Programmation Orientée Objet
IG2I, LE4-SI – 2020-2021

TP 2 – Méthodes constructives

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.

L'objectif de ce TP est de concevoir un algorithme qui nous permet d'obtenir de bonnes 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. Nous verrons dans les TPs suivants comment améliorer la solution initiale construite lors de ce TP.

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

- 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 [2]. 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 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 1. 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.

3 Insertion simple

3.1 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 2. 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 lors du dernier TP 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.
Pour le test, vous pouvez tester sur l'instance 'A.n32.k5.vrp' que vous obtenez bien une solution réalisable.

3.2 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 (selon le checker que vous avez développé lors du TP précédent).

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 3. 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.

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 Plus proche voisin

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.

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

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

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**.

Question 4. 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 une tournée existante ;
- si l'ajout dans une tournée existante n'est pas possible, alors faites l'ajout dans une nouvelle tournée.

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

Question 5. 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 6. 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.

4.2 Meilleure insertion

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

- lorsqu'un client est inséré dans une tournée, 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 la première tournée qui a la capacité suffisante, il serait préférable de vérifier parmi toutes les tournées qui ont la capacité suffisante, quelle est celle qui permettra de réaliser une insertion du client à moindre coût.

Et donc l'insertion d'un client k dans la solution courante doit être réalisée à l'endroit (choix de la tournée et position dans la tournée) qui augmente le moins le coût de la solution.

Dans la suite, nous allons voir en détail comment implémenter l'algorithme de meilleure insertion.

4.2.1 Évaluation de l'insertion d'un client à une position

Dans un premier temps, nous nous intéressons à évaluer le coût d'insertion d'un client k à la position pos d'une tournée. Ceci nous permettra par la suite de comparer toutes les insertions possibles et d'implémenter celle qui a le plus faible coût d'insertion.

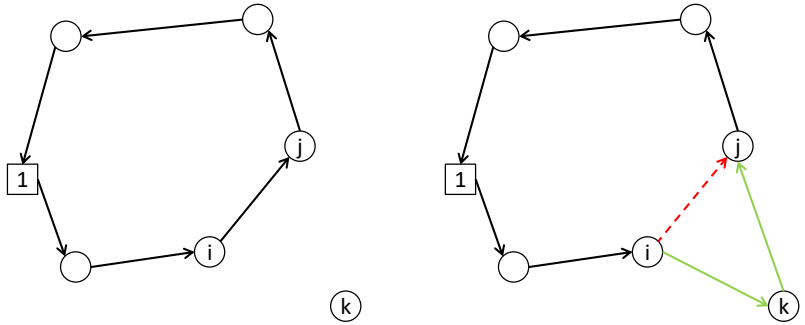
Lorsque l'on parle d'insertion avec le plus faible coût, il convient de préciser que l'insertion d'un client k à la position pos revient à insérer le client k entre deux points (client ou dépôt) i et j. On supposera que dans une tournée qui contient n clients les positions des clients sont indicées de 0 à $n-1$, le dépôt est présent en position -1 (début de la tournée) et en position n (fin de la tournée). L'insertion d'un nouveau client k peut se faire aux positions 0 à n. L'insertion d'un client k en position $pos \in \{0; 1; \cdots, n\}$ se fait donc entre le point i actuellement en position $pos-1$ et le point j actuellement en position pos.

Comme nous l'avons fait pour l'insertion d'un client en fin de tournée, nous souhaitons évaluer l'insertion d'un client k à la position pos en temps constant $O(1)$. Pour rappel, le dépôt est noté 1, et on note c_{ij} le coût entre deux points i et j. Deux cas sont à considérer.

- Si on veut insérer k dans une tournée qui n'a pas de clients, alors cela engendre un coût $c_{1k} + c_{k1}$ (voir Figures 3a et 3b du TP1).
- Si on veut insérer k dans une tournée qui contient déjà des clients, entre le point i à la position $pos - 1$ et le point j à la position pos, cela 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. Ceci est illustré dans la Figure 1.



(a) On souhaite ajouter le client k entre les points i et j d'une tournée

(b) La tournée visite le client k entre les points i et j

Figure 1 – Modifications du coût total d'une tournée lors de l'ajout d'un client k entre deux points i et j (ici k est inséré en position 2 de la tournée). Les coûts à rajouter sont ceux des routes en vert, et les coûts à retirer sont ceux des routes en pointillés rouges.

