Programmation Orientée Objet
IG2I, LE4 - SI – 2017-2018

TP 6 – Métaheuristiques

Diego Cattaruzza, Maxime Ogier

Contexte et objectif du TP

Dans ce TP, nous nous intéressons encore à la livraison des caisses de bières de l'entreprise Bière2I. Dans le dernier TP, vous avez implémenté un algorithme de recherche locale afin de pouvoir améliorer une solution réalisable du problème. Êtes-vous satisfaits avec la qualité des solutions obtenues ? Peut-être que votre réponse est oui. Malheureusement Monsieur Houblon ne l'est toujours pas. Il vous demande de lui proposer des solutions de meilleure qualité.

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.

Bonnes pratiques à adopter

N'oubliez pas de tester votre code au fur et à mesure de votre avancement. Par ailleurs, il vous est très fortement recommandé d'utiliser la Javadoc pour comprendre le fonctionnement des méthodes que vous appelez. Il est également important que votre propre code soit commenté au format Javadoc (commentaire commençant par "`/**`" au-dessus des définitions des attributs et méthodes). Ceci vous permet (1) de spécifier rigoureusement le comportement d'une méthode au moment où vous l'implémentez, (2) de savoir ce que fait une méthode sans avoir besoin de regarder son code (qui d'ailleurs peut ne pas être accessible), et donc (3) de faciliter le partage et la réutilisabilité du code.

1 Introduction

Supposons que vous obtenez une solution s avec une des méthodes constructives et l'application d'une recherche locale codée lors du dernier TP. La solution s que vous avez obtenue appartient à l'ensemble $\mathcal{S}$ de toutes les solutions de votre problème de tournées de véhicules (dans ce contexte, mais ce concept s'applique à n'importe quel problème d'optimisation). La solution 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 s que vous avez obtenue est un minimum local (i.e. aucune solution voisine de 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 $\mathcal{S}$ étant très grand (de l'ordre 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 s, l'application de la recherche locale permet de fournir une solution s' voisine de s de meilleure qualité. L'idée des métaheuristiques est de permettre également de *diversifier* la recherche dans l'ensemble de solutions $\mathcal{S}$: étant donné une solution s, il faut pouvoir fournir une solution s' qui soit "différente" (i.e. pas voisine) de 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.

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, et 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 rechecrhe 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 s on peut passer à une solution s' même si s' a un coût plus élevé que 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.

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. En effet, il est important de voir que si d'une solution s on passe à une solution voisine s' telle que le coût de s' est supérieur au coût de s, alors à l'itération suivante, il est fort probable que la meilleure solution voisine de s' soit la solution s. Il faut donc éviter de tels cycles.

Afin d'éviter de cycler, on des 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 la dizaine). 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 améliorer 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... 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 $f(s)$, où s représente une solution réalisable du problème. Nous utilisons les notations suivantes :

- s : la solution courante ;
- s^* : la meilleure solution connue (à l'itération courante) ;
- f^* : la valeur de la solution s^* ;
- $\mathcal{O}(s)$: le voisinage de s, i.e. l'ensemble des solutions voisines de s par rapport à un ensemble d'opérateurs ;
- $\widetilde{\mathcal{O}}(s)$: l'ensemble des solutions de $\mathcal{O}(s)$ 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.

Algorithm 1 Schéma d'une recherche tabou

construire une solution initiale s_0 (par exemple avec une méthode constructive et une recherche locale)
$s = s_0$
$f^* = f(s_0)$
$s^* = s_0$
$T = \emptyset$
while critère d'arrêt non satisfait **do**
  $\bar{s}$ = solution de $\widetilde{\mathcal{O}}(s)$ telle que $f(\bar{s})$ soit minimale
  ajouter dans T le mouvement qui a permis de passer de s à $\bar{s}$
  enlever le plus ancien mouvement dans T si la taille est dépassée
  $s = \bar{s}$
  if $f(s) < f^*$ **then**
    $f^* = f(s)$
    $s^* = s$
  end if
end while

3 Implémentation de la recherche tabou

Dans cette section, nous allons donner quelques éléments pour implémenter la recherche tabou. Il vous revient d'adapter cela par rapport au code que vous avez déjà développé.

