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

TP 4 – Recherche locale – 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*). Dans le TP précédent, vous avez construit des solutions réalisables pour le problème de CVRP. Normalement, vous avez dû constater que vous étiez capable de produire en quelques millisecondes des solutions de bien meilleure qualité qu'avec l'insertion simple.

Cependant, il est probable qu'on soit capable de produire des solutions de meilleure qualité, en ajoutant un peu de temps de résolution.

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

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

Ce TP permet d'illustrer les notions suivantes :

- le concept de transformation locale (autrement dit opérateur) ;
- les opérateurs d'amélioration *intra-tournée* ;
- le *design pattern Factory*.

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 Opérateurs de recherche locale et voisinages

1.1 Notions d'opérateur et de voisinage

Supposons que vous obtenez une solution $\mathcal{S}$ avec une des méthodes constructives codées lors du TP précédent. La solution $\mathcal{S}$ que vous avez obtenue appartient à l'ensemble $\mathcal{SOL}$ de toutes les solutions réalisables de votre problème de CVRP. La solution $\mathcal{S}$ que vous avez obtenue n'est pas forcément la meilleure solution que vous pouvez obtenir. Dans ce TP nous allons nous intéresser à la manière d'améliorer la qualité de cette solution.

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

Notez que vous avez déjà rencontré cette notion d'opérateur lors du TP précédent. Dans le cadre de la meilleure insertion, un opérateur d'insertion permettait d'ajouter un nouveau client à une solution courante, afin d'arriver à la fin à une solution réalisable. Dans le cadre de la recherche locale, un opérateur permet de passer d'une solution réalisable à une autre solution réalisable.

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

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

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

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

Il est important de bien noter ici que ces notions d'opérateur, de voisinage et de recherche locale ne sont pas spécifiques au problème de CVRP. Cela peut s'appliquer à n'importe quel problème d'optimisation.
Par ailleurs, pour chaque problème, il peut y avoir plusieurs opérateurs différents. Nous allons voir dans ce TP quelques opérateurs très classiques pour le problème de CVRP. Face à un autre problème, il faudrait réfléchir à des opérateurs spécifiques pour ce problème.
Enfin, gardez bien en tête que, pour que l'exploration d'un voisinage soit efficace (en terme de temps d'exécution), **il faut être capable de pouvoir évaluer très efficacement le coût d'une solution voisine, sans devoir recalculer le coût de toute la solution.**

1.2 Mise en place

Dans la suite du TP, et lors du TP suivant, nous allons implémenter 4 opérateurs de recherche locale. Nous les présentons très rapidement ici, et chaque opérateur sera détaillé dans la suite du sujet.

- Deux opérateurs *intra-tournée* qui modifient l'ordre des clients à l'intérieur d'une seule tournée :
  - l'opérateur de déplacement intra-tournée : cet opérateur déplace un client dans la même tournée, à une position différente ;
  - l'opérateur d'échange intra-tournée : cet opérateur échange les positions de deux clients au sein d'une même tournée.
- Deux opérateurs *inter-tournées* qui modifient les clients présents dans deux tournées :
  - l'opérateur de déplacement inter-tournées : cet opérateur déplace un client depuis une tournée vers une autre tournée ;
  - l'opérateur d'échange inter-tournées : cet opérateur échange deux clients de deux tournées différentes.

Comme nous l'avons fait pour l'opérateur d'insertion dans le TP précédent, chaque opérateur de recherche locale sera représenté par une classe qui hérite de la classe abstraite **Operateur**. Cependant, tous les opérateurs de recherche locale ont des similarités, et les opérateurs intra-tournée d'une part, et inter-tournées d'autre part, ont chacun des similarités. Afin d'éviter d'écrire du code redondant, nous allons ici mettre en place une hiérarchie de classes abstraites dont les 4 classes des opérateurs de recherche locale hériteront. Le diagramme de classe proposé en Figure 1 présente la hiérarchie de ces classes abstraites ainsi que les 4 classes concrètes qui correspondent aux opérateurs de recherche locale que nous souhaitons implémenter.

