POO pour l'optimisation
IG2I, LE4-SI – 2020-2021

TP 1 – Mise en place du modèle objet

Diego Cattaruzza, Maxime Ogier

Contexte et objectif du TP

Ce TP ainsi que les TPs suivants seront l'occasion de présenter et d'implémenter différents algorithmes d'optimisation. Le problème étudié sera le même pour tous les TPs afin de pouvoir comparer l'efficacité des algorithmes. Nous étudierons le problème de tournées de véhicules, dénoté par la suite CVRP (*Capacitated Vehicle Routing Problem*). Nous choisissons ce problème car : (1) il n'est pas facile à résoudre (c'est un problème dit NP-difficile), (2) c'est un problème très étudié en informatique et plus particulièrement dans le domaine de la recherche opérationnelle, et (3) c'est un problème de base que l'on trouve très souvent dans l'industrie, en particulier dans les domaines de la logistique et du transport.

Afin de donner un contexte applicatif au problème de CVRP, nous pouvons reprendre celui vu en cours de LE2-POO, qui est décrit par la suite.

L'entreprise nordiste Bière2I est spécialisée dans la livraison des bières. Avec l'arrivée du printemps, elle doit faire face à une grosse demande de la part de ses différents clients. Cette année, la demande pour la bière de printemps est vraiment exceptionnelle et les gestionnaires de la livraison ne sont pas certains d'avoir le temps de pouvoir planifier toutes les livraisons à la main.

Pour améliorer son planning de livraison, Bière2I décide donc de faire appel aux meilleurs étudiants de la région Hauts-de-France pour concevoir des outils informatiques de qualité qui puissent fournir un planning de livraison capable de gérer des quantités de demandes importantes.

L'objectif des TPs en LE4-POO est de concevoir un outil très performant au niveau algorithmique pour aider Bière2I dans son travail. Dans ce premier TP, nous nous focalisons sur la mise en place du modèle objet. Ceci doit nous permettre de lire des instances et de proposer des solutions au problème de CVRP.

Ce TP permet d'illustrer les notions suivantes :

- la modélisation objet ;
- les notions d'instance et de solution ;
- de nouvelles structures de données : **LinkedHashMap** ;
- le développement d'un checker.

Bonnes pratiques à adopter

- Testez votre code au fur et à mesure de votre avancement.
- Respectez les conventions 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 Introduction

Nous allons détailler ici un contexte applicatif (le même que celui vu en LE2-POO), puis nous ferons le lien avec le problème général : le CVRP.

1.1 Contexte applicatif : Bière2I

Monsieur Houblon, le responsable de la logistique chez Bière2I, vous explique que son travail consiste à planifier la livraison d'une quantité de bière à un ensemble de clients. Chaque jour, des chauffeurs partent avec leur véhicule depuis le dépôt de la brasserie, visitent des clients et reviennent au dépôt. Pour utiliser la terminologie de Monsieur Houblon, ils accomplissent une *tournée*.

Le travail de Monsieur Houblon consiste à décider quel sous-ensemble des clients à livrer il faut affecter à chaque chauffeur, de façon à minimiser les coûts de transport (coût horaire des chauffeurs, coût du carburant) en respectant

aussi la capacité de chaque véhicule (nous ne pouvons pas livrer deux clients qui demandent 60 caisses de bouteilles chacun, si le véhicule ne peut en contenir que 100).

Monsieur Houblon vous explique que la brasserie ne possède pas de véhicules, mais elle paie chaque jour un sous traitant qui lui met à disposition autant de chauffeurs que nécessaire. Chaque chauffeur a son propre camion, arrive à la brasserie pour charger les bières, livre les clients, puis revient à la brasserie. Pour ce service, Monsieur Houblon doit payer un coût proportionnel à la distance parcourue par les chauffeurs pendant leur tournée. Voilà pourquoi il cherche à ce que cette distance soit la plus faible possible. Un exemple de planning de livraison est donné en Figure 1.

