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

TP 4 – Méthodes constructives

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. Pour le moment, nous avons mis en place le modèle de données ainsi que les éléments nécessaires pour la persistance des données.

L'idée est à présent de voir comment nous pouvons améliorer ce système d'informations en proposant un algorithme efficace pour résoudre le problème de planification des livraisons.

L'objectif de ce TP est donc de concevoir un algorithme pour planifier la livraison des clients de Bière2I. En particulier, nous nous intéressons au développement d'algorithmes qui sont appelés "méthodes constructives" : la solution est construite au fur et à mesure du déroulement de l'algorithme.

Ce TP permet d'illustrer les notions suivantes :

- solution réalisable ;
- méthodes heuristiques ;
- méthodes constructives ;
- méthodes d'insertions ;
- algorithme de *Clarke and Wright*.

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

Le problème que Monsieur Houblon, le responsable de la logistique, doit résoudre tous les jours s'appelle un *problème de tournées de véhicules avec capacité* (*Capacitated Vehicle Routing Problem* : CVRP).

Ce type de problème peut être formalisé de la manière suivante. On considère un dépôt unique, numéroté 0, et un ensemble de N points nommés *clients*. Les clients forment un ensemble $\mathcal{N} = \{1; 2; ...; N\}$. À chaque client $i \in \mathcal{N}$, il faut livrer une quantité $q_i > 0$ d'un produit (ici des caisses de bière). Cette quantité q_i est appelée la *demande* du client. Pour effectuer les livraisons, on dispose d'une *flotte* de K véhicules, notée $\mathcal{K} = \{1; 2; ...; K\}$. Les véhicules sont supposés homogènes : ils partent tous du dépôt, ils ont une capacité $Q > 0$, et ont des coûts identiques. Un véhicule qui livre un sous-ensemble de clients $\mathcal{S} \subseteq \mathcal{N}$ part du dépôt, se déplace une fois chez chaque client de $\mathcal{S}$, puis revient au dépôt. Lorsqu'un véhicule se déplace d'un point i vers un point j cela implique de payer un coût de déplacement c_{ij}. Les coûts c_{ij} sont définis pour tout $(i, j) \in \{0; 1; 2; ...; N\} \times \{0; 1; 2; ...; N\}$, et ne sont pas nécessairement symétriques (c_{ij} peut être différent de c_{ji}).

Une *tournée* est une séquence $r = (i_0; i_1; i_2; ...; i_s; i_{s+1})$, avec $i_0 = i_{s+1} = 0$ (i.e. la tournée part du dépôt et revient au dépôt). L'ensemble $\mathcal{S} = \{i_1; i_2; ...; i_s\} \subseteq \mathcal{N}$ est l'ensemble des clients visités dans la tournée. La tournée r a un coût $c(r) = \sum_{p=0}^{p=s} c_{i_p i_{p+1}}$. Une tournée est *réalisable* si la capacité du véhicule est respectée : $\sum_{p=1}^{p=s} q_p \leq Q$.

Une solution du CVRP consiste en un ensemble de K tournées réalisables, une pour chaque véhicule $k \in \mathcal{K}$. Une solution est dite *réalisable* si chaque tournée est réalisable et que chaque client est dans au moins une tournée. Le coût d'une solution est la somme des coûts des tournées qui forment la solution.

L'objectif est de trouver une solution réalisable avec un coût minimum. Les solutions réalisables qui ont un coût minimum sont dites *optimales*.

2 Méthode constructive

Le problème de CVRP est un problème "difficile" à résoudre. Plus précisément, il s'agit d'un problème NP-difficile : il n'existe pas d'algorithme polynomial pour résoudre le problème, à moins que P=NP.

Ainsi, de nombreuses méthodes *heuristiques* sont proposées pour résoudre ce problème. Une méthode heuristique est un algorithme qui fournit rapidement une solution réalisable à un problème, mais cette solution n'est pas nécessairement optimale.

Parmi ces méthodes heuristiques, on trouve des heuristiques *constructives*. Ce type d'heuristique opère de manière gloutonne, i.e. on prend des décisions les unes à la suite des autres sans jamais les remettre en cause. Cela permet de proposer une solution réalisable, en un temps de résolution très rapide.

3 Insertion simple

Dans un premier temps, nous proposons d'étudier une heuristique d'insertion très simple. L'idée est de considérer successivement chacun des clients. Pour chaque client $i \in \mathcal{N}$, on considère les véhicules déjà utilisés (auxquels on a affecté des clients), et si c'est possible on y ajoute le client i, puis on passe au client suivant. Si aucun véhicule déjà utilisé ne peut accueillir le client i, alors on utilise un nouveau véhicule de l'ensemble $\mathcal{K}$ pour livrer le client i.

