusieurs méthodes gloutonnes visant à minimiser la durée totale d'un problème de Job-Shop. On considère 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 pouvant être réalisées, appelé ensemble des tâches *possibles* (initialement, les premières tâches de tous les jobs).
- **Boucle** : tant qu'il y a des tâches possibles

  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 possibles

On dit qu'une tâche est *possibles* 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 possibles et que toutes les tâches doivent être traitées.

Attention. Pour implémenter efficacement l'ensemble des tâches possibles, on peut utiliser une collection dynamique (`ArrayList`, `HashSet`).

Un paramètre déterminant dans la construction d'une heuristique gloutonne est la manière dont est choisie la prochaine tâche parmi toutes les tâches possibles. 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 restante.

A faire

- Documentation. `https://insa-4ir-meta-heuristiques.github.io/jobshop/`. Lisez les pages `Entry Points` et `Solvers`
- Implémentation. Complétez la classe `GreedySolver` implémentant une recherche gloutonne basée sur `ResourceOrder` pour les priorités **SPT et LRPT**.
- Test. Pour vérifier 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 expérimentale. Evaluez ces heuristiques sur les instances $ft06, ft10, ft20$ et sur les instances $ta01, \ldots, ta40$. Pour ces 43 instances, la valeur de la solution optimale est-elle 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 possible à celles pouvant commencer au plus tôt. Par exemple, si il y a trois tâches possibles $(1, 2), (2, 1)$ et $(3, 3)$ dont les dates de début au plus tôt sont respectivement 4, 6 et 4 ; alors le choix de la prochaine tâche à faire sera effectué entre les tâches $(1, 2)$ et $(3, 3)$ qui sont celles 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 (SPT ou LRPT).

A faire

- Implémentation. Pour les priorités SPT et LRPT, implémentez la restriction aux tâches pouvant commencer au plus tôt. Pour cela, il faut modifier la gestion de la liste des tâches possibles et pouvoir mettre à jour les dates de début des tâches non encore sélectionnées. Pour respecter les contraintes du problème, une tâche ne peut pas commencer si la tâche précédente sur le job n'est pas terminée ou si la ressource n'est pas disponible. **Ces nouvelles règles sont appelées EST_SPT et EST_LRPT.**
- Test. Pour vérifier votre implémentation, vous pouvez utiliser l'instance `aaa3` qui produit des résultats déterministes pour chacune des priorités proposées.
- Evaluation expérimentale. Evaluez ces nouvelles heuristiques gloutonnes sur les mêmes instances (43 instances) que précédemment.

3.3 Heuristiques gloutonnes avec de l'aléatoire

Toujours dans le but d'améliorer les résultats des méthodes gloutonnes, on va introduire un paramètre aléatoire pour perturber le choix de la prochaine tâche parmi la liste des tâches possibles. Par exemple, si on fixe le paramètre de randomisation à $0,99$ dans 99% dans cas on applique le choix de l'heuristique et dans 1% des cas on choisit n'importe quelle tâche parmi celles possibles. Cette perturbation permet d'obtenir une solution différente à chaque exécution de la méthode heuristique "randomisée". On peut retenir la meilleure solution obtenue.

A faire Avant de vous lancer dans cette partie, vérifiez le planning des séances de TP sur moodle.

- Implémentation. Concevez et évaluez une version randomisée de vos heuristiques gloutonnes (sur les mêmes instances que précédemment). Pensez à ajouter deux paramètres pour fixer : (a) le taux d'aléatoire et (b) le nombre d'exécution de la méthode randomisée.
- Test. Différentes valeurs des paramètres de randomisation sont à comparer.

3.4 Bilan

A la fin de cette section, vous disposez de plusieurs heuristiques gloutonnes fonctionnelles pour la résolution du problème de Job-Shop. Lors des évaluations expérimentales, vous avez pu comparer différentes heuristiques entre elles (par exemple SPT versus LRPT, EST_SPT versus EST_LRPT). De plus, vous devez avoir constaté que les heuristiques basées sur EST obtiennent de meilleurs résultats (par exemple EST_SPT versus SPT et EST_LRPT versus LRPT). Vous pouvez également compléter ce bilan en y intégrant les variantes d'heuristiques avec aléatoire.

4 Méthode de descente - Intensification

L'objectif de cette étape est d'implémenter une méthode de descente. Cette méthode débute avec la génération d'une solution initiale (par exemple avec une méthode gloutonne traitée à l'étape précédente) et effectue des explorations successives du voisinage de solutions.