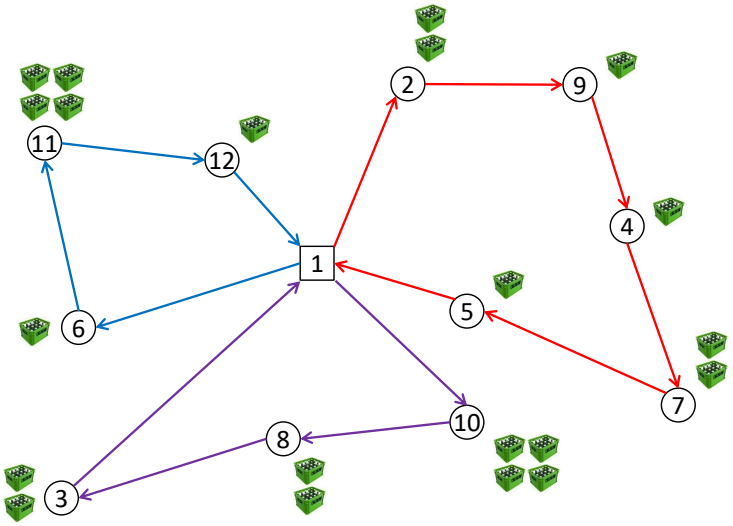


Figure 1 – Exemple de planification avec 3 tournées qui partent du dépôt (1) pour livrer les clients.

1.2 Le CVRP : *Capacitated Vehicle Routing Problem*

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 [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 : $\sum_{p=1}^{p=s} q_{i_p} \leq Q$.

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 Modélisation objet du CVRP

2.1 Modèle conceptuel de données

Suite à quelques discussions avec Monsieur Houblon, un modèle conceptuel de données est proposé dans la Figure 2. Notez bien que ce diagramme de classe n'est pas tout à fait identique à celui réalisé en LE2-POO.

Voici quelques indications supplémentaires par rapport au diagramme de classe.

- Les points du réseau routier sont caractérisés par un identifiant, leur abscisse et leur ordonnée. La classe **Point** sera une classe abstraite.
- Les dépôts sont des points spéciaux du réseau. Toutes les tournées partent du dépôt et y reviennent. La classe **Depot** hérite de la classe **Point**.
- Les clients sont des points spéciaux du réseau. Ils sont caractérisés par une demande. La classe **Client** hérite de la classe **Point**.
- Les routes modélisent le réseau routier et relient deux points du réseau. La classe **Route** est une classe d'association entre deux points (départ et arrivée) et est caractérisée par un coût (par exemple la distance pour aller du point de départ vers le point d'arrivée).

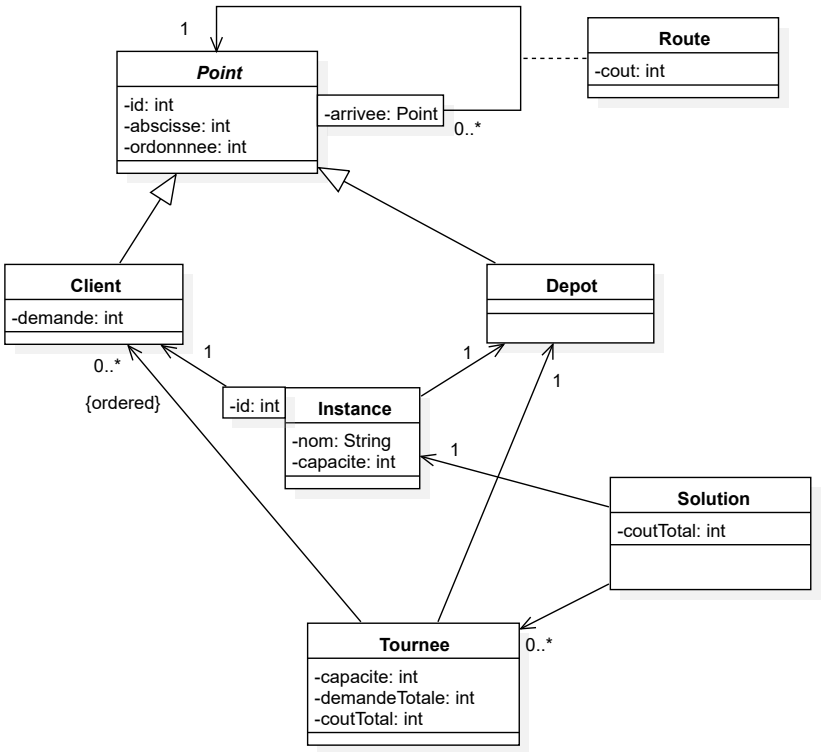


Figure 2 – Diagramme de classe pour le système d'information des notre application.