3.1 Mise en place

Question 1. Dans un premier temps, reprenez le projet du TP précédent, et assurez-vous que les éléments suivants y apparaissent :

- la classe **Point** possède un attribut de type **Map<Point, Route>** pour stocker les routes partant du point ; la clé est le point de destination ;
- la classe **Instance** a pour attributs un id, un nom, une liste de points, une liste de véhicules et un planning (et bien évidemment les listes sont initialisées dans les constructeurs) ;
- les classes **Point**, **Vehicule** et **Planning** ont un attribut de type **Instance** (et donc avec les annotations qui vont bien pour faire le *mapping* objet-relationnel) ;
- le dao permet d'accéder aux objets de type **Instance**, et l'interface **InstanceDao** déclare une méthode abstraite **public Instance findByName(String name)** qui permet de récupérer une instance en donnant son nom ;
- la classe **Instance** possède une méthode **toString** qui affiche les points et les véhicules de l'instance.

Vous pouvez utilisez le test suivant pour vérifier que tout fonctionne correctement.

```
DaoFactory fabrique = DaoFactory.getDaoFactory(PersistenceType.JPA);
InstanceDao instanceManager = fabrique.getInstanceDao();
Instance inst = instanceManager.findByName("tiny_test");
System.out.println(inst);
```

Question 2. Avant de commencer l'algorithmique, il faut nous assurer que la classe **Instance** propose des méthodes qui nous serons utiles par la suite. Implémentez dans la classe **Instance** les méthodes suivantes :

- **public void clear()** qui permet de vider les éléments liés à la solution : les clients ne doivent plus être affectés à un véhicule, et avoir une positon par défaut, les véhicules doivent vider leur liste de clients et réinitialiser les attributs, et le planning doit vider son ensemble de véhicules et réinitialiser son coût ; bien évidemment, vous pouvez implémenter les méthodes que vous souhaitez dans d'autres classes que **Instance** ;
- **public List<Client> getClients()** qui renvoie une liste avec tous les clients de l'instance ;
- **public List<Vehicule> getVehicules()** qui renvoie une liste avec tous les véhicules de l'instance (faites une copie de la liste en attribut, ce sera beaucoup plus propre) ;
- **public void addVehiculeInPlanning (Vehicule v)** qui ajoute le véhicule **v** au planning ;
- **public void updatePositions()** qui met à jour les positions des clients dans les véhicules du planning ;
- **public double getCoutPlanning()** qui renvoie le coût du planning ;
- **public void printPlanning()** qui affiche le planning.

3.2 Algorithme

À présent, passons à la partie algorithmique. Un pseudo-code de l'algorithme d'insertion simple vous est proposé dans l'Algorithme 1.

Question 3. Ajoutez un paquetage **algo**, avec une classe **HeuristiqueConstructive** qui a en attribut un objet de type **Instance**, et un constructeur par données. Ajoutez une méthode **public void insertionSimple()** qui réalise l'algorithme d'insertion simple sur l'instance en attribut de la classe. N'hésitez pas à ajouter des méthodes avec une visibilité **private**, et ainsi à avoir plusieurs méthodes courtes : un dizaine de lignes au maximum ! Testez que tout fonctionne correctement avec l'exemple suivant.

```
DaoFactory fabrique = DaoFactory.getDaoFactory(PersistenceType.JPA);
InstanceDao instanceManager = fabrique.getInstanceDao();
Instance inst = instanceManager.findByName("tiny_test");
HeuristiqueConstructive heuristique = new HeuristiqueConstructive(inst);
heuristique.insertionSimple();
System.out.println("Cout : "+inst.getCoutPlanning());
inst.printPlanning();
instanceManager.update(inst);
```

Algorithme 1 : Algorithme d'insertion simple

```
1: //Nettoyer l'instance
2: 𝒩 = liste de clients;
3: 𝒦 = liste de véhicules;
4: ℒ = ∅; //Liste de véhicules utilisés
5: for all client c ∈ 𝒩 do
6:    affecte = false;
7:    for all véhicule v ∈ ℒ do
8:       if v.addClient(c) then
9:          affecte = true;
10:         break;
11:      end if
12:   end for
13:   if affecte = false then
14:      retirer un véhicule v de 𝒦;
15:      v.addClient(c); //Doit renvoyer true
16:      ajouter v dans ℒ; //Mettre à jour le planning
17:   end if
18: end for
19: //Mettre à jour les positions des clients dans le planning
```

3.3 Premiers tests

Lorsqu'on développe des algorithmes pour résoudre un problème, un bon usage est d'implémenter une méthode qui vérifie la validité de la solution : est-elle réalisable ? les valeurs sont-elles correctement calculées ?

