POO pour l'optimisation
IG2I, LE4-SI – 2021-2022

TP 6 – Métaheuristiques

Diego Cattaruzza, Maxime Ogier

Contexte et objectif du TP

Dans ce TP, nous nous intéressons encore à la résolution du problème de CVRP (*Capacitated Vehicle Routing Problem*). Dans le TP précédent, vous avez implémenté un algorithme de recherche locale afin de pouvoir améliorer une solution réalisable du problème de CVRP. Normalement, vous avez dû constater que vous étiez capable d'améliorer une solution réalisable en quelques centaines de millisecondes.

Cependant, il est probable qu'on soit capable de produire des solutions encore de meilleure qualité, en ajoutant un peu de temps de résolution.

En fait, le problème de la recherche locale est qu'elle s'arrête lorsqu'il n'existe plus aucune solution voisine qui permet d'améliorer la solution courante. Dans ce cas on a trouvé un *minimum local*. Mais il est très peu probable que ce minimum local soit un *minimum global*, i.e. une solution optimale du problème.

Afin de ne pas stopper l'algorithme lorsque la recherche locale ne trouve plus de solution voisine améliorante, il est possible d'utiliser une *métaheuristique*. Une métaheuristique est une méthode générique (non spécifique à un problème) qui permet de guider la recherche de solutions en essayant d'échapper aux minimas locaux. Il existe plusieurs métaheuristiques. Nous allons ici en voir une en détail : *la recherche tabou*.

Ce TP permet d'illustrer les notions suivantes :

- les principaux types de métaheuristiques ;
- la recherche tabou ;
- mouvement tabou ;
- critère d'aspiration ;
- file FIFO de taille fixe : **LinkedBlockingQueue**.

Bonnes pratiques à adopter

- Testez votre code au fur et à mesure de votre avancement.
- Respectez les convention de nommage Java.
- Faites des indentations de manière correcte.
- Donnez des noms compréhensibles et pertinents à vos variables, attributs, classes, méthodes.
- Respectez autant que possible le principe d'encapsulation.
- Utilisez des structures de données adaptées.
- Faites des méthodes courtes avec peu de niveaux d'indentation.
- Ne faites pas de duplication de code, mais privilégiez le code réutilisable.
- Utilisez les outils de votre IDE pour faire du *refactoring* (clic-droit puis '*Refactor*' sur Netbeans).
- Utilisez la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez.
- Quand c'est nécessaire, commentez votre propre code de façon pertinente au format Javadoc (commentaire commençant par `"/**"` au-dessus des définitions des attributs et méthodes).

1 Introduction

Supposons que vous obtenez une solution $\mathcal{S}$ avec une des méthodes constructives et l'application d'une recherche locale codée lors du dernier TP. La solution $\mathcal{S}$ que vous avez obtenue appartient à l'ensemble $\mathcal{SOL}$ de toutes les solutions de votre problème de CVRP (dans ce contexte, mais ce concept s'applique à n'importe quel problème d'optimisation). La solution $\mathcal{S}$ que vous avez obtenue n'est pas forcément la meilleure solution que vous pouvez obtenir. Le problème de la recherche locale est que la solution $\mathcal{S}$ que vous avez obtenue est un minimum local (i.e. aucune solution voisine de $\mathcal{S}$ n'est meilleure) ; mais pour autant il y a de grandes chances que ce ne soit pas un minimum global (i.e. une solution optimale). Dans ce TP, nous allons nous intéresser à la manière d'améliorer la qualité de cette solution.

L'ensemble des solutions $\mathcal{SOL}$ étant très grand (plus de $N!$ pour un problème avec N clients), il n'est pas possible de parcourir toutes les solutions. La recherche locale nous offre un moyen intéressant d'*intensifier* la recherche : étant donné une solution $\mathcal{S}$, l'application de la recherche locale permet de fournir une solution $\mathcal{S}'$ voisine de $\mathcal{S}$ de meilleure qualité. L'idée des métaheuristiques est de permettre également de *diversifier* la recherche dans l'ensemble de solutions $\mathcal{SOL}$: étant donné une solution $\mathcal{S}$, il faut pouvoir fournir une solution $\mathcal{S}'$ qui soit "différente" (i.e. pas voisine) de $\mathcal{S}$ et de bonne qualité.