4.1 Principe général

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. Il a été présenté en cours (voisinage de petite taille, guidé par la structure du problème).

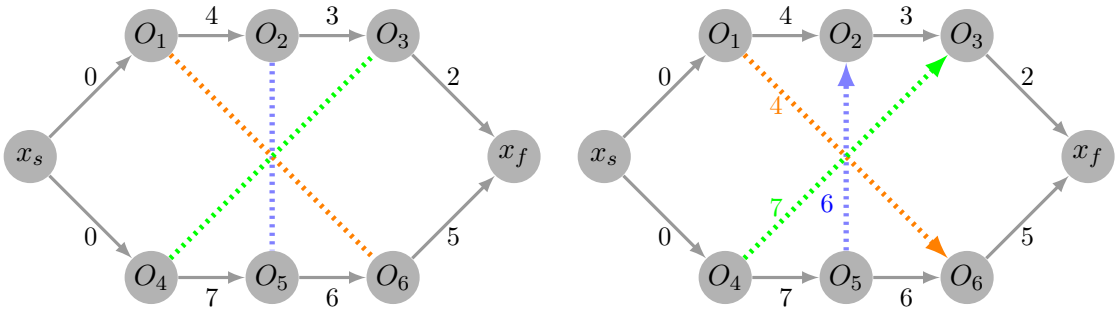
4.2.1 Exemple illustratif

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.

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).

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

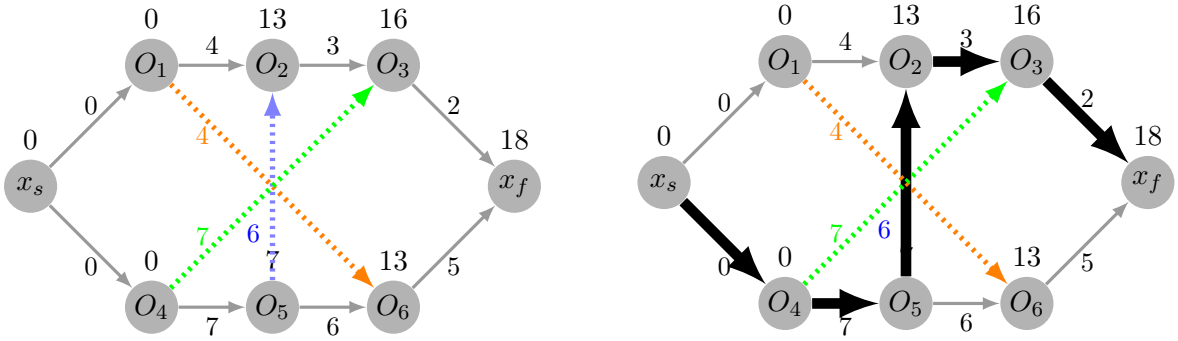
(b) Une solution

FIGURE 1 – Graphe de l'exemple illustratif

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.

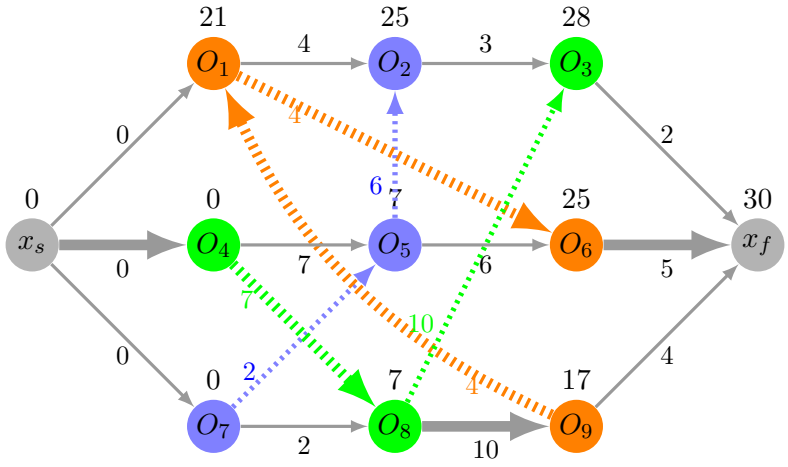


FIGURE 3 – Exemple comportant deux blocs de tâches sur le chemin critique

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. De plus, une voisinage vide (aucune permutation possible) implique que la solution courante est optimale.