Dans notre cas, une solution est réalisable si :

- dans chaque véhicule la capacité est respectée (la somme des demandes des clients livrés est inférieure ou égale à la capacité) ;
- chaque client est livré dans au moins un véhicule.

Il faut également vérifier que les valeurs sont bien calculées :

- la capacité utilisée dans chaque véhicule (la somme des demandes des clients livrés) ;
- le coût (la distance) pour chaque véhicule ;
- le coût total de la solution (somme des coûts des véhicules).

Question 4. Ajoutez dans la classe **Planning** une méthode **public boolean check()** qui vérifie que le planning est bien réalisable et que les valeurs sont correctement calculées. N'hésitez pas à rajouter les méthodes que vous jugez nécessaires. À la fin de la méthode **insertionSimple**, faites un appel à la méthode **check** du planning (on peut utiliser une méthode intermédiaire dans la classe **Instance**). Testez que le planning obtenu est bien correct.

Question 5. Nous pouvons à présent faire un test sur quelques instances. Récupérez sur Moodle les instances : *A-n32-k05*, *A-n48-k07*, *A-n63-k10*, *B-n41-k06*, *P-n016-k08*, *P-n050-k10*, *P-n076-k05* et *CMT01*. Exécutez les scripts pour rentrer chacune de ces instances sur la base de données. Vous pouvez ensuite regarder les résultats obtenus sur ces instances, avec le test suivant par exemple.

```
public static void testToutesInstances() {
    DaoFactory fabrique = DaoFactory.getDaoFactory(PersistenceType.JPA);
    InstanceDao instanceManager = fabrique.getInstanceDao();
    for(Instance inst : instanceManager.findAll()) {
        HeuristiqueConstructive heur = new HeuristiqueConstructive(inst);
        heur.insertionSimple();
        System.out.println("Instance : "+inst.getNom()
                           +"\tCout : "inst.getCoutPlanning()
                           +"\tNb vehicules : "+inst.getNbVehiculesPlannning());
        instanceManager.update(inst);
    }
}
```

4 Amélioration de l'insertion

Vous avez certainement constaté que la méthode d'insertion simple était un peu simpliste, et qu'il est donc facile de l'améliorer.

4.1 Client le plus proche

Dans un premier temps, une première piste d'amélioration est de faire attention à l'ordre dans lequel on considère l'insertion des clients, ainsi que l'ordre dans lequel on considère les véhicules.

Pour ce faire, on peut mettre en place les deux idées suivantes :

- après avoir inséré un client c, le prochain client à insérer est le client le plus proche (en distance) de c et qui n'est pas encore affecté à un véhicule ;
- quand on considère tous les véhicules déjà utilisés (la liste $\mathcal{L}$) pour y insérer un client c, il serait préférable de parcourir la liste dans le sens inverse (en partant de la fin) ; en effet, il est probable que le dernier véhicule soit moins rempli et qu'il contienne le client précédent (qui est proche du client c à insérer d'après ce qui a été mentionné au premier point).

Ainsi, l'algorithme du client le plus proche est une adaptation de l'algorithme d'insertion simple, explicité dans l'Algorithme 2.

Algorithme 2 : Algorithme d'insertion du client le plus proche

```
1: //Nettoyer l'instance
2: N = liste de clients;
3: K = liste de véhicules;
4: L = ∅; //Liste de véhicules utilisés
5: c = choisir un client dans N; //client à insérer
6: while N n'est pas vide do
7:    retirer c de N;
8:    affecte = false;
9:    for all véhicule v ∈ reverse(L) do
10:      if v.addClient(c) then
11:         affecte = true;
12:         break;
13:      end if
14:   end for
15:   if affecte = false then
16:      retirer un véhicule v de K;
17:      v.addClient(c); //Doit renvoyer true
18:      ajouter v dans L; //Mettre à jour le planning
19:   end if
20:   c = client de N le plus proche de c; //Mise à jour de c
21: end while
22: //Mettre à jour les positions des clients dans le planning
23: //Vérifier la validité du planning
```

Question 6. Dans la classe **HeuristiqueConstructive**, ajoutez une méthode **public void insertionClientProche()** qui réalise l'algorithme d'insertion du client le plus proche. Il faut essayer de réutiliser ce que vous avez développé précédemment, quitte à y apporter quelques modifications. Faites des méthodes courtes ! Testez que tout fonctionne correctement, et comparez vos résultats avec l'algorithme d'insertion simple. Vous pouvez également tester l'utilité de parcourir la liste $\mathcal{L}$ de véhicules utilisés dans l'ordre inverse.