Question 7. Dans la classe **Tournee**, ajoutez les trois méthodes suivantes avec une visibilité **private** :

- **getPrec(int position)** qui renvoie le point de la tournée qui précède la position **position** : si **position** vaut 0, alors il s'agit du dépôt ;
- **getCurrent(int position)** qui renvoie le point de la tournée qui est actuellement à la position **position** : si **position** est égal au nombre de clients dans la tournée, alors il s'agit du dépôt ;
- **isPositionInsertionValide(int position)** qui renvoie un booléen indiquant s'il est possible ou non d'insérer un client en position **position**.

Question 8. Dan la classe **Tournee**, ajoutez une méthode **deltaCoutInsertion(int position, Client clientToAdd)** qui renvoie le coût engendré par l'insertion du client **clientToAdd** à la position **position** de la tournée. Cette méthode renverra l'infini si la position passée en paramètre est incorrecte.

Évitez de dupliquer du code. Lors du TP précédent, vous avez certainement calculé le coût engendré par l'ajout d'un client à la fin de la tournée. À présent, ce coût peut être calculé en faisant appel à la méthode **deltaCoutInsertion(int position, Client clientToAdd)** en passant les bons paramètres. Après avoir modifié votre code, vérifiez que vous n'avez pas fait d'erreur en testant à nouveau les algorithmes d'insertion simple et de plus proche voisin : les résultats ne doivent pas être modifiés.

Question 9. Dans le paquetage **test**, ajoutez une classe **TestMeilleureInsertion** dans laquelle vous ferez un test pour vérifier que le calcul du coût d'insertion d'un client est correct.

Vous pouvez créer une petite instance avec 4 clients qui sont alignés (par exemple toutes les ordonnées sont 0). Ainsi, il est facile pour vous de calculer les distances. Vous insérez ensuite 3 clients dans une tournée. Vous pouvez vous aider du code ci-dessous par exemple.

```
1  int id = 1;
2  Depot d = new Depot(id++, 0, 0);
3  Instance inst = new Instance("test", 100, d);
4  Client c1 = new Client(10, id++, 0, 5);
5  Client c2 = new Client(10, id++, 0, 10);
6  Client c3 = new Client(10, id++, 0, 20);
7  Client cIns = new Client(10, id++, 0, 15);
8  inst.ajouterClient(c1);
9  inst.ajouterClient(c2);
10 inst.ajouterClient(c3);
11 inst.ajouterClient(cIns);
12 Tournee t = new Tournee(inst);
13 t.ajouterClient(c1);
14 t.ajouterClient(c2);
15 t.ajouterClient(c3);
```

Ensuite, vous affichez le coût d'insertion du client **cIns** à différentes positions (entre -1 et 4) et vérifiez que les coûts sont correctement calculés.

4.2.2 Stockage des informations sur l'opérateur d'insertion

À présent, nous sommes capables d'évaluer le coût engendré par l'insertion d'un client k à la position *pos* dans une tournée. Afin de pouvoir sélectionner l'insertion qui produira le plus faible coût d'insertion, il va nous falloir stocker dans des objets les informations liées à chaque insertion. Ainsi, au fur et à mesure de la recherche de la meilleure insertion, on ne mémorisera que l'objet qui permet l'insertion à moindre coût.

Nous allons appeler *opérateur* une fonction qui modifie une solution selon un comportement bien défini. Cette notion est très générale et nous verrons par la suite que nous l'utilisons très souvent. Pour le moment, nous allons nous intéresser à une catégorie d'opérateurs qui sont les opérateurs d'insertion d'un client dans une tournée.

Ainsi, nous allons définir le comportement générique d'un opérateur, ce qui nous servira par la suite, et le comportement spécifique de l'opérateur d'insertion. D'un point de vue de l'implémentation, ceci se fera avec une classe abstraite **Operateur** et une classe **InsertionClient** qui hérite de **Operateur**.

Notez bien ici que cette notion d'opérateur est générique, et peut être appliquée à de nombreux problèmes d'optimisation. Nous allons montrer ici comment développer des opérateurs spécifiques au problème de CVRP. Face à un autre problème, la démarche resterait la même.