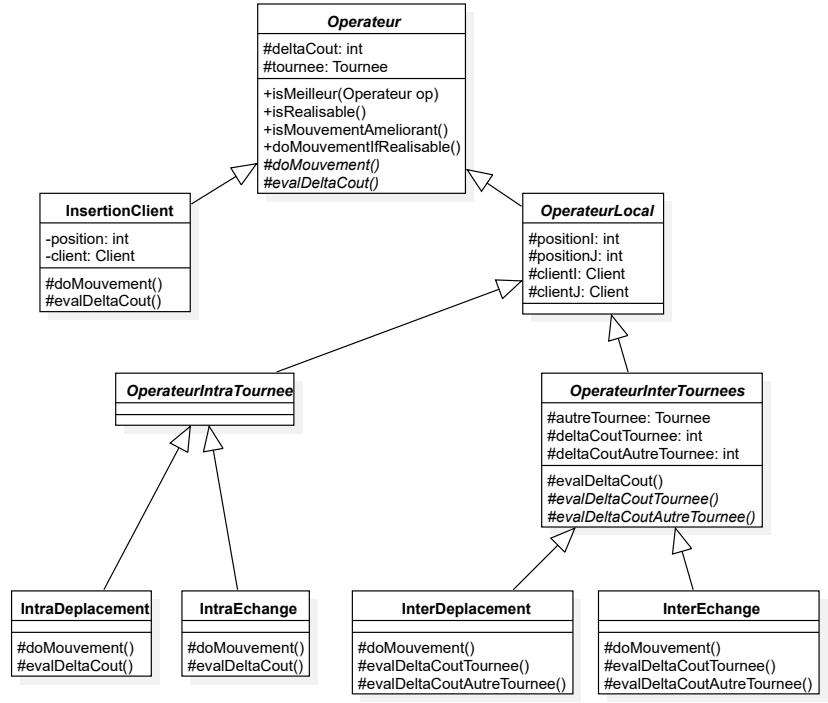


Figure 1 – Diagramme de classe pour les différents types d'opérateurs.

La classe mère de cette hiérarchie est la classe abstraite **Operateur** que vous avez déjà implémentée. Elle est définie par **deltaCout**, le coût supplémentaire engendré par la solution si l'on applique l'opérateur, ainsi que la tournée sur laquelle s'applique l'opérateur. La classe **Operateur** définit deux méthodes abstraites : **doMouvement()** et **evalDeltaCout()**.

La classe **InsertionClient** correspond à l'opérateur d'insertion d'un nouveau client. Vous avez déjà implémenté cette classe lors du TP précédent pour l'algorithme de la meilleure insertion. En particulier dans cette classe, vous avez implémenté les deux méthodes abstraites définies dans la classe mère **Operateur**.

La classe **OperateurLocal** est une classe abstraite. Elle est la classe mère de tous les opérateurs de recherche locale. Comme il s'agit d'un opérateur, elle est une classe fille de **Operateur**. En plus d'être défini par un coût et une tournée (dans la classe **Operateur**), un opérateur de recherche locale est défini par :

- les deux clients qui peuvent être affectés par l'opérateur ; on les nomme ici **ClientI** et **ClientJ** ;
- les positions respectives de ces clients dans leur tournée ; on les nomme ici **positionI** et **positionJ**.

Les classes **OperateurIntraTournee** et **OperateurInterTournees** sont des classes abstraites qui héritent de **OperateurLocal** et permettent de distinguer facilement les opérateurs intra-tournée et inter-tournées.

Par ailleurs, les opérateurs inter-tournées sont définis par trois attributs supplémentaires :

- **autreTournee** qui est donc la seconde tournée (il faut bien deux tournées pour un opérateur inter-tournées) ;
- **deltaCoutTournee** qui est le coût supplémentaire engendré par l'application de l'opérateur inter-tournées sur la tournée **tournee** (celle définie dans la classe **Operateur**) ;
- **deltaCoutAutreTournee** qui est le coût supplémentaire engendré par l'application de l'opérateur inter-tournées sur la tournée **autreTournee**.