Une caractéristique importante d'une méthode métaheuristique est d'être générique, c'est-à-dire que la méthode n'est pas dédiée à un problème spécifique (par exemple le CVRP), mais peut s'appliquer à n'importe quel problème d'optimisation.
Par ailleurs, il faut bien noter qu'une métaheuristique (comme une heuristique) ne garantit pas du tout que la solution obtenue soit une solution optimale.

Il existe deux types principaux de métaheuristiques : les métaheuristiques à base de voisinage et les métaheuristiques à base de population. Les métaheuristiques à base de voisinage fonctionnent en appliquant des opérateurs de voisinage à une solution courante. Ainsi, à chaque itération une unique solution est considérée. À la différence d'une recherche locale, il est possible d'appliquer un opérateur qui dégrade la qualité de la solution, ou d'appliquer des opérateurs de voisinage large qui modifient beaucoup la solution. Ceci permet de faire de la diversification. Les métaheuristiques à base de voisinage les plus connues sont : la recherche tabou, le recuit simulé, la recherche à voisinage variable. Les métaheuristiques à base de population consistent à garder en mémoire un ensemble de solutions (la population). À chaque itération, l'ensemble de ces solutions évolue. Les métaheuristiques à base de population les plus connues sont : les algorithmes génétiques, les algorithmes de colonies de fourmis.

Nous allons ici voir comment implémenter une métaheuristique à base de voisinage : la recherche tabou.

2 Recherche tabou

2.1 Présentation générale

La recherche tabou a été initialement proposée par Fred Glover [2]. L'idée était de permettre à des méthodes de recherche locale de ne pas rester coincées dans un minimum local. Le principe de base de la recherche tabou est de poursuivre la recherche locale même lorsqu'un minimum local est trouvé, en autorisant d'appliquer des mouvements non améliorants (d'une solution $\mathcal{S}$, on peut passer à une solution $\mathcal{S}'$ même si $\mathcal{S}'$ a un coût plus élevé que $\mathcal{S}$). Afin d'éviter de cycler en retombant sur une solution déjà rencontrée, on utilise une mémoire, appelée *liste tabou*, qui enregistre l'historique récent des mouvements appliqués à la solution courante. C'est là l'idée clé de l'algorithme, qui vient des concepts de l'intelligence artificielle.

Notez bien qu'une recherche tabou sans mémoire a peu de chances d'être efficace. En effet, lorsqu'on trouve une solution $\mathcal{S}^*$ qui est un minimum local, alors la recherche tabou va ensuite générer une solution $\mathcal{S}'$ qui est de moins bonne qualité que $\mathcal{S}^*$. Ainsi, à l'itération suivante, il apparait que $\mathcal{S}^*$ est une solution voisine de $\mathcal{S}'$, qui est de meilleure qualité. Il est fort probable que $\mathcal{S}^*$ soit la meilleure solution voisine de $\mathcal{S}'$, et donc la recherche tabou va cycler et revenir à la solution $\mathcal{S}^*$.

2.2 Structure de voisinage

Comme mentionné précédemment, la recherche tabou peut être vue comme une extension d'une recherche locale avec l'utilisation d'une mémoire à court terme. Ainsi, comme dans une recherche locale, il faudra définir les opérateurs utilisés dans le voisinage. Le choix des opérateurs de voisinage est un élément important de la performance d'une recherche tabou.

Comme nous avons déjà proposé des opérateurs de voisinage pour la recherche locale, nous pouvons reprendre les mêmes pour la recherche tabou.

