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

TP 2 – Méthodes constructives – Partie 1

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*). Pour le moment, nous avons mis en place le modèle objet, ce qui nous permet de lire des instances et d'avoir le modèle pour stocker des solutions.

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

Dans un premier temps, nous ajouterons quelques méthodes à notre classe **Solution** afin de pouvoir construire une solution, et nous mettrons en place un checker afin de pouvoir valider que nos solutions sont correctes.

L'objectif de ce TP est de concevoir un algorithme très simple qui nous permet d'obtenir des solutions initiales. 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 :

- le développement d'un checker ;
- solution réalisable ;
- méthodes heuristiques ;
- méthodes constructives ;
- méthode d'insertion simple ;
- méthode d'insertion du plus proche voisin.

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 Rappel des notations et du problème de CVRP

Le CVRP (*Capacitated Vehicle Routing Problem*) peut être formalisé de la manière suivante [1]. On considère un dépôt unique, numéroté 1, et un ensemble de N points nommés *clients*. Les clients forment un ensemble $\mathcal{N} = \{2; 3; ...; N+1\}$. 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 véhicules. 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 $(i, j) \in \{1; 2; 3; ...; N+1\} \times \{1; 2.3; ...; N+1\}$, 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 $i_0 = i_{s+1} = 1$ (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 tournées réalisables. Une solution est dite *réalisable* si chaque tournée est réalisable et que chaque client est dans exactement 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 Ajouter des clients dans la solution

Lors du TP précédent, nous avons mis en place le modèle objet de la classe **Solution**. Nous allons ici ajouter quelques méthodes simples dans la classe **Solution** afin de pouvoir construire des solution en y ajoutant des clients.

Question 1. Ajoutez dans la classe **Solution** les deux méthodes suivantes.

- Une méthode qui ajoute un client à la solution dans une nouvelle tournée.
- Une méthode qui ajoute un client à la solution dans une tournée existante de la solution (on peut parcourir les tournées et faire l'ajout dans la première tournée où c'est possible). Cette méthode renvoie un booléen qui indique si le client a pu être ajouté dans une tournée ou non.

Testez que votre code fonctionne correctement.

Dans ces deux méthodes, n'oubliez pas de mettre à jour le coût total de la solution en temps constant $O(1)$.
Pour le test, considérez par exemple l'instance 'A-n32-k5.vrp', puis construisez une solution complète de la manière suivante :

- parcourez tous les clients de l'instance ;
- pour chaque client essayez de l'ajouter dans une tournée existante ;
- si l'ajout dans une tournée existante n'est pas possible, alors faites l'ajout dans une nouvelle tournée.

3 Checker pour la solution

Avant de pouvoir commencer à implémenter des algorithmes pour résoudre le problème de CVRP, il nous reste une dernière étape à effectuer : mettre en place une méthode pour vérifier qu'une solution est valide. Nous appellerons cela un checker. Ceci va nous être grandement utile lors du développement des algorithmes car ça permettra de détecter très facilement un bug dans le code : à chaque fois que nous obtiendrons une nouvelle solution, nous ferrons appel au checker pour valider la solution.

Notez bien qu'en principe, une fois que le développement des algorithmes est terminé et que l'on souhaite évaluer les performances en temps de calcul, on arrête alors d'appeler le checker car cela consomme du temps.
Par ailleurs, nous parlons ici de checker que le développeur développe directement dans son code. Notez cependant qu'il est très fréquent dans un projet d'avoir également un checker externe développé par une autre personne que le développeur et qui permet de valider la solution finale. Un tel checker externe sera mis à votre disposition lors du projet. Mais cela est un complément au checker interne, ça ne le remplace pas.

Une tournée est réalisable si :

- la demande totale et le coût total sont correctement calculés ;
- la demande totale est inférieure ou égale à la capacité.

Une solution est réalisable si :

- ses tournées sont toutes réalisables ;
- le coût total de la solution est correctement calculé ;
- tous les clients sont présents dans exactement une seule des tournées de la solution.

Question 2. Dans la classe **Solution**, ajouter une méthode **check()** qui renvoie un booléen qui indique si la solution est réalisable ou non. Testez que votre checker fonctionne correctement.

Si une solution est non réalisable, il est fortement recommandé d'afficher sur la console un message qui permet d'indiquer précisément où est l'erreur.
Écrivez des méthodes courtes. Vous êtes donc très fortement encouragés à écrire une méthode (avec une visibilité **private**) pour chaque élément à tester. De plus, en toute logique, tester la validité d'une tournée devrait être réalisé dans la classe **Tournee**.
Par ailleurs, n'oubliez pas de respecter le principe d'encapsulation, en particulier si dans la classe **Solution** vous avez besoin de connaître les clients dans une tournée.
Pour tester, vérifier que la solution construite au test de la question précédente est correcte. Vous pouvez également vérifiez que si vous ne mettez pas tous les clients dans une solution, alors cette dernière n'est pas correcte.

4 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.