4.2 Meilleure insertion

Il est encore possible de proposer une amélioration de la méthode d'insertion. Il y a deux éléments sur lesquels nous pouvons jouer :

- lorsqu'un client est inséré dans une tournée (un véhicule), il serait pertinent de ne pas systématiquement insérer le client à la fin (avant le retour au dépôt), et on pourrait évaluer quelle est la meilleure position (parmi toutes les positions possibles) pour insérer le client dans la tournée ;
- au lieu d'insérer un client dans le premier véhicule qui a la capacité suffisante, il serait préférable de vérifier parmi tous les véhicules qui ont la capacité suffisante, quel est le véhicule qui permettra de réaliser une insertion du client à moindre coût.

Et donc l'insertion d'un client c dans la liste de véhicules utilisés $\mathcal{L}$ doit être réalisée à l'endroit (choix du véhicule et position dans le véhicule) qui augmente le moins possible le coût de la solution.

Lorsque l'on parle d'insertion à moindre coût, il convient de préciser que l'insertion d'un client k entre deux points (client ou dépôt) i et j engendre un coût de

$$c_{ik} + c_{kj} - c_{ij}.$$

Il faut compter les coûts pour aller de i vers k, puis de k vers j ; et retrancher le coût pour aller de i vers j car en insérant le client k le trajet (i, j) ne sera plus effectué.

Le pseudo-code de l'algorithme est décrit dans les Algorithmes 3, 4 et 5.

Algorithme 3 : Algorithme d'insertion avec choix de la meilleure insertion

```
1: //Nettoyer l'instance
2: N = liste de clients;
3: K = liste de véhicules;
4: L = ∅; //Liste de véhicules utilisés
5: c = choisir un client dans N; //client à insérer
6: while N n'est pas vide do
7:    retirer c de N;
8:    affecte = meilleure insertion de c dans L; //Algorithme 4
9:    if affecte = false then
10:      retirer un véhicule v de K;
11:      v.addClient(c); //Doit renvoyer true
12:      ajouter v dans L; //Mettre à jour le planning
13:   end if
14:   c = client de N le plus proche de c; //Mise à jour de c
15: end while
16: //Mettre à jour les positions des clients dans le planning
17: //Vérifier la validité du planning
```

Question 7. Dans la classe **HeuristiqueConstructive**, ajoutez une méthode **public void meilleureInsertion()** qui réalise l'algorithme de meilleure insertion. Il faut essayer de réutiliser ce que vous avez développé précédemment, quitte à y apporter quelques modifications. Faites des méthodes courtes ! N'hésitez pas à définir une classe **MeilleureInsertionInfos** pour stocker les informations sur la meilleure insertion : coût, position, véhicule, client. Testez que tout fonctionne correctement, et comparez vos résultats avec ceux des algorithmes précédents.

Algorithme 4 : Algorithme de meilleure insertion

Require: c : client à insérer ; $\mathcal{L}$: véhicules déjà utilisés
affecte = false;
$\delta^{best} = +\infty$; //Meilleur coût
$v^{best} = \emptyset$; //Meilleur véhicule
$pos^{best} = \emptyset$; //Meilleure position
for all $v \in \mathcal{L}$ **do**
  δ = coût de meilleure insertion de c dans v; //$+\infty$ si insertion pas possible (à cause de la capacité)
  pos = position pour meilleure insertion de c dans v;
  //*delta* et *pos* sont calculés avec l'Algorithme 5
  if $\delta < \delta^{best}$ **then**
    $\delta^{best} = \delta$;
    $v^{best} = v$;
    $pos^{best} = pos$;
  end if
end for
if $\delta^{best} < +\infty$ **then**
  insérer c dans v^{best} à la position pos^{best};
  return true;
end if
return false;

Algorithme 5 : Calcul meilleure insertion dans une tournée

Require: c : client à insérer
if ajouter c n'est pas possible à cause de la capacité **then**
  return $(+\infty; \emptyset)$;
end if
δ^{best} = coût pour insérer c en position 0; //Meilleur coût
$pos^{best} = 0$; //Meilleure position
for all pos de 1 jusqu'au nombre de clients déjà présents **do**
  δ = coût pour insérer c en position pos;
  if $\delta < \delta^{best}$ **then**
    $\delta^{best} = \delta$;
    $pos^{best} = pos$;
  end if
end for
return $(\delta^{best}; pos^{best})$;