- Une instance d'un problème est obtenue en spécifiant les valeurs prises par toutes les données du problème. La classe **Instance** est donc caractérisée par un nom, la capacité Q des véhicules, le dépôt d'où partent toutes les tournées et l'ensemble des clients à livrer. Notez que les coûts des routes qui relient les points de l'instance sont accessibles à travers la classe d'association **Route** qui lie deux objets de type **Point** entre eux.
- Une tournée représente le trajet effectué par un véhicule pour livrer des clients. La classe **Tournee** est caractérisée par la capacité Q des véhicules (dupliqué de l'attribut du même nom dans la classe **Instance**), un dépôt (le même que celui de l'instance), et une collection triée de clients : les clients à visiter dans la tournée, dans l'ordre de visite. Par ailleurs, la classe **Tournee** est caractérisée par la demande totale livrée (somme des demandes des clients livrés dans la tournée) et le coût total pour livrer tous les clients de la tournée (somme des coûts des routes empruntées pour livrer les clients de la tournée).
- Une solution est liée à une instance et se compose d'un ensemble de tournées. La classe **Solution** est caractérisée aussi par le coût total (qui est la somme des coûts des tournées).

Notez bien que dans la modélisation proposée, nous avons essayé de bien distinguer l'instance et la solution. Ici, l'instance correspond aux données récupérées par Monsieur Houblon (capacité des véhicules, dépôt et clients à livrer), tandis que la solution correspond à ce que Monsieur Houblon doit proposer pour résoudre son problème de tournées de véhicules. Face à un problème d'optimisation, la première étape est toujours de clarifier à quoi correspondent une instance et une solution.

2.2 Modélisation du réseau routier

Nous nous intéressons dans un premier temps à la représentation des différentes localisations de l'instance ainsi qu'aux routes qui permettent de passer d'un point à l'autre. Chacune des localisations peut être considérée comme un point caractérisé par un identifiant (supposé unique), une abscisse et une ordonnée.

Question 1. Après avoir ouvert Netbeans, créez un nouveau projet nommé **CVRP**. Créez un paquetage **instance**, et dans ce paquetage un paquetage **reseau**. Ce dernier paquetage contiendra les classes pour la modélisation de l'infrastructure routière.

Écrivez une classe **Point** qui représente un point, dont vous définirez les attributs. Ajoutez un constructeur par données. Complétez la classe **Point** avec les accesseurs et mutateurs nécessaires, ainsi que les redéfinitions des méthodes **hashCode**, **equals** et **toString**.

Nous supposons ici que l'identifiant sera lu dans un fichier d'instance. Ce n'est donc pas à vous de le générer automatiquement. Par ailleurs, notez bien que les coordonnées des points sont entières. Des attributs qui ne changent jamais de valeur peuvent être déclarés **final**. Notez également que la classe **Point** est abstraite. N'oubliez pas que les assistants Netbeans sont là pour vous aider (clic-droit puis *Insert Code...*).

Toutes les tournées de livraison partent d'un même endroit : le dépôt (ici l'entrepôt de la brasserie). Lors des tournées de livraison, il faut livrer des clients (ici en leur fournissant des caisses de bières). Les clients sont donc des points spéciaux : en plus de leurs coordonnées, ils sont caractérisés par une demande représentant la quantité à leur livrer (ici le nombre de caisses de bière).

Question 2. Dans le paquetage **reseau**, ajoutez deux classes **Depot** et **Client** qui héritent de **Point**. Complétez ces classes avec les attributs, constructeurs par données et méthodes nécessaires. Faites un test pour vérifier que tout fonctionne correctement.