2.3 Tabous

Les mouvements tabous sont une des caractéristiques distinctives majeures de la recherche tabou par rapport à une recherche locale. Comme mentionné auparavant, les mouvements tabous sont utilisés afin d'éviter de cycler sur des solutions déjà rencontrées lorsque l'on implémente des mouvements non améliorants.

Afin d'éviter de cycler, on déclare tabous les mouvements qui inversent les effets des mouvements récemment appliqués à la solution courante. Un mouvement tabou ne peut pas être appliqué à la solution courante. Par exemple, dans le cas du CVRP, si le client c_1 a été déplacé de la tournée r_1 vers la tournée r_2, alors on peut déclarer tabou le mouvement qui consiste à déplacer le client c_1 de la tournée r_2 vers la tournée r_1. Notez que ce mouvement ne restera pas tabou pendant toute la durée de l'algorithme, mais uniquement pendant quelques itérations. L'utilisation des mouvements tabous a l'avantage de permettre de diversifier la recherche en évitant de revenir sur des solutions déjà rencontrées.

Les mouvements tabous sont stockés dans une mémoire à court terme (la liste tabou), et en général le nombre de mouvements dans la liste ne varie pas et est relativement petit (de l'ordre de quelques dizaines). La liste est gérée de manière FIFO : le premier mouvement rentré dans la liste sera le premier à en sortir lorsque la capacité est atteinte.

En général, il existe plusieurs possibilités pour mémoriser les informations liées à un mouvement. On pourrait par exemple envisager de mémoriser la solution complète. Mais dans ce cas, vérifier si un mouvement est tabou est couteux en terme de comparaison (il faut comparer deux à deux les solutions). Ainsi, souvent on enregistre seulement les transformations qui ont été effectuées, et on rend tabou les transformations inverses. Par exemple, si on a appliqué le mouvement suivant : le client c_1 a été déplacé de la tournée r_1 vers la tournée r_2, alors on peut enregistrer le mouvement sous la forme d'un triplet (c_1, r_1, r_2) et ainsi rendre tabou le mouvement inverse qui consiste à déplacer le client c_1 de la tournée r_2 vers la tournée r_1. Notez qu'en enregistrant ce type d'informations, des cycles sont possibles : par exemple, on déplace ensuite c_1 de r_2 vers r_3, puis on déplace c_1 de r_3 vers r_1 : on se retrouve à la solution initiale ! Il est possible d'enregistrer différemment un mouvement tabou. En suivant le même exemple, on pourrait seulement enregistrer le couple c_1, r_1, et ainsi déclarer tabou tout mouvement qui consisterait à insérer le client c_1 dans la tournée r_1 (cette fois-ci

peut importe la tournée d'où il a été déplacé). Une autre manière encore plus forte d'enregistrer le mouvement tabou serait de simplement mémoriser c_1 et de rendre tabou tout mouvement qui fait bouger le client c_1.

Par ailleurs, si différents types de mouvements sont utilisés dans le voisinage, il est possible de mémoriser plusieurs listes tabou : une pour chaque type de mouvement. Une liste tabou a une taille fixe, dont il faut fixer la taille en faisant quelques tests.

2.4 Critère d'aspiration

Bien que ce soit l'élément central de la recherche tabou, les mouvements tabous sont parfois trop puissants : ils peuvent empêcher de réaliser des mouvements intéressants, sans risque de cycler ; ou ils peuvent entrainer une stagnation du processus de recherche.

Il est donc nécessaire de pouvoir autoriser, quand c'est utile, de pouvoir implémenter des mouvements tabous. C'est ce que l'on appelle un *critère d'aspiration*. Le critère d'aspiration le plus simple et le plus couramment utilisé consiste à autoriser un mouvement tabou dans le cas où ce mouvement va permettre d'améliorer la valeur de la meilleure solution connue. En effet, ceci a du sens, car si on améliore la valeur de la meilleure solution connue, c'est que l'on a trouvé une solution qui n'avait jamais été visitée auparavant dans la recherche.

