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

TP 5 – Recherche Locale

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 construit des solution réalisables pour livrer les clients de l'entreprise. Êtes-vous satisfaits avec la qualité des ces solutions ? Peut-être que votre réponse est oui. Malheureusement Monsieur Houblon ne l'est pas. Il vous demande de lui proposer des solutions de meilleure qualité.

L'objectif de ce TP est donc de concevoir des algorithmes d'amélioration d'une solution. Ces algorithmes prennent en entrée une solution et essayent de l'améliorer.

Une façon classique de réaliser cela est de soumettre une solution à ce que l'on appelle une *recherche locale*. L'idée de la recherche locale est la suivante. Étant donné une solution s, la recherche locale analyse les solutions *voisines* de s. Intuitivement une solution s' est voisine d'une solution s si elle diffère *légèrement* de s. Le but de la recherche locale est de trouver, parmi toutes les solution s' *légèrement* différentes de s, une de meilleur qualité (si une telle solution existe).

Ce TP permet d'illustrer les notions suivantes :

- le concept de transformation locale (autrement dit opérateur) ;
- les opérateurs d'amélioration *intra-tournée* ;
- les opérateurs d'amélioration *inter-tournée* ;
- la notion de recherche locale.

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 codées 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. Dans ce TP nous allons nous intéresser à la manière d'améliorer la qualité de cette solution.

Nous allons appeler *opérateur* (et nous allons l'indiquer avec $\mathcal{O}$) une fonction qui modifie une solution selon un comportement bien défini. Un exemple d'opérateur peut être la fonction qui déplace un client dans une tournée à une position différente de celle couramment occupée.

On indique avec $\mathcal{O}(s)$ toutes les solutions qui peuvent être obtenues à partir de la solution s en appliquant l'opérateur $\mathcal{O}$. Nous avons bien évidement $\mathcal{O}(s) \subseteq \mathcal{S}$. L'ensemble $\mathcal{O}(s)$ est ce que l'on appelle un voisinage de la solution s. Nous disons alors qu'une solution $z \in \mathcal{O}(s)$ est une solution *voisine* de s.

Il faut bien remarquer que l'ensemble $\mathcal{O}(s)$ est dépendant de la définition de l'opérateur $\mathcal{O}$; et en général avec deux opérateurs différents $\mathcal{O}_1$ et $\mathcal{O}_2$, nous obtenons deux voisinages $\mathcal{O}_1(s)$ et $\mathcal{O}_2(s)$ de la solution s potentiellement différents.

Nous allons nous concentrer dans un premier temps sur l'*exploration* d'un voisinage $\mathcal{O}(s)$ d'une solution s. L'exploration d'un voisinage $\mathcal{O}(s)$ consiste à évaluer toutes les solutions $z \in \mathcal{O}(s)$ et à trouver celle avec le moindre coût. L'espoir est de trouver une solution z avec un coût plus petit que le coût de s. Remarquez que cela n'est pas toujours possible : si la solution s est la solution optimale, vous ne pourrez jamais en trouver une autre avec un coût plus faible !

Dans un second temps, nous allons voir comment enchainer l'exploration de différents voisinages pour obtenir un algorithme de recherche locale.

2 Opérateurs intra-tournée

Dans cette section nous allons nous concentrer sur l'optimisation d'une seule tournée de notre solution s obtenue grâce aux méthodes constructives du TP précédent.

2.1 Opérateur de déplacement

L'opérateur de déplacement intra-tournée, indiqué par $\mathcal{O}_{intra-dep}$ considère chaque tournée de la solution courante s et évalue le déplacement de tout client dans une position différente de la même tournée. S'il existe un déplacement tel que la solution s est améliorée, alors ce déplacement est exécuté.

Le voisinage $\mathcal{O}_{intra-dep}(s)$ contient $\sum_{k\in\mathcal{K}} r_k \cdot r_{k-1}$ solutions différentes, où r_k est le nombre de clients desservis par la tournée k, et $\mathcal{K}$ est l'ensemble des véhicules disponibles. Le pseudo-code de l'opérateur $\mathcal{O}_{intra-dep}$ est donné dans l'Algorithme 1.