Question 1. Avant d'implémenter la recherche tabou, nous pouvons remarquer que nous aurons besoin de mémoriser la meilleure solution connue (s^*). Pour faire ceci, il est nécessaire de faire une copie en profondeur de l'objet **Planning**. La copie en profondeur signifie qu'il faudra faire une copie des **Vehicule** contenus dans l'ensemble de véhicules. Dans la classe **Planning**, ajoutez une méthode **public Planning getCopie()** qui renvoie une copie du planning. Vous pouvez vous aider d'une méthode **public Vehicule getCopie()** dans la classe **Vehicule**. Enfin, dans la classe **Instance**, ajoutez une méthode **public Planning getCopiePlanning()** qui renvoie une copie du planning lié à l'instance.

Question 2. Nous pouvons également remarquer qu'à la fin de la recherche tabou, il faudra mémoriser le meilleur planning. Pour ce faire, ajouter dans la classe **Instance** une méthode **public boolean setPlanning(Planning p)** qui modifie le planning lié à l'instance. Cette méthode retourne **true** si le planning est correct et a bien été affecté à l'instance, et **false** sinon. Pensez bien à mettre à jour les positions des clients dans les véhicules, et à utiliser votre *checker* pour vérifier que le planning est correct. Pensez également à mettre à jour la liste des véhicules dans l'instance.

Question 3. Dans le paquetage **algo**, vous pouvez rajouter une classe **MouvementTabou** qui représente un mouvement tabou. Remarquez qu'un mou-

vement tabou n'est pas quelque chose de très différent des informations sur les mouvements intra-tournée ou inter-tournées.

Question 4. Dans le paquetage **algo**, ajoutez une classe **RechercheTabou**, avec en attributs un objet de type **Instance**, et une liste de **MouvementTabou**. Ajoutez un constructeur par donnée de l'instance, et une méthode **public void rechercheTabou()**. N'hésitez pas à ajouter d'autres méthodes afin de garder un code lisible. Vous pourrez supposer que la classe **Instance** contient une méthode **public MouvementTabou bestMove(List<MouvementTabou> tabuList)** qui renvoie le mouvement qui a été effectué afin de trouver une solution du voisinage ($\widetilde{\mathcal{O}}(s)$) qui soit de plus petit coût possible.

Question 5. À présent, ajoutez dans la classe **Planning** une méthode **public MouvementTabou bestMove(List<MouvementTabou> tabuList)**. Cette méthode sera appelée par la méthode du même nom dans la classe **Instance**. Dans un premier temps, considérez un voisinage avec un seul mouvement (le déplacement intra-tournée) : il faut chercher dans ce voisinage le mouvement qui ne soit pas tabou et qui donne la meilleure solution (même si le surcoût est positif). Ensuite, on implémente effectivement ce mouvement. L'idée n'est pas très différente de ce que vous avez fait lors de la recherche locale : il faut donc au maximum réutiliser votre code (mais sans faire d'affreux copier-collers). En fait la tâche la plus délicate est de bien vérifier que vous ne faites pas de mouvement tabou. Faites un test pour vérifier que votre code fonctionne.

Question 6. Pour le moment, nous n'avons pas pris en compte le critère d'aspiration. Modifiez votre code afin de prendre en compte cet élément. Faites un test pour vérifier que votre code fonctionne.

Question 7. Vous pouvez à présent rajouter dans la méthode **bestMove** de la classe **Planning** la recherche dans les autres voisinages que vous avez implémenté. Il est conseillé de d'abord tester chaque voisinage un par un pour vérifier que la recherche tabou ne cycle pas. Faites un test pour vérifier que votre code fonctionne. Quels résultats obtenez-vous ? Arrivez-vous à améliorer les résultats de la recherche locale ?

Question 8. Maintenant, il ne reste plus qu'à faire quelques paramétrages :

- la taille de la liste tabou ;
- le critère d'arrêt ;
- utiliser des listes différentes selon le type de mouvement.

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.*