2.5 Critère d'arrêt

La méthode tabou est une méthode itérative, qui peut continuer indéfiniment si on ne connait pas à l'avance la solution optimale... Et, en général, on ne connait pas une solution optimale. En pratique, la recherche est évidemment stoppée à un certain point. Les critères d'arrêt les plus utilisés dans la recherche tabou sont :

- après un certain nombre d'itérations ;
- après une certaine quantité de temps CPU ;
- après un certain nombre d'itérations sans amélioration de la solution (le critère le plus souvent utilisé) ;
- quand la valeur de l'objectif atteint un seuil prédéfini.

2.6 Schéma d'une recherche tabou

À partir des différents éléments qui ont été présentés précédemment, nous proposons ici un schéma général pour l'algorithme de recherche tabou. Nous supposons que nous cherchons à minimiser une fonction $f(\mathcal{S})$, où $\mathcal{S}$ représente une solution réalisable du problème. Nous utilisons les notations suivantes :

- $\mathcal{S}$: la solution courante ;
- $\mathcal{S}^*$: la meilleure solution connue (à l'itération courante) ;
- f^* : la valeur de la solution $\mathcal{S}^*$;
- $\mathcal{O}(\mathcal{S})$: le voisinage de $\mathcal{S}$, i.e. l'ensemble des solutions voisines de $\mathcal{S}$ par rapport à un ensemble d'opérateurs ;
- $\widetilde{\mathcal{O}}(\mathcal{S})$: l'ensemble des solutions de $\mathcal{O}(\mathcal{S})$ qui sont admissibles, i.e. des solutions qui ne sont pas tabous ou qui sont autorisées d'après le critère d'aspiration ;
- T la liste tabou.

Le pseudo-code de la recherche tabou est donné dans l'Algorithme 1.

Algorithme 1 : Schéma d'une recherche tabou.

```
1: construire une solution initiale S_0 (par exemple avec une méthode
   constructive et une recherche locale)
2: S = S_0
3: f* = f(S_0)
4: S* = S_0
5: T = ∅
6: while critère d'arrêt non satisfait do
7:    S̄ = solution de Õ(S) telle que f(S̄) soit minimale
8:    ajouter dans T le mouvement qui a permis de passer de S à S̄
9:    enlever le plus ancien mouvement dans T si la taille est dépassée
10:   S = S̄
11:   if f(S) < f* then
12:      f* = f(S)
13:      S* = S
14:   end if
15: end while
```

Notez que l'Algorithme 1 présente un schéma de recherche tabou à des fins pédagogiques pour comprendre ce que signifie une recherche tabou. Mais nous n'implémenterons pas directement cet algorithme dans notre code. Nous nous attacherons à mettre en place les éléments de la recherche tabou en modifiant le moins possible le code déjà développé, et en évitant de réécrire du code redondant.

3 Implémentation de la recherche tabou

Dans cette section, nous allons proposer un ensemble de modifications à apporter à votre code afin de mettre en place un algorithme de recherche tabou. Nous allons mettre en place les éléments suivants :

- définition des mouvements tabous pour chacun des opérateurs de recherche locale ;
- mise en place de la liste tabou ;
- modification de la la détermination du meilleur opérateur dans une tournée pour prendre en compte la liste tabou ;
- copie en profondeur d'une solution ;
- mise en place de l'algorithme de recherche tabou ;
- ajout du critère d'aspiration.

3.1 Mouvements tabous

Dans cette section, nous allons nous intéresser à la définition des mouvements tabous liés à chacun des opérateurs de recherche locale que nous avons implémenté. Pour ce faire, nous allons définir dans la classe abstraite **OperateurLocal** une méthode abstraite **isTabou(OperateurLocal operateur)** qui renvoie un booléen indiquant si l'opérateur passé en paramètre est un mouvement tabou par rapport au mouvement associé à l'opérateur courant **this**. Par la suite, cette méthode abstraite sera implémentée dans chacune des classes concrètes liées aux opérateurs de recherche locale.

Question 1. Dans la classe **OperateurLocal**, ajoutez une méthode abstraite **isTabou(OperateurLocal operateur)** qui renvoie un booléen.