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

A faire

- Documentation.
  - `https://insa-4ir-meta-heuristiques.github.io/jobshop/`. Lisez les pages `Solvers` et `Benchmarking`
  - Code : Lisez le code fourni (et les commentaires) dans la classe `Nowicki`.
- Implémentation.
  - Implémentez la méthode `allSwaps` 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`.
  - Implémentez la méthode `generateFrom` de la classe `Swap`, qui doit permettre de générer un nouveau `ResourceOrder` à partir d'un `Swap`.
  - Les solutions voisines générées doivent être réalisables. Comment pouvez-vous détecter des voisins non réalisables ?
  - Implémentez la méthode de descente dans `DescentSolver`.
- Visualisation. Visualisez l'évolution de la fonction objectif (makespan) des solutions visitées lors des itérations de la méthode de descente. Vérifiez que le makespan diminue au cours des itérations.
- Evaluation expérimentale.
  - Evaluez votre méthode de descente sur les 43 instances déjà considérées en utilisant différentes initialisations.
  - 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).
- Compréhension et Analyses
  - est-ce que la méthode `descente-xxx` améliore la méthode `glouton-xxx` ?
  - analyser le temps de calcul de la méthode de descente : quel est le temps d'exploration du voisinage d'une solution, combien d'itérations de descente y-a-t-il ?

4.3 Amélioration d'une méthode de descente

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.

4.4 Bilan

A la fin de cette section, vous disposez d'une méthode de descente. Cette méthode peut débuter avec différentes solutions initiales utilisant les heuristiques gloutonnes développées dans la section précédente.

Avec les résultats des évaluations expérimentales, vous pouvez comparer entre elles vos méthodes de descente avec les différentes initialisations. De plus, vous pouvez noter que la solution issue d'une méthode de descente est une meilleure solution que la solution initiale utilisée, par exemple en comparant Descent_SPT avec SPT et avec toute autre combinaison.

Vous pouvez compléter ce bilan en y intégrant les variantes de descente `multi-start`.

5 Méthode Tabou - Compromis intensification et diversification

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. Pour introduire de la diversification dans l'exploration de l'espace de recherche, la méthode Tabou permet d'explorer des solutions non améliorantes et elle 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, trois voisins sont générés : un voisin avec la permutation (O_4, O_8), 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 trois voisins, par exemple celui obtenu avec la permutation (O_1, O_6) et va interdire la permutation inverse, c'est à dire (O_6, O_1).

De plus, 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 liste de paires de tâches représentant les permutations interdites. Proposer une implémentation efficace de cette liste tabou (test d'appartenance, ajout, gestion de la durée des interdictions).

A Faire

- Documentation. `https://insa-4ir-meta-heuristiques.github.io/jobshop/`. Lisez les pages `Entry Points`, `Solvers` et `Benchamrking`
- 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 ...).
  - Première version : 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).
  - Deuxième version : ajoutez la structure permettant de gérer les solutions Tabou (toute solution Tabou est rejetée). Une solution reste Tabou pendant un nombre $dureeTaboo$ d'itérations.
  - Troisième version : Modifiez votre méthode `solve` pour accepter une solution tabou lorsqu'elle améliore la meilleure solution déjà trouvée.
- Visualisation. 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**. Vérifiez que le makespan peut augmenter au cours des itérations mais qu'il diminue globalement.
- Evaluation expérimentale.
  - Evaluez votre méthode de recherche Tabou sur les 43 instances précédentes en utilisant différentes initialisations et en considérant différentes valeurs pour les paramètres de la méthode :
    - nombre maximum d'itérations de la méthode ($maxIter$), durée des interdictions pour les Tabou ($dureeTaboo$), éventuellement 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).

5.4 Améliorations de la méthode Tabou

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

5.5 Bilan

A la fin de cette section, vous disposez d'une méthode Tabou. Cette méthode peut débuter avec différentes solutions initiales et dispose de plusieurs paramètres.

Avec les résultats des évaluations expérimentales, vous pouvez comparer entre elles différentes variantes de méthodes Tabou et identifier des paramétrages de la méthode.

De plus, vous pouvez noter que la solution obtenue avec une méthode Tabou (bien paramétrée) est une meilleure solution que la solution obtenue avec une méthode de descente (pour une même initialisation). Par exemple, vous pouvez comparer Tabou(paramètres)_SPT avec Descent_SPT et avec SPT ainsi que toute autre combinaison.

Vous pouvez compléter ce bilan en y intégrant d'autres améliorations de la méthode Tabou.