Il faut bien noter ici qu'il faut calculer les coûts dans chacune des deux tournées, et on obtient le coût total engendré par l'application de l'opérateur sur la solution (**deltaCout**) en faisant la somme des coûts sur les deux tournées. Ainsi, la méthode abstraite **evalDeltaCout** est implémentée dans la classe **OperateurInterTournees**, et cette même classe définit deux méthodes abstraites qui ont pour fonction de calculer le coût supplémentaire dans chacune des deux tournées d'un opérateur inter-tournées.

Enfin, les 4 opérateurs de recherche locale que nous allons étudier dans ce TP correspondent à une classe concrète : **IntraDeplacement**, **IntraEchange**, **InterDeplacement** et **InterEchange**. Chacune de ces classes concrètes hérite de **OperateurIntra** ou **OperateurInter**, et implémente en conséquence les méthodes abstraites définies dans les classes mères : la méthode **doMouvement()** et une ou deux méthodes pour le calcul du coût supplémentaire engendré par l'application de l'opérateur.

Question 1. Si vous ne l'avez pas déjà fait lors du TP précédent, ajoutez dans la classe **Operateur** une méthode **isMouvementAmeliorant()** qui renvoie un booléen indiquant si le mouvement lié à l'opérateur est améliorant ou non.

Notez bien que nous avons défini **deltaCout** comme le coût supplémentaire engendré par la solution si l'on applique l'opérateur. Ainsi, une valeur de **deltaCout** négative signifie que le mouvement est améliorant.
Cette méthode nous servira par la suite, car seuls les mouvements améliorant la solution seront implémentés.

Question 2. Dans le paquetage **operateur**, ajoutez la classe abstraite **OperateurLocal** qui hérite de **Operateur**. Définissez ses attributs, et ajoutez un constructeur par défaut.

Ajoutez également un constructeur par la donnée de la tournée, et des deux positions des clients dans cette tournée. Les deux attributs **ClientI** et **ClientJ** seront alors initialisés dans ce constructeur à partir de la tournée et des positions. En revanche, l'attribut **deltaCout** ne sera pas encore initialisé.

Ajoutez les accesseurs nécessaires ainsi qu'une redéfinition de la méthode **toString**.

Pour récupérer le client à une certaine position d'une tournée, vous allez certainement développer une méthode dans la classe **Tournee**. Faites bien en sorte que si la position passée en paramètre de cette méthode est incorrecte, alors la méthode ne déclenche pas d'exception, mais renvoie **null**. Ceci nous permettra d'appeler cette méthode sans avoir à se soucier de la validité de la position.
Par ailleurs, notez qu'on fait le choix ici de ne pas initialiser l'attribut **deltaCout**. Pour initialiser cet attribut, il faudrait faire appel à la méthode **evalDeltaCout()**. Or, cette dernière aura besoin de l'attribut **autreTournee** dans le cas des opérateurs inter-tournées. Mais l'attribut **autreTournee** ne sera initialisé que dans le constructeur de la classe **OperateurInterTournees**. C'est pourquoi l'initialisation de l'attribut **deltaCout** se fera dans les classes filles de **OperateurLocal**.

Question 3. Dans le paquetage **operateur**, ajoutez la classe abstraite **OperateurIntraTournee** qui hérite de **OperateurLocal**. Ajoutez un constructeur par défaut. Ajoutez également un constructeur par la donnée de la tournée, et des deux positions des clients dans cette tournée. L'attribut **deltaCout** sera aussi initialisé dans ce constructeur.

Notez que l'on peut faire appel à la méthode abstraite **evalDeltaCout()** même si elle ne sera implémentée que dans les classes filles. C'est un des intérêts des classes abstraites.

Question 4. Dans le paquetage **operateur**, ajoutez la classe abstraite **OperateurInterTournees** qui hérite de **OperateurLocal**. Définissez ses attributs, et ajoutez un constructeur par défaut.

Ajoutez également un constructeur par la donnée des deux tournées et des positions des clients dans ces tournées. L'attribut **deltaCout** sera aussi initialisé dans ce constructeur.