Question 2. Dans la classe **IntraDeplacement**, implémentez la méthode **isTabou(OperateurLocal operateur)** de telle sorte que l'opérateur passé en paramètre est tabou si les trois conditions suivantes sont réunies :

- **operateur** est de type **IntraDeplacement** ;
- les tournées sur lesquelles s'appliquent l'opérateur courant **this** et **operateur** sont identiques (attribut **tournee**) ;
- les clients déplacés par l'opérateur de déplacement intra-tournée sont identiques (attribut **clientI**).

Notez que nous proposons ici de définir comme tabou le déplacement d'un même client dans une même tournée, peu importe à quel endroit il est déplacé. Comme expliqué dans la Section 2.3, d'autres définitions seraient possibles.

Question 3. Dans la classe **IntraEchange**, implémentez la méthode **isTabou(OperateurLocal operateur)** de telle sorte que l'opérateur passé en paramètre est tabou si les trois conditions suivantes sont réunies :

- **operateur** est de type **IntraEchange** ;
- les tournées sur lesquelles s'appliquent l'opérateur courant **this** et **operateur** sont identiques (attribut **tournee**) ;
- les clients échangés par l'opérateur d'échange intra-tournée sont identiques (attributs **clientI** et **clientJ**).

Notez bien que l'échange de deux clients est symétrique : échanger **clientI** avec **clientJ** est identique à échanger **clientJ** avec **clientI**. Donc, pour définir le mouvement tabou, il faut bien regarder dans les deux tournées si les paires $\{\textbf{clientI}, \textbf{clientJ}\}$ sont identiques. Pour rappel, en mathématique, la paire $\{i, j\}$ est identique à la paire $\{j, i\}$.

Question 4. Dans la classe **InterDeplacement**, implémentez la méthode **isTabou(OperateurLocal operateur)** de telle sorte que l'opérateur passé en paramètre est tabou si les deux conditions suivantes sont réunies :

- **operateur** est de type **InterDeplacement** ;
- les clients déplacés par l'opérateur de déplacement inter-tournées sont identiques (attribut **clientI**).

Notez que nous proposons ici de définir comme tabou le déplacement d'un même client dans une autre tournée, peu importe à quel endroit il est déplacé, de quelle tournée il est supprimé, et dans quelle tournée il est ajouté. Comme expliqué dans la Section 2.3, d'autres définitions seraient possibles.

Question 5. Dans la classe **InterEchange**, implémentez la méthode **isTabou(OperateurLocal operateur)** de telle sorte que l'opérateur passé en paramètre est tabou si les deux conditions suivantes sont réunies :

- **operateur** est de type **InterEchange** ;
- au moins un des clients échangés par l'opérateur d'échange inter-tournées est identique dans les deux tournées (attributs **clientI** ou **clientJ**).

Notez bien ici qu'il suffit qu'au moins un client soit commun aux deux paires de clients échangés pour que le mouvement soit déclaré tabou.

Question 6. Modifiez les tests des classes **TestIntraDeplacement**, **TestIntraEchange**, **TestInterDeplacement** et **TestInterEchange** afin de tester que la méthode **isTabou(OperateurLocal operateur)** est correctement implémentée dans chacune des classes où vous avez implémenté cette méthode.

3.2 Liste tabou

Maintenant que nous avons défini les mouvements tabous, nous allons nous intéresser à la mise en place de la liste tabou. Cette liste est en fait une file de type FIFO (*First In First Out*) de taille fixe dans laquelle l'élément entré en premier sort de la file si la taille est atteinte. En Java, une file FIFO de taille fixe implémente l'interface **BlockingQueue**. Une implémentation que nous pouvons utiliser ici est la classe **LinkedBlockingQueue** qui se base sur une implémentation de type liste chaînée. Cette classe possède en particulier :

- un constructeur **LinkedBlockingQueue(int capacity)** qui permet de définir la capacité de la file ;
- une méthode **offer(E e)** qui insère l'élément **e** à la fin de la queue si c'est possible, et renvoie un booléen qui indique si l'insertion a pu avoir lieu ;
- une méthode **poll()** qui retourne et supprime l'élément en tête de la file.