Dans le cadre du problème de CVRP, un opérateur est toujours défini par une tournée (celle sur laquelle on applique une modification), ainsi qu'un coût (celui engendré par l'application de l'opérateur). Par défaut, le coût d'un opérateur est infini. La manière dont ce coût est calculé dépend de chaque opérateur, et sera donc implémentée dans les classes filles de la classe **Operateur**. Il est cependant possible de comparer deux opérateurs : le meilleur est celui qui a le coût le plus faible (dans le cas d'un problème de minimisation des coûts). Par ailleurs, on dit qu'un opérateur conduit à un mouvement réalisable si le coût de l'opérateur n'est pas infini.

Question 10. Ajoutez un paquetage **operateur**. Dans ce paquetage, ajoutez une classe abstraite **Operateur**. Cette classe sera complétée au fur et à mesure du TP. Pour le moment, définissez les attributs de cette classe, un constructeur par défaut, et un constructeur par la donnée de la tournée (le coût n'est pas passé en paramètre, mais calculé par chaque opérateur spécifique). Ajoutez les méthodes suivantes :

- **getDeltaCout()** qui renvoie le coût de l'opérateur ;
- **isMouvementRealisable()** qui renvoie un booléen indiquant si le mouvement associé à l'opérateur est réalisable ou non ;
- **isMeilleur(Operateur op)** qui indique si l'opérateur **this** est meilleur que l'opérateur **op** ou non (renvoie **true** si **this** est meilleur que **op**).

Définissez également une méthode abstraite **evalDeltaCout** qui renvoie un entier correspondant au coût engendré par l'application de l'opérateur. Enfin, redéfinissez la méthode **toString**.

Notez que le langage Java permet de définir une visibilité intermédiaire entre **public** et **private**. Il s'agit de la visibilité **protected**. Si un attribut ou une méthode est déclarée avec la visibilité **protected**, alors il est accessible dans la classe qui définit cet attribut ou cette méthode, mais aussi dans toutes les classes filles et les classes du même paquetage.
Ainsi, les attributs et la méthode abstraite **evalDeltaCout** ne doivent surtout pas avoir une visibilité **public** afin de bien respecter l'encapsulation, mais peuvent avoir une visibilité **protected** afin de pouvoir être accessibles dans les classes filles de **Operateur**.

L'opérateur d'insertion d'un client est un opérateur pour lequel on précise le client qui va être inséré ainsi que la position d'insertion dans la tournée. L'évaluation du coût d'insertion est réalisée grâce à la méthode d'évaluation du coût d'insertion que nous avons développée préalablement dans la classe **Tournee**.

Question 11. Dans le paquetage **operateur**, ajoutez une classe **InsertionClient** qui hérite de **Operateur**. Définissez ses attributs et implémentez la méthode **evalDeltaCout**.

Ajoutez un constructeur par défaut, un constructeur par la donnée de la tournée, du client et de la position (le coût est alors évalué avec la méthode **evalDeltaCout**). Ajoutez les accesseurs nécessaires ainsi qu'une redéfinition de la méthode **toString**.

Question 12. Modifiez votre test de la classe **TestMeilleureInsertion** pour vérifier que :

- vous arrivez bien à créer des objets de type **InsertionClient** ;
- les valeurs des attributs de ces objets sont correctes (en particulier le coût) ;
- les méthodes **isMeilleur** et **isMouvementRealisable** renvoient des résultats corrects.

4.2.3 Recherche de la meilleure insertion pour un client

Maintenant que nous sommes capables de comparer entre eux des opérateurs d'insertion de clients, il nous faut des algorithmes qui nous permettent de déterminer, pour un client donné, le meilleur opérateur d'insertion dans une solution.

Afin de faciliter le développement algorithmique et de bien répartir les traitements dans les différentes classes, nous allons procéder en deux étapes :

- dans un premier temps, nous chercherons le meilleur opérateur d'insertion dans une tournée ;

- dans un second temps, nous chercherons le meilleur opérateur d'insertion parmi les tournées d'une solution.