Algorithm 1 Opérateur de déplacement

Require: Solution s, véhicule/tournée k
$\Delta cout = +\infty$
for all $k \in \mathcal{K}$ **do**
  $size_k$ = nombre de clients desservis par k
  for all client c desservi par le véhicule k **do**
    pos_c = position du client c dans le véhicule k
    for all $pos = 1, \ldots, size_k$, $pos \neq pos_c$ **do**
      $\delta cout$ = evaluerDeplacement(s, k, c, pos);
      if $\delta cout < \Delta cout$ **then**
        $\Delta cout = \delta cout$, $best_k = k$, $best_c = c$, $best_{pos} = pos$
      end if
    end for
  end for
end for
if $\Delta cout < 0$ **then**
  implementerDeplacement(s, $best_k$, $best_c$, $best_{pos}$)
  return true
end if
return false

Question 1. Dans le paquetage **algo** de votre projet, ajoutez une classe **IntraTourneeInfos**. Cette classe contient les informations liées à un opérateur dans une même tournée. Elle a donc un attribut de type **Vehicule**, deux attributs de type **int** (la position à laquelle le client était, et la position à laquelle le client est déplacé), et un attribut de type **double** qui contient la différence de coût entre la solution initiale et la solution après le déplacement. Ajoutez-y un constructeur par défaut, un constructeur par données, et un constructeur par copie.

Question 2. Dans le paquetage **algo** de votre projet, ajoutez une classe **RechercheLocale**. Cette classe a un attribut de type **Instance**. Ajoutez-y un constructeur par donnée de l'instance et une méthode **private boolean deplacementIntraVehicule()**. Cette méthode vérifie d'abord que l'objet planning de l'instance ne soit pas **null** (ceci est une méthode améliorante, pas une méthode constructive, donc nous devons avoir déjà calculé un planning). Ensuite, cette méthode appelle la méthode (pas encore codée) **public boolean deplacementIntraVehicule()** de la classe **Planning**. Enfin, la méthode renvoie **true** si le planning a été amélioré, et **false** sinon.

Question 3. Dans la classe **Planning** ajoutez une méthode **public boolean deplacementIntraVehicule()**. Cette méthode parcourt les véhicules du planning et, pour chaque véhicule, évalue le déplacement de chaque client dans une position différente. Pour ce faire, cette méthode appelle la méthode (pas encore codée) **public IntraTourneeInfos deplacementIntraVehicule()** de la classe **Vehicule**. Ensuite, la méthode exécute/implémente le déplacement qui permet d'améliorer le plus la solution courante. Pour ce faire, nous faisons appel à la méthode (pas encore codée) **public boolean doDeplacementIntraTournee()** de la classe **IntraTourneeInfos**.

Question 4. Dans la classe **Vehicule**, ajoutez une méthode **public IntraTourneeInfos deplacementIntraVehicule()** qui évalue le déplacement de chaque client dans la tournée dans une autre position. La méthode renvoie un objet de type **IntraTourneeInfos** associé au meilleur déplacement possible. N'hésitez pas à ajouter d'autres méthodes pour rendre votre code plus lisible. Notez bien que pour évaluer le coût du déplacement d'un client à une autre position, il n'est pas nécessaire d'effectuer le déplacement. Faites des dessins pour vous aider.

Question 5. Dans la classe **IntraTourneeInfos**, implémentez la méthode **public boolean doDeplacementIntraTournee()** qui effectue le déplacement, à partir des informations contenues dans l'objet, si cela permet de diminuer le coût. N'hésitez pas à ajouter d'autres méthodes (dans la classe **Vehicule** par exemple) pour rendre votre code plus lisible. Testez le bon fonctionnement de votre opérateur.

Question 6. Ajoutez à votre classe **RechercheLocale** une méthode principale. Faites en sorte que l'opérateur de déplacement intra-tournée soit appelé tant que la solution est améliorée par cet opérateur. Vérifiez sur les différentes instances si vous arrivez à améliorer les solutions obtenues par les méthodes constructives.

Question 7. Remarquez-vous la possibilité d'améliorer l'efficacité de votre implémentation de l'opérateur de déplacement intra-tournée ?

2.2 Opérateur d'échange

L'opérateur d'échange intra-tournée, indiqué par $\mathcal{O}_{intra-ech}$ considère chaque couple de clients c_1, c_2 d'une même tournée de la solution courante s et évalue l'échange de leurs positions. S'il existe un échange tel que la solution s est améliorée, alors cet échange est exécuté.

Le voisinage $\mathcal{O}_{intra-ech}(s)$ contient $\sum_{k\in\mathcal{K}} r_k \cdot r_{k-1}$ solutions différentes, où r_k est le nombre de clients desservis par la tournée k, et $\mathcal{K}$ est l'ensemble des véhicules disponibles. Le pseudo-code de l'opérateur $\mathcal{O}_{intra-ech}$ est donné dans l'Algorithme 2.