Évidemment, vous êtes fortement encouragés à consulter la Javadoc afin d'avoir plus d'informations sur la classe **LinkedBlockingQueue** et ses méthodes.

La liste tabou contiendra des éléments de type **OperateurLocal**, et possédera des méthodes permettant d'ajouter un opérateur et de savoir si un opérateur représente un mouvement tabou par rapport aux opérateurs actuellement dans la liste.

Par ailleurs, notez que nous souhaitons que cette liste tabou soit unique, et qu'il y en ait une seule instance pour notre code. En effet, il n'y a aucune raison d'avoir en même temps plusieurs instances de listes tabous. Ainsi, nous mettrons en place le *design pattern* singleton pour nous assurer que l'instance de la liste tabou soit unique. Vous pouvez vous référer au TP2 du cours de LE3-POO si vous ne vous rappelez plus ce qu'est le *design pattern* singleton.

Question 7. Dans le paquetage **operateur**, ajoutez une classe **ListeTabou** qui permettra de représenter la liste tabou. Définissez ses attributs, et ajoutez un constructeur par défaut ainsi que les accesseurs et mutateurs nécessaires et une redéfinition de la méthode **toString()**.

Pour la taille fixe de la file FIFO, vous pouvez prendre par exemple la valeur 80. Vous pourrez par la suite faire des tests pour modifier cette valeur si besoin.

Question 8. Ajoutez dans la classe **ListeTabou** une méthode **add(OperateurLocal operateur)** qui ajoute l'opérateur passé en paramètre à la queue de la file, en s'assurant que la taille ne soit pas dépassée. L'élément en tête de la file peut être supprimé afin de garantir que la taille ne soit pas dépassée.

Question 9. Dans la classe **ListeTabou**, ajoutez une méthode **isTabou(OperateurLocal operateur)** qui renvoie un booléen indiquant si l'opérateur passé en paramètre correspond à un mouvement tabou par rapport aux opérateurs présents dans la file.

Remarquez à nouveau ici le grand intérêt de l'héritage et des méthodes abstraites. On traite les opérateurs de la liste tabou de la même manière sans connaître leur classe concrète.

Question 10. Dans la classe **ListeTabou**, ajoutez une méthode **vider()** qui vide la file.

Cette méthode permettra de s'assurer que la liste tabou est bien vide quand on souhaite faire une recherche locale classique sans prise en compte des mouvements tabous.

Question 11. Mettez en place le *design pattern* singleton dans la classe **ListeTabou** afin qu'on ne puisse créer qu'une seule instance de type **ListeTabou**.

3.3 Meilleur opérateur non tabou

À présent que nous disposons d'une liste tabou, nous allons modifier notre code afin de pouvoir récupérer le meilleur opérateur non tabou dans une solution. Si votre code est bien fait, il y a juste deux méthodes de la classe **Tournee** à modifier.

Question 12. Dans la classe **Tournee**, modifiez les méthodes **getMeilleurOperateurIntra(TypeOperateurLocal type)** et **getMeilleurOperateurInter(Tournee autreTournee, TypeOperateurLocal type)** afin que ces méthodes renvoient le meilleur opérateur non tabou intra-tournée et inter-tournées respectivement.

Remarquez ici l'avantage d'avoir mis en place le *design pattern* singleton pour la liste tabou. Vous pouvez avoir accès, de manière statique, à l'instance de la classe **ListeTabou**. Ainsi, vous évitez de passer la liste tabou en paramètre de toutes les méthodes appelantes.

Question 13. Dans la classe **RechercheLocale**, au début de la méthode **solve()**, videz la liste tabou afin d'être certain que la recherche locale ne prenne pas en compte les mouvements tabous. Vérifiez que vos résultats de recherche locale n'ont pas changé.