5 Algorithme de *Clarke and Wright*

Nous nous intéressons à une méthode différente pour la construction d'une solution réalisable. Il s'agit de l'algorithme de *Clarke and Wright* [1] qui date de 1964. L'idée est la suivante. Initialement, on construit une tournée $(0, i, 0)$ pour chaque client $i \in \mathcal{N}$; c'est-à-dire que chaque tournée consiste en un aller-retour vers un client. Cette solution initiale est donc une solution réalisable, mais très certainement très coûteuse ... De manière itérative on essaye d'améliorer cette solution en fusionnant des tournées ensemble, tant que cela est possible et permet de diminuer le coût de la solution. Plus précisément, la fusion d'une tournée $r = (0, ..., i, 0)$ (tournée qui termine par le client i) avec une tournée $s = (0, j, ..., 0)$ (tournée qui commence par le client j) va donner la tournée $t = (0, ..., i, j, ..., 0)$, i.e. que l'on va directement de i vers j sans repasser par le dépôt. Bien évidemment, la fusion ne peut se faire que si la capacité du véhicule est respectée dans la nouvelle tournée t. La fusion des deux tournées $r = (0, ..., i, 0)$ et $s = (0, j, ..., 0)$ entraine un gain de :

$$g_{rs} = c_{ij} - c_{i0} - c_{0j}.$$

Ainsi, à chaque itération, on calcule tous les gains possibles en fusionnant toutes les paires de tournées de la liste $\mathcal{L}$, puis on implémente réellement la fusion des deux tournées qui permettent le meilleur gain. La liste $\mathcal{L}$ contient donc une tournée en moins à la fin de chaque itération. Les Algorithmes 6 and 7 décrivent plus précisément la méthode.

Algorithme 6 : Algorithme de *Clarke and Wright*

//Nettoyer l'instance
$\mathcal{N}$ = liste de clients;
$\mathcal{K}$ = liste de véhicules;
$\mathcal{L} = \emptyset$; //Liste de véhicules utilisés
for all client $c \in \mathcal{N}$ **do**
  retirer un véhicule v de $\mathcal{K}$;
  $v.addClient(c)$; //Doit renvoyer true
  ajouter v dans $\mathcal{L}$; //Mettre à jour le planning
end for
//On a alors une solution réalisable initiale
ameliore = true;
while ameliore **do**
  ameliore = fusionTournees($\mathcal{L}$); //Algorithme 7
end while
//Mettre à jour les positions des clients dans le planning
//Vérifier la validité du planning

Algorithme 7 : Algorithme de fusion des tournées

Require: $\mathcal{L}$ liste de véhicules
$r^{best} = \emptyset$;
$s^{best} = \emptyset$;
$g^{best} = +\infty$; //Meilleur gain
for all véhicule $r \in \mathcal{L}$ **do**
  for all véhicule $s \in \mathcal{L}$ **do**
    if r et s peuvent être fusionnés (capacité respectée) **then**
      $g = c_{ij} - c_{i0} - c_{0j}$; //$r$ termine par le client i, et s commence par le client j
      if $g < g^{best}$ **then**
        $g^{best} = g$;
        $r^{best} = r$;
        $s^{best} = s$;
      end if
    end if
  end for
end for
if $g^{best} \geq 0$ **then**
  return false;
end if
fusionner r et s dans r; //Mettre à jour tous les attributs
supprimer s de $\mathcal{L}$;
return true;

Question 8. Dans la classe **HeuristiqueConstructive**, ajoutez une méthode **public void clarkeAndWright()** qui réalise l'algorithme de *Clarke and Wright*. Il faut essayer de réutiliser ce que vous avez développé précédemment, quitte à y apporter quelques modifications. Faites des méthodes courtes ! Il faut implémenter des méthodes dans la classe **Vehicule**. Testez que tout fonctionne correctement, et comparez vos résultats avec ceux des algorithmes précédents.

References

[1] Clarke, G. and Wright, J. W., *Scheduling of vehicles from a central depot to a number of delivery points. Operations research, 12(4), 568-581. 1964.*

[2] Toth, P. and Vigo, D., *Vehicle routing: Problems, methods, and applications (2nd ed.), MOS-SIAM series on optimization, 2014.*