Commençons par nous intéresser à la recherche du meilleur opérateur d'insertion d'un client k dans une tournée. Pour cela, vous devez d'abord vérifier si le client peut être inséré dans la tournée (est-ce que la capacité sera toujours respectée ?). Si le client k peut être inséré, alors il faut tester toutes les positions d'insertion possibles et renvoyer le meilleur opérateur d'insertion du client k dans la tournée (celui qui engendre le plus faible coût).

Question 13. Dans la classe **Tournee**, ajoutez une méthode **getMeilleureInsertion(Client clientToInsert)** qui renvoie le meilleur opérateur d'insertion du client **clientToInsert**.

Notez que savoir si un client peut être ajouté à la tournée est quelque chose que vous avez déjà dû faire dans la méthode **ajouterClient(Client clientToAdd)**. Donc ne faites pas de code redondant, et introduisez si besoin une nouvelle méthode avec une visibilité **private** afin de réaliser ce traitement.

Question 14. Modifiez votre test de la classe **TestMeilleureInsertion** pour vérifier que vous arrivez bien à trouver le meilleur opérateur d'insertion dans la tournée.

Maintenant, nous nous intéressons à la recherche du meilleur opérateur d'insertion d'un client k dans une solution. Il faut donc déterminer le meilleur opérateur parmi les tournées de la solution. Pour ce faire, il suffit de parcourir toutes les tournées de la solution, et de renvoyer le meilleur opérateur d'insertion du client k dans ces tournées.

Question 15. Dans la classe **Solution**, ajoutez une méthode **getMeilleureInsertion(Client clientToInsert)** qui renvoie le meilleur opérateur d'insertion du client **clientToInsert**.

4.2.4 Implémentation du mouvement de meilleure insertion

Nous sommes donc capables à présent de déterminer le meilleur opérateur d'insertion pour un client. Avant d'implémenter l'algorithme de meilleure insertion, il nous faut également être capables d'implémenter un mouvement d'insertion d'un client.

Ici, il est important de comprendre que le terme "implémenter un mouvement" signifie "modifier la solution en appliquant le mouvement associé à l'opérateur". Dans notre cas, implémenter le mouvement d'insertion d'un client signifie donc "ajouter le client à la solution dans la tournée et à la position définies par l'opérateur". Cette implémentation engendrera dans la tournée (et donc dans la solution) un coût égal à celui calculé dans l'opérateur d'insertion.

Cette fois, nous allons procéder en trois temps :

- on implémente le mouvement d'insertion dans une tournée ;
- on implémente le mouvement lié à l'opérateur dans cet opérateur si le mouvement est réalisable (voir méthode **isMouvementRealisable** de la classe **Operateur**) ;
- on implémente le mouvement lié à un opérateur d'insertion dans la classe **Solution**.

Notez bien ici que nous proposons de procéder de la manière suivante pour implémenter le mouvement d'un opérateur d'insertion **infos** :

- on fait appel à une méthode de la classe **Solution** pour implémenter le mouvement de **infos** ;
- cette méthode de la classe solution fait appel à une méthode de la classe **Operateur** pour implémenter le mouvement s'il est réalisable, et met à jour en conséquence le coût de la solution ;
- la méthode de la classe **Operateur** (en fait de la classe **InsertionClient** qui hérite de **Operateur**) fait appel à une méthode de la classe **Tournee** pour effectivement modifier la tournée selon les information contenues dans **infos**.

Question 16. Dans la classe **Tournee**, ajoutez une méthode **doInsertion(InsertionClient infos)** qui implémente le mouvement lié à l'opérateur d'insertion **infos**. Cette méthode renvoie un booléen qui indique si le mouvement d'insertion a bien été implémenté.

Il faut donc insérer le client à la bonne position, et mettre correctement à jour les attributs de la tournée (coût total et demande totale). Pour cela on se sert des informations contenues dans le paramètre **infos**.
Vous pouvez faire appel à la méthode **check** de la classe **Tournee** à la fin de la méthode **doInsertion** afin de vérifier que les attributs ont bien été mis à jour.

Question 17. Dans la classe **Operateur**, ajoutez une méthode abstraite **doMouvement()** qui renvoie un booléen indiquant si le mouvement lié à l'opérateur a bien été réalisé.

Ajoutez également une méthode **doMouvementIfRealisable()** avec une visibilité **public** qui implémente le mouvement lié à l'opérateur (méthode **doMouvement**) si celui-ci est réalisable. Cette méthode renvoie un booléen qui indique si le mouvement a été implémenté ou non.