3.4 Copie en profondeur d'une solution

Comme dans une recherche tabou nous appliquons des opérateurs qui peuvent dégrader la solution, nous devons garder en mémoire la meilleure solution connue. Pour ce faire, il faut faire une copie en profondeur de la solution. Ceci se

traduit par la mise en place d'un constructeur par copie dans la classe **Solution**. Les tournées qui composent la solution doivent elles aussi être copiées en profondeur à l'aide d'un constructeur par copie dans la classe **Tournee**.

Question 14. Dans la classe **Tournee**, ajoutez un constructeur par copie qui réalise une copie en profondeur de la tournée.

L'important ici est de bien faire une copie de la liste, et pas seulement de sauvegarder la référence sur la liste. Rappelez-vous que la classe **LinkedList** permet de construire une nouvelle liste par copie. Cela est suffisant dans notre cas car nous n'avons pas besoin de copier les clients contenus dans la tournée.

Question 15. Dans la classe **Solution**, ajoutez un constructeur par copie qui réalise une copie en profondeur de la solution.

Testez dans la méthode principale de la classe **Solution** que la solution copiée est bien valide et qu'un changement sur la solution n'entraine pas de modifications sur la copie.

L'important ici est de bien faire une copie en profondeur de la liste des tournées. Il faut donc créer une nouvelle liste et y ajouter une copie de chaque tournée contenue dans la liste.

3.5 Algorithme de recherche tabou

Tous les éléments sont à présent en place pour implémenter la recherche tabou. Un pseudo-code de l'algorithme de recherche tabou est proposé dans l'Algorithme 2. Nous proposons ici un critère d'arrêt (Ligne 7) basé sur le nombre consécutif d'itérations sans amélioration de la meilleure solution. La valeur maximale est fixée à 10000 (Ligne 4). Le meilleur opérateur *best* est initialisé avec un des opérateurs par défaut (Ligne 8). Pour chaque type d'opérateur, on cherche le meilleur opérateur non tabou dans la solution courante $\mathcal{S}$ avec la méthode **getMeilleurOperateurLocal(TypeOperateurLocal type)** de la classe **Solution** (Ligne 10). Par la suite, on essaye d'implémenter le meilleur opérateur de recherche locale non tabou sur la solution courante $\mathcal{S}$ (Ligne 15) en appliquant la méthode **doMouvementRechercheLocale(OperateurLocal infos)** de la classe **Solution**. Enfin, si on arrive à améliorer la meilleure solution connue $\mathcal{S}_{best}$, on met cette dernière à jour avec le constructeur par copie de la classe **Solution** (Ligne 19).

Question 16. Dans le paquetage **solveur**, ajoutez une classe **RechercheTabou** qui implémente l'interface **Solveur**. Ajoutez un attribut qui représente le solveur utilisé pour obtenir une solution initiale. Ajoutez un constructeur par la donnée de ce solveur. Implémentez les méthodes de l'interface **Solveur**. En particulier, la méthode **solve** devra avoir le comportement de l'algorithme de recherche tabou (Algorithme 2).

Testez que votre code fonctionne correctement.

Algorithme 2 : Implémentation de l'algorithme de recherche tabou.

1: $\mathcal{S}$ = solution initiale (e.g. après application d'une méthode constructive et d'une recherche locale)
2: $\mathcal{S}_{best} = \mathcal{S}$
3: $\mathcal{OPER} = \{\mathcal{O}_{intra-dep}, \mathcal{O}_{intra-ech}, \mathcal{O}_{inter-dep}, \mathcal{O}_{inter-ech}\}$
4: $nbIterMax = 10000$
5: $nbIterSansAmelioration = 0$
6: vider la liste tabou
7: **while** $nbIterSansAmelioration < nbIterMax$ **do**
8: $best$ = opérateur local par défaut
9: **for all** $\mathcal{O} \in \mathcal{OPER}$ **do**
10: op = meilleur opérateur non tabou de type $\mathcal{O}$ dans $\mathcal{S}$
11: **if** op est meilleur que $best$ **then**
12: $best = op$
13: **end if**
14: **end for**
15: **if** le mouvement lié à $best$ est implémenté dans $\mathcal{S}$ **then**
16: ajouter $best$ à la liste tabou
17: **end if**
18: **if** $\mathcal{S}$ a un coût inférieur à $\mathcal{S}_{best}$ **then**
19: $\mathcal{S}_{best} = \mathcal{S}$
20: $nbIterSansAmelioration = 0$
21: **else**
22: $nbIterSansAmelioration = nbIterSansAmelioration + 1$
23: **end if**
24: **end while**
25: **return** $\mathcal{S}_{best}$