Question 3. Dans le paquetage **reseau**, ajoutez une classe **Route** qui représente la route depuis un point de départ vers un point de destination. Une route est également caractérisée par un attribut de type entier qui représente le coût engendré par le parcours de cette route. Ici, nous considérerons que ce coût est égal à l'arrondi (au nombre entier le plus proche) de la distance euclidienne entre le point de départ et le point d'arrivée. Faites un test pour vérifier que tout fonctionne correctement.

Notez bien que dans le diagramme UML de la Figure 2, la classe **Route** est une classe d'association entre deux points. Le coût de la route étant calculé à partir des points de départ et d'arrivée, il ne doit pas être en argument du constructeur. Pour le calcul du coût, n'hésitez pas à définir une méthode avec une visibilité **private**, et à utiliser des variables intermédiaires. Les méthodes statiques de la classe **Math** peuvent également être utiles. N'oubliez pas de redéfinir les méthodes **equals**, **hashCode** et **toString**.

Nous avons défini les points à visiter dans le réseau, ainsi que les routes qui relient ces points. L'idée est maintenant de relier ces objets. Nous avons déjà mémorisé dans la classe **Route** les points extrémités. Il nous reste, dans la classe **Point**, à mémoriser les routes qui permettent de se déplacer vers les autres points.

Question 4. Modifiez la classe **Point** afin de prendre en compte la classe d'association **Route**.

- Ajoutez un attribut qui permet de mémoriser l'ensemble des routes partant d'un point.
- Ajoutez une méthode **ajouterRoute(Point destination)** qui ajoute une route vers le point **destination**.
- Ajoutez une méthode **getCoutVers(Point destination)** qui renvoie le coût de la route qui va vers le point **destination** (valeur infinie si cette route n'existe pas).

Faites un test pour vérifier que tout fonctionne correctement.

Notez bien que dans le diagramme UML de la Figure 2, l'association entre deux objets de la classe **Point** (via la classe d'association **Route**) est une association qualifiée. Pour implémenter ce genre d'association, on utilise une table associative (type **Map** en Java).

2.3 Modélisation de l'instance

À présent, nous souhaitons modéliser une instance du problème de CVRP. D'après le diagramme UML de la Figure 2, une instance est caractérisée par un nom, la capacité des véhicules (ici le nombre de caisses de bières que l'on peut mettre dans un camion), un dépôt (d'où partent les véhicules) et les clients à livrer. Nous supposerons que le nom de l'instance est unique.

Question 5. Dans le paquetage **instance**, ajoutez une classe **Instance** dont vous définirez les attributs. Ajoutez un constructeur par la donnée du nom, de la capacité, et du dépôt. Ajoutez les accesseurs et mutateurs nécessaires ainsi que les redéfinitions utiles de méthodes de la classe **Object**.

Notez bien que dans le diagramme UML de la Figure 2, l'association entre une instance et un client est une association qualifiée. Pour implémenter ce genre d'association, on utilise une table associative (type **Map** en Java). Par ailleurs, notez qu'il est tout de même utile de savoir dans quel ordre les éléments de la table associative seront parcourus. Pour ce faire, il existe en Java l'implémentation **LinkedHashMap** qui permet de savoir dans quel ordre est réalisé le parcours des éléments. Vous êtes très vivement encouragés à regarder la Javadoc afin d'avoir plus d'informations sur les **LinkedHashMap**. Par ailleurs, notez que les clients ne sont pas en paramètres du constructeur par données. Nous ajouterons par la suite une méthode qui permet d'ajouter un par un les clients à l'instance. Ceci permet un meilleur contrôle pour l'objet **Instance** sur son ensemble de clients.

Question 6. Modifiez la classe **Instance** en ajoutant quelques méthodes qui seront utiles par la suite.