La méthode **doMouvement** pourra avoir une visibilité **protected** car elle n'est pas sensée être accessible des autres classes mais doit être définie dans les classes filles.

Question 18. Dans la classe **InsertionClient**, implémentez la méthode **doMouvement** qui réalise le mouvement lié à l'opérateur d'insertion sur la tournée de cet opérateur.

Question 19. Modifiez votre test de la classe **TestMeilleureInsertion** pour vérifier que :

- si un opérateur n'est pas réalisable, alors il n'est pas implémenté ;
- si un opérateur est réalisable, alors il est implémenté correctement (les modifications sur la tournée sont correctes).

Question 20. Dans la classe **Solution**, ajoutez une méthode **doInsertion(InsertionClient infos)** qui implémente le mouvement lié à l'opérateur d'insertion **infos** s'il est réalisable. Cette méthode renvoie un booléen qui indique si le mouvement d'insertion a bien été implémenté.

N'oubliez pas de mettre à jour le coût de la solution à l'aide des informations contenues dans le paramètre **infos**.

4.2.5 Algorithme de la meilleure insertion

Maintenant, nous avons mis en place tous les éléments nécessaires à l'élaboration de l'algorithme de la meilleure insertion. L'Algorithme 3 explicite l'algorithme de meilleure insertion. Notez en particulier que la ligne 4 de l'Algorithme 3 nécessite de parcourir tous les clients de $\mathcal{N}$ afin de trouver le meilleur opérateur d'insertion d'un de ces clients dans la solution courante $\mathcal{S}$.

Algorithme 3 : Algorithme de meilleure insertion.

```
1:  N = liste de clients
2:  S = solution vide (sans tournées)
3:  while N n'est pas vide do
4:      best = meilleur opérateur d'insertion dans S d'un client de N
5:      if le mouvement lié à best a pu être implémenté dans S then
6:          retirer le client lié à best de N
7:      else
8:          ajout du premier client de N dans une nouvelle tournée de S
9:          retirer le premier client de N
10:     end if
11: end while
12: return S
```

Question 21. Dans le paquetage **solveur**, ajoutez une classe **MeilleureInsertion** qui implémente l'interface **Solveur**. Implémentez les méthodes de l'interface **Solveur**. En particulier, la méthode **solve** devra avoir le comportement de l'algorithme de la meilleure insertion (Algorithme 3).

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 meilleur opérateur d'insertion d'un client de $\mathcal{N}$ dans la solution courante $\mathcal{S}$ (ligne 4 de l'Algorithme 3). Cette méthode doit parcourir tous les client de $\mathcal{N}$ et trouver le meilleur opérateur d'insertion d'un de ces clients dans $\mathcal{S}$.
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 22. Afin de bien tester l'algorithme de meilleure insertion et de pouvoir évaluer ses performances, nous allons ajouter ce solveur dans le test de la classe **TestAllSolveur**. Dans la classe **TestAllSolveur**, dans la méthode **addSolveurs()**, ajoutez dans la liste des solveurs une instance du solveur **MeilleureInsertion**.

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.

5 Algorithme de *Clarke and Wright*

Nous nous intéressons à présent 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 $(1, i, 1)$ 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 probablement 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.

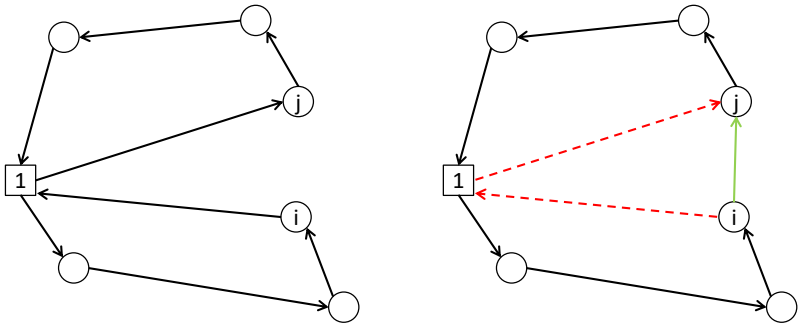
Plus précisément, la fusion d'une tournée $r = (1, ..., i, 1)$ (tournée qui termine par le client i) avec une tournée $s = (1, j, ..., 1)$ (tournée qui commence par le client j) va donner la tournée $t = (1, ..., i, j, ..., 1)$, i.e. que l'on va directement de i vers j sans repasser par le dépôt. Bien 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 $r = (1, ..., i, 1)$ et $s = (1, j, ..., 1)$ entraine un gain de :