Toutes les méthodes constructives que nous allons développer sont en fait des solveurs qui permettent, à partir d'une instance, de proposer une solution réalisable pour cette instance. Ainsi, dans un premier temps, nous allons ajouter dans notre code une interface **Solveur** qui devra être implémentée par toutes les méthodes de résolution que nous développerons par la suite.

Question 3. Ajouter un paquetage **solveur**. Dans ce paquetage, ajoutez une interface **Solveur** avec les méthodes suivantes :

- **getNom()** qui renverra le nom de la méthode de résolution sous forme d'une chaîne de caractères ;
- **solve(Instance instance)** qui prend en paramètre une instance du CVRP, et renvoie une solution réalisable pour cette instance.

5 Insertion simple

5.1 Mise en place de l'algorithme

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 tournées actuelles de la solution (donc des tournées non vides), et si c'est possible on y ajoute le client i, puis on passe au client suivant. Si aucune tournée de la solution ne peut accueillir le client i, alors on insère ce client i dans une nouvelle tournée.

Un pseudo-code de l'algorithme d'insertion simple vous est proposé dans l'Algorithme 1.

Question 4. Dans le paquetage **solveur**, ajoutez une classe **InsertionSimple** qui implémente l'interface **Solveur**. Implémentez les méthodes de l'interface **Solveur**. En particulier, la méthode **solve** devra avoir le comportement de l'algorithme d'insertion simple (Algorithme 1).

Testez que votre code fonctionne correctement.

Algorithme 1 : Algorithme d'insertion simple.

```
1: 𝒩 = liste de clients
2: 𝒮 = solution vide (sans tournées)
3: for all client i ∈ 𝒩 do
4:    affecte = false
5:    if ajout du client i dans une tournée de 𝒮 then
6:       affecte = true
7:    end if
8:    if affecte = false then
9:       ajout du client i dans une nouvelle tournée de 𝒮
10:   end if
11: end for
12: return 𝒮
```

La méthode **solve** doit être très simple à implémenter si vous avez bien implémenté dans la classe **Solution** la méthode qui ajoute un client à la solution dans une nouvelle tournée, et la méthode qui ajoute un client à la solution dans une tournée existante de la solution. C'était l'objet de la première question de ce sujet.
Pour le test, vous pouvez tester sur l'instance 'A.n32.k5.vrp' que vous obtenez bien une solution réalisable.

5.2 Test

Maintenant que vous avez vérifié que l'algorithme d'insertion simple semblait fonctionner correctement, nous allons faire un test plus complet qui nous permettra de valider cet algorithme sur un ensemble d'instances.

Lors du TP précédent, vous avez pu télécharger un répertoire *instances* contenant 10 instances du CVRP. C'est sur ces 10 instances que nous allons faire notre test.

Vous trouverez également sur Moodle un répertoire *test* qui contient une classe Java **TestAllSolveur** qui permet de tester et comparer les performances de plusieurs méthodes de résolution sur un jeu d'instances. Vous utiliserez donc cette classe pour tester et comparer la performance de vos algorithmes. Il suffit juste de vérifier que les signatures de certaines méthodes et des constructeurs sont concordants avec votre propre code. Les résultats du test seront enregistrés dans un fichier '.csv' et vous pourrez voir, pour chaque solveur et chaque instance, le coût de la solution, le temps de résolution en millisecondes, et si la solution est valide (d'après le checker que vous avez développé au début de ce TP).

Lorsque l'on développe un algorithme pour résoudre un problème d'optimisation, il est important de tester les performances de l'algorithme sur un jeu d'instances contenant plusieurs instances avec des caractéristiques différentes. Tester sur une seule instance peut produire des biais : on aurait alors tendance à concevoir un algorithme très performant sur cette seule instance, et probablement moins performant sur de nombreuses autres instances.
Par ailleurs, il sera également important, lors du développement futur des algorithmes, de vérifier que les performances des algorithmes précédents ne sont pas affectées.

Question 5. Mettez le répertoire *test* dans le répertoire *src* de votre projet (**test** est le nom du paquetage). Vérifiez que la classe **TestAllSolveur** du paquetage **test** est correcte. Pour cela, des commentaires commençant par 'TO CHECK' vous indiquent les lignes à regarder avec attention. Il n'est pas utile de s'attarder sur le reste du code. Dans la méthode principale **main**, se trouvent le chemin du répertoire de test ainsi que le nom du fichier de résultats.

Testez cette classe afin de vérifier que vous obtenez bien des solutions valides sur toutes les instances avec votre algorithme d'insertion simple. Vérifiez également que le temps de résolution est de l'ordre de quelques millisecondes.

6 Amélioration de l'insertion : plus proche voisin

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

Dans un premier temps, une piste d'amélioration est de faire attention à l'ordre dans lequel on considère les clients à insérer. En effet, jusqu'à présent, nous avons inséré les clients dans l'ordre dans lequel ils étaient présents dans l'instance. Il y a donc de fortes chances que deux clients éloignés l'un de l'autre soient insérés de manière consécutive dans la même tournée. La solution obtenue n'est donc pas de très bonne qualité.