- Ajoutez une méthode **getNbClients()** qui renvoie le nombre de clients de l'instance.
- Ajoutez une méthode **getClientById(int id)** qui renvoie le client de l'instance ayant l'identifiant **id** (**null** si aucun client n'a cet identifiant).
- Ajoutez une méthode **getClients()** qui renvoie tous les clients de l'instance sous forme d'une liste chaînée.
- Ajoutez une méthode **ajouterClient(Client clientToAdd)** qui ajoute le client **clientToAdd** aux clients de l'instance. Cette méthode ajoute également toutes les routes : entre le dépôt et **clientToAdd** et entre les clients déjà présents dans l'instance et **clientToAdd**. Cette méthode renvoie un booléen indiquant si l'ajout a effectivement eu lieu ou non.

Si vous avez bien utilisé une **Map** pour stocker les clients, la méthode **getClientById(int id)** est très simple à implémenter.
Pour implémenter la méthode **getClients()** pensez bien à respecter le principe d'encapsulation : il existe, pour les classes implémentant l'interface **List**, un constructeur par copie. Cela permet d'éviter de donner un accès public à la référence sur les clients de l'instance. Par ailleurs, cela permet, pour la méthode appelante, de trier la liste ou supprimer des éléments sans que cela impacte la liste des clients dans l'instance.

Pour implémenter la méthode **ajouterClient(Client clientToAdd)**, n'oubliez pas de garder des méthodes courtes. N'hésitez pas à introduire des méthodes avec une visibilité **private** et d'effectuer une partie du traitement algorithmique dans ces méthodes. Aussi, rappelez-vous qu'entre deux points i et j, il y a deux routes : une route qui va de i vers j, et une route qui va de j vers i.

2.4 Lecture des instances

Le problème de CVRP est un problème bien connu en informatique. Nous chercherons donc à le résoudre pour quelques instances classiques de la littérature scientifique. Notez bien que ces instances représentent souvent des cas fréquents dans la pratique : certains clients sont regroupés au même endroit, certains client peuvent demander une très grande quantité, le dépôt peut être excentré par rapport aux clients.

Vous trouverez sur Moodle un répertoire *instances* qui contient 10 instances, au format '.vrp'. Vous trouverez également sur Moodle un répertoire *io* qui contient un lecteur d'instance (une classe Java). Vous utiliserez donc ce lecteur d'instance. Il suffit juste de vérifier que les signatures de certaines méthodes et des constructeurs sont concordants avec votre propre code.

Question 7. Mettez le répertoire *instances* dans le dossier de votre projet (au même niveau que le répertoire *src*). Mettez le répertoire *io* dans le répertoire *src* de votre projet (**io** est le nom du paquetage). Vérifiez que la classe **InstanceReader** du paquetage **io** 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.

Tester cette classe afin de vérifier que vous arrivez bien à lire une instance (une méthode **main** est présente dans la classe).

Question 8. À présent, testez la classe **Instance** pour vérifier que tout fonctionne correctement.

Vous pouvez, par exemple, lire l'instance 'A-n32-k5.vrp', puis vérifier que son nom est correct, la capacité des camions est 100, le dépôt a l'identifiant 1 et ses coordonnées sont (82;76), il y a bien 31 clients, la somme des demandes des clients est 410, les routes ont des coûts positifs et pas infinis.

2.5 Modélisation des solutions

À présent, nous nous intéressons à la dernière étape de modélisation de ce TP : les solutions du problème de CVRP. Une solution est liée à une instance particulière. Elle est composée d'un ensemble de tournées, et elle est caractérisée par un coût total qui représente la somme des coûts des tournées. Une tournée représente le trajet effectué par un camion pour livrer des clients. Elle est donc caractérisée par un dépôt qui représente son point de départ et d'arrivée, et par les clients à visiter, dans l'ordre dans lequel ils sont visités. Une tournée est également caractérisée par la capacité du véhicule, la demande totale (somme des demandes des clients livrés dans la tournée) et le coût total (somme des coûts des routes empruntées par la tournée).