Algorithm 2 Opérateur d'echange

Require: Solution s, véhicule/tournée k
$\Delta cout = +\infty$
for all $k \in \mathcal{K}$ **do**
  for all client c_1 desservi par le véhicule k **do**
    for all client $c_2 \neq c_1$ desservi par le véhicule k **do**
      $\delta cout$ = evaluerEchange(s, k, c_1, c_2);
      if $\delta cout < \Delta cout$ **then**
        $\Delta cout = \delta cout$, $best_k = k$, $best_{c_1} = c_1$, $best_{c_2} = c_2$
      end if
    end for
  end for
end for
if $\Delta cout < 0$ **then**
  implementerEchange(s, $best_k$, $best_{c_1}$, $best_{c_2}$)
  return true
end if
return false

Question 8. Implémentez l'opérateur d'échange intra-tournée. Ajoutez à votre projet les classes et les méthodes que vous jugez nécessaires.

2.3 Opérateur 2-opt

L'opérateur 2-opt, indiqué par $\mathcal{O}_{intra-2opt}$ considère deux routes non-consécutives r_1 et r_2 de la même tournée. On note d_1 et a_1 les points de départ et d'arrivée de la route r_1, et on note d_2 et a_2 les points de départ et d'arrivée de la route r_2. Ces points peuvent être le dépôt.

L'opérateur $\mathcal{O}_{intra-2opt}$ remplace r_1 et r_2 par deux autres routes r_3 et r_4, telles que r_3 part de d_1 et arrive en d_2, tandis que r_4 part de a_1 et arrive en a_2. Remarquez que dans la nouvelle tournée, les points entre d_2 et a_1 sont maintenant visités dans l'ordre inverse. Un exemple est donné en Figure 1.

S'il existe une telle opération de remplacement des routes telle que la solution s est améliorée, alors ce remplacement est exécuté.

Le voisinage $\mathcal{O}_{intra-2opt}(s)$ contient $\sum_{k\in\mathcal{K}} r_k \cdot r_{k-1}$ solutions différentes, où r_k est le nombre de clients desservis par la tournée k, et $\mathcal{K}$ est l'ensemble des véhicules disponibles. Le pseudo-code de l'opérateur $\mathcal{O}_{intra-2opt}$ est donné dans l'Algorithme 3.

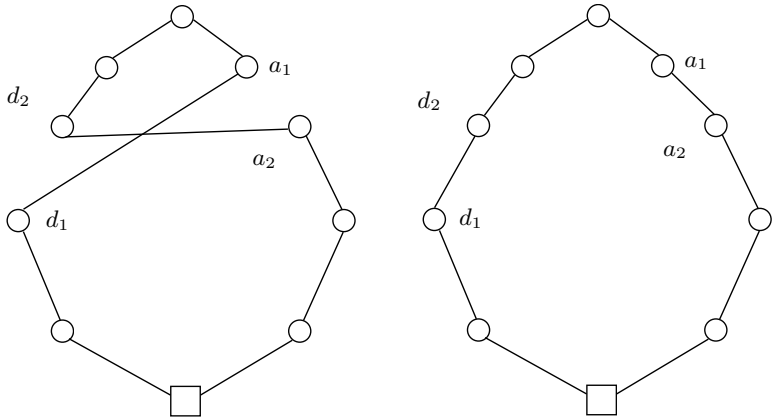

Figure 1: Opérateur 2-opt : à gauche la solution avant implémentation de l'opérateur, à droite après implémentation.

Algorithm 3 Opérateur 2-opt

Require: Solution s, véhicule/tournée k
$\Delta cout = +\infty$
for all $k \in \mathcal{K}$ **do**
  for all route $r_1 = (d_1, a_1)$ parcourue par le véhicule k **do**
    for all route $r_2 = (d_2, a_2) \neq r_1$ et non-consécutive à r_1 parcourue par le véhicule k **do**
      $\delta cout$ = value2opt(s, k, r_1, r_2);
      if $\delta cout < \Delta cout$ **then**
        $\Delta cout = \delta cout$, $best_k = k$ $best_{r_1} = r_1$, $best_{r_2} = r_2$
      end if
    end for
  end for
end for
if $\Delta cout < 0$ **then**
  implementer2opt(s, $best_k$, $best_{r_1}$, $best_{r_2}$)
  return true
end if
return false

Question 9. Implémentez l'opérateur 2-opt. Ajoutez à votre projet les classes et les méthodes que vous jugez nécessaires.