Afin de pallier ce problème, l'algorithme du plus proche voisin se base sur l'idée suivante : après avoir inséré un client i dans une tournée de la solution, on choisit d'insérer ensuite le client j tel que j est le plus proche du client i (le coût de i à j est le plus faible).

Ainsi, l'algorithme du plus proche voisin est une adaptation de l'algorithme d'insertion simple, explicité dans l'Algorithme 2. Notez bien que, pour que l'algorithme fonctionne correctement, il faut pouvoir ajouter un client dans la dernière tournée de la solution (ligne 6 de l'Algorithme 2). En effet, ceci permet de garantir que les clients les plus proches sont bien dans la même tournée.

Notez que nous avons besoin ici d'avoir accès à la dernière tournée ajoutée à la solution. C'est une des raisons pour laquelle l'utilisation d'une structure de données de type **List** a été proposée pour stocker les tournées dans la classe **Solution**.

Algorithme 2 : Algorithme d'insertion du plus proche voisin.

1: $\mathcal{N}$ = liste de clients
2: $\mathcal{S}$ = solution vide (sans tournées)
3: i = premier client de $\mathcal{N}$ // Client à insérer
4: **while** $\mathcal{N}$ n'est pas vide **do**
5: affecte = false
6: **if** ajout du client i dans la dernière tournée de $\mathcal{S}$ **then**
7: affecte = true
8: **end if**
9: **if** affecte = false **then**
10: ajout du client i dans une nouvelle tournée de $\mathcal{S}$
11: **end if**
12: retirer i de $\mathcal{N}$
13: j = client de $\mathcal{N}$ le plus proche de i
14: $i = j$ // Mise à jour du client à insérer
15: **end while**
16: **return** $\mathcal{S}$

Question 6. Ajoutez dans la classe **Solution** une méthode qui ajoute un client à la solution dans la dernière tournée existante de la solution (voir la ligne 6 de l'Algorithme 2). Cette méthode renvoie un booléen qui indique si le client a pu être ajouté dans la dernière tournée ou non.

Testez que votre code fonctionne correctement.

N'oubliez pas de mettre à jour le coût total de la solution en temps constant $O(1)$.

Évitez de dupliquer du code. La méthode d'ajout d'un client dans la dernière tournée a certainement des points similaires avec la méthode d'ajout d'un client dans une tournée existante (vous avez développé cette méthode dans le TP précédent). Par exemple, vous pouvez développer dans la classe **Solution** une méthode (avec une visibilité **private**) **ajouterClient(Client client, Tournee t)** qui renvoie un booléen pour indiquer si l'ajout a eu lieu ou non. Cette méthode sera alors ensuite appelée dans les deux méthodes d'ajout d'un client dans une tournée existante et dans la dernière tournée.

Pour le test, considérez par exemple l'instance 'A-n32-k5.vrp', puis construisez une solution complète de la manière suivante :

- parcourez tous les clients de l'instance ;
- pour chaque client, essayez de l'ajouter dans la dernière tournée de la solution ;
- si l'ajout dans la dernière tournée n'est pas possible, alors faites l'ajout dans une nouvelle tournée.

Vérifiez ensuite que la solution obtenue est validée par le checker.

Question 7. Dans le paquetage **solveur**, ajoutez une classe **InsertionPlusProcheVoisin** qui implémente l'interface **Solveur**. Implémentez les méthodes de l'interface **Solveur**. En particulier, la méthode **solve** devra avoir le comportement de l'algorithme du plus proche voisin (Algorithme 2).

Testez que votre code fonctionne correctement.

Rappelez-vous de garder un code propre. En particulier, n'hésitez pas à faire une méthode avec une visibilité **private** pour rechercher le client le plus proche (ligne 13 de l'Algorithme 2). Cette méthode prend en paramètres le client i qui vient d'être inséré et la liste $\mathcal{N}$ des clients restant à insérer, et elle retourne le client j dans la liste $\mathcal{N}$ qui est le plus proche du client i.

Par ailleurs, notez que si vous avez implémenté correctement la méthode **getClients** de la classe **Instance**, alors supprimer un client de la liste des clients (ligne 12 de l'Algorithme 2) ne pose pas de problème en terme d'encapsulation des données de l'instance.

Pour le test, vous pouvez vérifier que vous obtenez bien une solution réalisable sur l'instance 'A-n32-k5.vrp', et que cette solution a un coût plus faible que celle obtenue avec l'algorithme d'insertion simple.

Question 8. Dans la classe **TestAllSolveur**, dans la méthode **addSolveurs()**, ajoutez dans la liste des solveurs une instance du solveur **InsertionPlusProcheVoisin**. Un commentaire commençant par 'TO ADD' vous indique à quel endroit précis faire cet ajout.

Testez donc que vous obtenez bien une solution valide sur toutes les instances, et comparez les coûts de la solution à ceux obtenus avec la méthode d'insertion simple.

References

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