Remarquez bien que dans la tournée, la notion d'ordre de visite des clients est très importante. En effet, c'est cela qui nous permet de déterminer les routes empruntées par la tournée, et donc le coût total de la tournée. Il faudra donc choisir une structure de données adaptée.
Par ailleurs, notez bien que certaines données sont redondantes, mais cela est fait exprès. En effet, dans une tournée, on mémorise la demande totale et le cout total, alors que l'on peut les recalculer "facilement" à partir des clients visités. Cependant, un tel calcul a une complexité $O(n)$ avec n le nombre de clients dans la tournée. Si ce calcul est souvent effectué, alors cela peut nuire grandement au temps d'exécution du code. Maintenir les données redondantes sur la demande totale et le coût total permet de récupérer ensuite ces informations en temps constant $O(1)$. Par contre, cela vous oblige à modifier correctement ces valeurs à chaque fois que vous modifiez les clients à visiter dans une tournée. Dans la mesure du possible, on essaye également de faire ces modifications en temps constant $O(1)$. Il en est de même pour le coût total de la solution.
Enfin, notez bien que la capacité et le dépôt d'une tournée sont identiques à ceux de l'instance liée à la solution à laquelle appartient la tournée. Cette duplication d'information permet d'alléger l'écriture du code de la classe **Tournee** : on accède directement à la capacité et au dépôt sans utiliser un accesseur sur un objet de la classe **Instance**. Par exemple, dans la classe **Tournee**, on écrira directement **this.capacite** au lieu de **this.instance.getCapacite()**.

Question 9. Créez un paquetage **solution**. Dans ce paquetage, écrivez une classe **Tournee** dont vous définirez les attributs. Ajoutez un constructeur par la donnée de l'instance. Ajoutez les accesseurs et mutateurs nécessaires ainsi que les redéfinitions utiles de méthodes de la classe **Object**. Faites un test pour vérifier que tout fonctionne correctement.

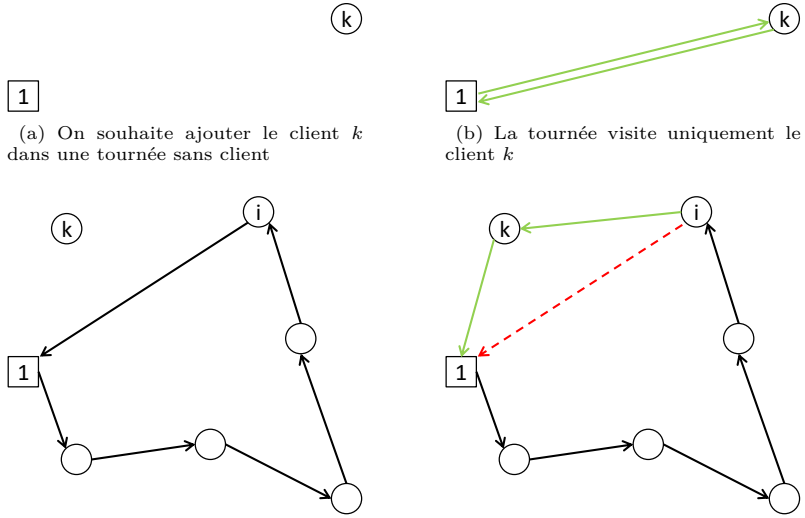
Prenez bien le temps de réfléchir à la structure de données pour stocker les clients visités. Dans le constructeur, l'instance en paramètre doit juste servir à initialiser la capacité et le dépôt. Rappelez-vous également que des attributs qui ne changent jamais de valeur peuvent être déclarés **final**. Pensez bien à ce qui fait l'égalité entre deux tournées. Par ailleurs, afin de pouvoir débugger facilement votre code par la suite, il est intéressant de faire dès maintenant une méthode **toString** propre et informative.

Nous allons maintenant nous focaliser sur l'ajout d'un client dans une tournée. Pour le moment, pour commencer par des choses simples, nous allons considérer qu'un client est ajouté à la fin de la tournée (c'est-à-dire avant le retour au dépôt). Dans un premier temps, notez bien que l'ajout d'un client dans une tournée n'est réalisable que si la demande totale de la tourne plus la demande du client à ajouter est inférieure ou égale à la capacité de la tournée. Ensuite, comme évoqué précédemment, ce qui est important lors de l'ajout d'un client dans une tournée, c'est de mettre à jour les valeurs de la demande totale et du coût total de la tournée, si possible en temps constant $O(1)$. Pour la mise à jour de la demande totale, c'est très simple (une addition suffira). Pour la mise à jour du coût total, c'est un petit peu plus compliqué. Évidemment, on pourrait, après avoir ajouté le nouveau client, additionner les coûts de toutes les routes de la nouvelle tournée. Mais ceci serait fait en temps linéaire $O(n)$ avec n le nombre de clients dans la tournée. La Figure 3 permet d'expliquer comment on peut mettre à jour la distance totale en temps constant. Pour rappel, le dépôt est noté 1, et on note c_{ij} le coût entre deux points i et j. Il y a deux cas à considérer.