$$g_{rs} = c_{ij} - c_{i1} - c_{1j}.$$

Il faut compter le coût pour aller directement de i vers j et retrancher les coûts pour aller de i vers le dépôt et du dépôt vers j. Ceci est illustré dans la Figure 2.

À chaque itération de l'algorithme de *Clarke and Wright*, on calcule tous les gains possibles en fusionnant toutes les paires de tournées de la solution, puis

on implémente réellement la fusion des deux tournées qui permettent le meilleur gain (le gain négatif le plus faible). La solution contient donc une tournée en moins à la fin de chaque itération.



(a) On souhaite fusionner la tournée qui termine par le client i et celle qui commence par le client j

(b) La tournée fusionnée visite le client j juste après le client i sans repasser par le dépôt

Figure 2 – Modifications du coût total d'une solution lors de la fusion de deux tournées. Les coûts à rajouter sont ceux des routes en vert, et les coûts à retirer sont ceux des routes en pointillés rouges.

Remarquez bien que l'algorithme de *Clarke and Wright* n'est pas similaire aux algorithmes d'insertion vu précédemment. En effet, ici, on construit très rapidement une solution initiale potentiellement de très mauvaise qualité. Puis, les itérations de l'algorithme permettent d'améliorer la solution jusqu'à ce qu'il n'y ait plus d'amélioration possible.

En ce qui concerne l'implémentation, nous allons suivre le même schéma que pour l'algorithme de meilleure insertion :

- évaluation du coût engendré par la fusion de deux tournées ;
- stockage des informations sur la fusion des tournées dans un opérateur ;
- algorithme pour la recherche de la meilleure fusion de tournées dans une solution ;
- implémentation du mouvement lié à un opérateur de fusion de tournées ;
- algorithme de *Clarke and Wright* pour résoudre le problème de CVRP.

Cependant, notez bien que, comme la démarche est la même, nous donnerons moins de détails. N'hésitez donc pas à prendre des initiatives.

L'Algorithme 4 vous présente la trame générale de l'algorithme de *Clarke and Wright* que vous pourrez implémenter lorsque vous aurez mis en place tous les éléments.

Algorithme 4 : Algorithme de *Clarke and Wright*

```
1: 𝒩 = liste de clients
2: 𝒮 = solution vide (sans tournées)
3: for all client k ∈ 𝒩 do
4:     ajouter k dans une nouvelle tournée de 𝒮
5: end for
6: fusion = vrai
7: while fusion est vrai do
8:     best = meilleur opérateur de fusion de deux tournées de 𝒮
9:     fusion = le mouvement lié à best a pu être implémenté dans 𝒮
10: end while
11: return 𝒮
```

Question 23. Dans la classe **Tournee**, ajoutez une méthode **deltaCoutFusion(Tournee aFusionner)** qui renvoie le coût engendré par la fusion de la tournée **aFusionner** avec la tournée (**this**). La Figure 2 vous montre comment ce coût est calculé.

Vous pourrez prendre la convention suivante : lors de la fusion, les clients de la tournée **aFusionner** sont ajoutés après les clients de la tournée courante **this**. Pour le coût, il faut donc considérer le dernier client de **this** et le premier client de **aFusionner**.

Question 24. Dans le paquetage **test**, ajoutez une classe **TestFusionTournees** dans laquelle vous ferez un test pour vérifier que le calcul du coût de la fusion de tournées est correct.

À nouveau, vous pouvez choisir des clients qui sont alignés afin de pouvoir facilement calculer les distances pour faire les vérifications.

L'opérateur de fusion de deux tournées est un opérateur pour lequel on précise la tournée qui va venir être fusionnée à la tournée de l'opérateur. L'opérateur de fusion de tournées sera donc caractérisé par deux tournées (une est déjà définie dans la classe abstraite **Operateur**, et l'autre sera appelée la tournée à fusionner). L'évaluation du coût de l'opérateur est réalisée grâce à la méthode d'évaluation du coût de fusion que nous avons développée préalablement dans la classe **Tournee**.