Question 17. Afin de bien tester l'algorithme de recherche tabou et de pouvoir évaluer ses performances, nous allons ajouter ce solveur dans le test avec la classe **TestAllSolveur**. Dans la classe **TestAllSolveur**, dans la méthode **addSolveurs()**, ajoutez dans la liste des solveurs une instance du solveur **RechercheTabou**.

Testez donc que vous obtenez bien une solution valide sur toutes les instances, et comparez les coûts de la solution à ceux obtenus avec les autres méthodes.

Ne vous inquiétez pas s'il faut quelques minutes pour obtenir les résultats. Avec un critère d'arrêt à 10000 itérations sans amélioration, cela peut prendre un peu de temps. Au delà de 15 minutes, c'est probablement qu'il y a un soucis dans votre code, et il faut alors certainement tester les instances une par une.

3.6 Critère d'aspiration

Afin d'améliorer les résultats obtenus par la recherche tabou, nous pouvons mettre en place le critère d'aspiration, comme expliqué dans la Section 2.4. Pour ce faire, nous allons procéder en deux temps :

- dans la classe **ListeTabou**, nous allons ajouter un attribut **deltaAspiration** qui pourra être mis à jour, et tel que si un mouvement engendre un coût inférieur à **deltaAspiration**, alors ce mouvement est automatiquement déclaré non tabou (même s'il est tabou par rapport à un mouvement de la liste tabou) ;
- dans la classe **RechercheTabou**, nous mettrons à jour la valeur de **deltaAspiration** comme étant la différence entre le coût de la meilleure solution connue $\mathcal{S}_{best}$ et le coût de la solution courante $\mathcal{S}$ (la valeur de **deltaAspiration** est donc négative ou nulle).

Question 18. Dans la classe **ListeTabou**, ajoutez un attribut de type entier **deltaAspiration**. Modifiez en conséquence le constructeur par défaut de la classe **ListeTabou**, et ajoutez un mutateur pour cet attribut.

Enfin, modifiez la méthode **isTabou(OperateurLocal operateur)** pour faire en sorte que si le coût supplémentaire de l'opérateur (attribut **deltaCout**) est strictement inférieur à **deltaAspiration**, alors la méthode renvoie **false** (le mouvement associé à **operateur** est automatiquement déclaré non tabou car il améliore la meilleure solution connue).

Question 19. Relancez les tests de la classe **TestAllSolveur**, et vérifiez que vous obtenez bien une solution valide sur toutes les instances, et comparez les coûts de la solution à ceux obtenus précédemment sans le critère d'aspiration.

4 Algorithme génétique

Pour les personnes qui seraient intéressées par l'implémentation d'un algorithme génétique sur le problème de CVRP, nous vous invitons à aller consulter l'article de Prins [3]. Si vous souhaitez des informations supplémentaires, n'hésitez pas à contacter les enseignants.

References

[1] Gendreau, M., Potvin, J.Y., *Handbook of metaheuristics. Vol. 2. Springer. 2010.*

[2] Glover, F., *Future paths for integer programming and links to artificial intelligence. Computers & Operations Research, 13(45), 553-549. 1986.*

[3] Prins, C., *A simple and effective evolutionary algorithm for the vehicle routing problem. Computers & Operations Research, 31(12), 1985-2002. 2004.*