- Si on veut ajouter un nouveau client k dans une tournée qui n'a pas de clients (voir Figure 3a), alors le coût total de la tournée est juste $c_{1k} + c_{k1}$ (voir Figure 3b).
- Si on veut ajouter un nouveau client k dans une tournée qui contient déjà des clients et dont on note i le dernier client (voir Figure 3c), alors on ajoute au coût total $c_{ik} + c_{k1} - c_{i1}$ (voir sur la Figure 3d les routes en vert qui sont ajoutées et celle en pointillés rouges qui est retirée). Au passage, on observe bien sur la Figure 3d que toutes les routes en noir ne changent pas lors de l'ajout du client k, et c'est pourquoi recalculer le coût de toute la tournée en $O(n)$ est vraiment peu efficace.

Question 10. Ajoutez dans la classe **Tournee** une méthode **ajouterClient(Client clientToAdd)**. Cette méthode ajoute le client **clientToAdd** en dernière position de la tournée, si l'ajout est possible. Elle renvoie un booléen qui indique si l'ajout a pu se faire ou non.

Faites un test pour vérifier que tout fonctionne correctement.



(a) On souhaite ajouter le client k dans une tournée sans client

(b) La tournée visite uniquement le client k

(c) On souhaite ajouter le client k après le dernier client i d'une tournée

(d) La tournée visite le client k après le client i et avant de revenir au dépôt

Figure 3 – Modifications du coût total d'une tournée lors de l'ajout d'un client k en fin de tournée. Les coûts à rajouter sont ceux des routes en vert, et les coûts à retirer sont eux des routes en pointillés rouges.

N'oubliez pas qu'un ajout peut être impossible à cause de la capacité du camion. Lors de l'ajout d'un client, mettez bien à jour les attributs en temps constant (voir Figure 3d).
Pour implémenter la méthode **ajouterClient(Client clientToAdd)**, n'oubliez pas de garder des méthodes courtes. N'hésitez pas à introduire des méthodes avec une visibilité **private** et d'effectuer une partie du traitement algorithmique dans ces méthodes.
Pour le test, vous pouvez par exemple prendre l'instance 'A-n32-k5.vrp', créer une tournée, puis essayer d'y ajouter successivement tous les clients de l'instance. Enfin, vérifiez que la tournée obtenue est correcte.

Question 11. Dans le paquetage **solution**, ajoutez une classe **Solution** dont vous définirez les attributs. Ajoutez un constructeur par la donnée de l'instance. Ajoutez les accesseurs et mutateurs nécessaires ainsi que les redéfinitions utiles de méthodes de la classe **Object**. Faites un test pour vérifier que tout fonctionne correctement.

Pour la structure de données qui stocke les tournées, nous vous conseillons d'utiliser une liste. Cette structure de données nous sera utile pour les TPs suivants lors du développement des algorithmes de résolution. En effet, utiliser une liste permet d'indexer les tournées par un nombre entier, et cela permet également d'avoir accès à la dernière tournée ajoutée à la solution.
Pensez bien à ce qui fait l'égalité entre deux solutions. Par ailleurs, afin de pouvoir débugger facilement votre code par la suite, il est intéressant de faire dès maintenant une méthode **toString** propre et informative.

Question 12. 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.

2.6 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 cela permettra de détecter très facilement un bug dans le code : à chaque nouvelle solution, on fera appel au checker.

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 13. Dans la classe **Solution**, ajouter une méthode **check()** qui renvoie un booléen 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.

References

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