Question 25. Dans le paquetage **operateur**, ajoutez une classe **FusionTournees** qui hérite de **Operateur**. Définissez ses attributs et implémentez la méthode **evalDeltaCout()**. Pour le moment, pour implémenter la méthode **doMouvement()** un simple **return false;** suffira.

Ajoutez un constructeur par défaut, un constructeur par la donnée de la tournée, et de la tournée à fusionner (le coût est alors évalué avec la méthode **evalDeltaCout()**). Ajoutez les accesseurs nécessaires ainsi qu'une redéfinition de la méthode **toString**.

Question 26. Modifiez votre test de la classe **TestFusionTournees** pour vérifier que :

- vous arrivez bien à créer des objets de type **FusionTournees** ;
- les valeurs des attributs de ces objets sont corrects (en particulier le coût) ;
- les méthodes **isMeilleur** et **isMouvementRealisable** renvoient des résultats corrects.

Question 27. Dans la classe **Solution**, ajoutez une méthode **getMeilleureFusion()** qui renvoie le meilleur opérateur de fusion de tournées de la solution courante.

Vous êtes très fortement encouragés à déléguer une partie du traitement dans la classe **Tournee**. En particulier, notez bien qu'on ne peut fusionner des tournées que si la capacité de la tournée obtenue après fusion est satisfaite. Aussi, on ne fusionne pas une tournée avec elle-même, et on ne fusionne pas de tournées vides.

Question 28. Modifiez votre test de la classe **TestFusionTournees** pour vérifier que vous arrivez bien à trouver la meilleure fusion de tournées dans une solution.

Question 29. Dans la classe **Tournee**, ajoutez une méthode **doFusion(FusionTournees infos)** qui implémente le mouvement lié à l'opérateur de fusion de tournées **infos**. Cette méthode renvoie un booléen qui indique si le mouvement de fusion a bien été implémenté.

Il faut donc insérer les clients de la tournée à fusionner à la bonne position, et mettre correctement à jour les attributs de la tournée (coût total et demande totale). Pour cela on se sert des informations contenues dans le paramètre **infos**.
Vous pouvez faire appel à la méthode **check** de la classe **Tournee** à la fin de la méthode **doFusion** afin de vérifier que les attributs ont bien été mis à jour.

Question 30. Dans la classe **FusionTournees**, implémentez correctement la méthode **doMouvement** qui réalise le mouvement lié à l'opérateur de fusion de tournées.

Question 31. Dans la classe **Solution**, ajoutez une méthode **doFusion(FusionTournees infos)** qui implémente le mouvement lié à l'opérateur de fusion de tournées **infos** s'il est réalisable et qu'il améliore la solution. Cette méthode renvoie un booléen qui indique si le mouvement de fusion de tournées a bien été implémenté.

Il est important de bien noter ici que le mouvement n'est implémenté que s'il est améliorant, c'est-à-dire que son coût est strictement négatif. Nous vous conseillons ici d'ajouter dans la classe **Operateur** une méthode **isMouvementAmeliorant()** qui revoie un booléen indiquant si le mouvement lié à l'opérateur améliorera le coût de la solution ou non.
Par ailleurs, n'oubliez pas de mettre à jour le coût de la solution à l'aide des informations contenues dans le paramètre **infos**. Aussi, n'oubliez pas de supprimer de la solution la tournée qui a été fusionnée.

Question 32. Modifiez votre test de la classe **TestFusionTournees** pour vérifier que vous arrivez bien à implémenter le mouvement lié au meilleur opérateur de fusion de tournées d'une solution.

Question 33. Dans le paquetage **solveur**, ajoutez une classe **ClarkeAndWright** qui implémente l'interface **Solveur**. Implémentez les méthodes de l'interface **Solveur**. En particulier, la méthode **solve** devra avoir le comportement de l'algorithme de *Clarke And Wright* (Algorithme 4).

Testez que votre code fonctionne correctement.

Question 34. Afin de bien tester l'algorithme de *Clarke and Wright* 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 **ClarkeAndWright**.

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.

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.*