3 Opérateurs inter-tournées

Vous avez bien sûr remarqué que les opérateurs vus dans la section précédente peuvent être étendus ; et le cas où les tournées sont différentes peut être considéré.

- L'opérateur de déplacement inter-tournées $\mathcal{O}_{inter-dep}$ considère le déplacement de chaque client dans toutes les autres tournées et dans toutes les positions de ces tournées.
- L'opérateur d'échange inter-tournées $\mathcal{O}_{inter-ech}$ considère deux clients c_1 et c_2 desservis par deux tournées différentes k_1 et k_2 ; et évalue l'échange de ces deux clients. Il évalue donc la possibilité de placer c_1 dans la position occupée par c_2 dans k_2 et de placer c_2 dans la position occupée par c_1 dans k_1.
- L'opérateur 2-opt inter-tournées (souvent indiqué par 2-opt*) $\mathcal{O}_{inter-2opt}$ considère deux routes $r_1 = (d_1, a_1)$ et $r_2 = (d_2, a_2)$ parcourues par deux tournées différentes k_1 et k_2 et évalue le remplacement de ces deux routes par :
  - $r_3 = (d_1, a_2)$ et $r_4 = (d_2, a_1)$;
  - $r_3 = (d_1, d_2)$ et $r_4 = (a_1, a_2)$.

  Remarquez qu'après implémentation de cet opérateur, des morceaux de tournées sont parcourus dans le sens inverse et que les tournées sont *mélangées*, c'est-à-dire la nouvelle tournée k_1 dessert une partie des clients desservis par l'ancienne tournée k_1 et une partie des clients desservis par l'ancienne tournée k_2. Il en est de même pour la tournée k_2.

Remarquez que maintenant la contrainte de capacité doit être vérifiée à chaque évaluation de chaque opérateur inter-tournées. Si la contrainte de capacité n'est pas vérifiée, il n'est pas utile d'évaluer l'opération.

Question 10. Implémentez les opérateurs inter-tournées de déplacement, d'échange et 2-opt*. Ajoutez à votre projet les classes et les méthodes que vous jugez nécessaires.

4 Recherche locale

Considérons un problème d'optimisation dont la fonction objectif est à minimiser (l'objectif est de trouver la solution qui minimise le coût). Étant donnée une solution s, c_s est son coût.

L'idée de la recherche locale est d'améliorer une solution donnée s en construisant une suite de solutions $(s = s_0, s_1, \ldots, s_n)$ telle que la solution s_{i+1} est une solution voisine de la solution s_i et que $c_{s_{i+1}} < c_{s_i}$. La solution s_n finale de la suite est dite *minimum local*. Un minimum local est une solution s^* telle que il n'existe pas de solution voisine $\bar{s}$ telle que $c_{\bar{s}} < c_{s^*}$. La recherche locale s'arrête alors.

Plusieurs manières d'implémenter la recherche locale existent. Ici, nous proposons d'enchainer la visite des voisinages associés aux opérateurs codés dans les sections précédentes. Notre recherche locale termine quand l'exploration de la suite de tous ces voisinages ne produit pas une meilleure solution par rapport à la solution courante. Le pseudo code est donné dans l'Algorithme 4.

Algorithm 4 Recherche locale

Require: Solution s
$s_0 = s$, $i = 1$
$\mathcal{OPER} = \{\mathcal{O}_{intra-dep}, \mathcal{O}_{intra-ech}, \mathcal{O}_{intra-2opt}, \mathcal{O}_{inter-dep}, \mathcal{O}_{inter-ech}, \mathcal{O}_{inter-2opt}\}$
repeat
  improved = false;
  for all $\mathcal{O} \in \mathcal{OPER}$ **do**
    $\Delta cout = +\infty$
    for all $s \in \mathcal{O}(s_{i-1})$ **do**
      $\delta cout = evaluate(s)$
      if $\delta cout < \Delta cout$ **then**
        $\Delta cout = \delta cout$, $s_{best} = s$
      end if
      if $\Delta cout < 0$ **then**
        improved = true
        $s_i = s_{best}$, $i++$
      end if
    end for
  end for
until not improved
return false

Question 11. En utilisant les opérateurs codés dans les deux sections précédentes, proposez et codez un algorithme de recherche locale. Ajoutez dans votre projet toutes les classes et les méthodes que vous jugez nécessaires.