Ajoutez les accesseurs nécessaires ainsi qu'une redéfinition de la méthode **toString**.

N'oubliez pas de redéfinir correctement l'attribut **ClientJ** (celui qui est dans la tournée **autreTournee**) dans le constructeur par données.
Pour le moment, initialisez simplement **deltaCout** en faisant appel à la méthode abstraite **evalDeltaCout()** et ne vous souciez pas d'initialiser les attributs **deltaCoutTournee** et **deltaCoutAutreTournee**.

Question 5. Dans la classe abstraite **OperateurInterTournees** ajoutez les deux méthodes abstraites **evalDeltaCoutTournee()** et **evalDeltaCoutAutreTournee()**. Ces méthodes renvoient un nombre entier qui correspond au surcoût engendré par l'application de l'opérateur sur chacune des tournées.

Implémentez la méthode abstraite **evalDeltaCout()** qui renvoie le surcoût engendré par l'application de l'opérateur sur la solution. Cette méthode met également à jour les attributs correspondant aux surcoûts sur les tournées.

Notez bien que le surcoût engendré par l'application de l'opérateur sur la solution correspond à la somme des surcoûts sur chacune des tournées. Cependant, faites bien attention que si une des tournées a un surcoût infini (le mouvement n'est pas réalisable), alors le surcoût engendré sur la solution doit aussi être infini.

1.3 *Design pattern Factory*

À présent, avant de commencer l'implémentation des opérateurs de recherche locale, nous allons régler un dernier point dans notre code.

Nous avons prévu de créer dans notre code deux opérateurs intra-tournée et deux opérateurs inter-tournées. Notez d'ailleurs qu'on pourrait tout à fait en avoir plus que deux. Or, comme chaque opérateur est différent et qu'ils héritent d'une classe abstraite (**OperateurIntraTournee** ou **OperateurInterTournees**), nous allons devoir définir des constructeurs dans chaque classe concrète qui représente un opérateur. Jusqu'ici il n'y a pas de problèmes.

Par contre, lors du développement du code nous allons devoir faire des traitements très similaires dans les classes **Tournee** et **Solution** en particulier. Ce qui changera dans ces traitements est juste l'appel aux constructeurs des différents opérateurs. Voici un exemple. Dans la classe **Tournee**, nous devrons faire une méthode qui renvoie le meilleur opérateur qui s'applique sur une tournée (avec les meilleures positions). Voici par exemple le code que vous pourriez écrire dans la classe **Tournee** pour trouver le meilleur opérateur de déplacement intra-tournée (**IntraDeplacement**).

```
1  public OperateurLocal getMeilleurIntraDeplacement() {
2     OperateurLocal best = new IntraDeplacement();
3     for(int i=0; i<clients.size(); i++) {
4        for(int j=0; j<clients.size()+1; j++) {
5           OperateurLocal op = new IntraDeplacement(this, i, j);
6           if(op.isMeilleur(best)) {
7              best = op;
8           }
9        }
10    }
11    return best;
12 }
```

Si vous voulez ensuite écrire dans la classe **Tournee** une méthode qui renvoie le meilleur opérateur d'échange intra-tournée (**IntraEchange**), vous allez écrire le code suivant.

```
1  public OperateurLocal getMeilleurIntraEchange() {
2     OperateurLocal best = new IntraEchange();
3     for(int i=0; i<clients.size(); i++) {
4        for(int j=0; j<clients.size()+1; j++) {
5           OperateurLocal op = new IntraEchange(this, i, j);
6           if(op.isMeilleur(best)) {
7              best = op;
8           }
9        }
10    }
11    return best;
12 }
```

Vous remarquez certainement que vous venez d'écrire deux fois le même code ; la seule différence étant l'appel aux constructeurs. Ce n'est jamais bon d'écrire du code redondant, notamment car c'est très difficile et pénible à maintenir et à modifier. Nous ne présentons que l'exemple avec deux opérateurs intra-tournée, mais imaginez si vous en rajoutez encore plusieurs autres. Il faudrait donc trouver une manière de remédier à ce problème.

Pour cela, nous pouvons utiliser le *design pattern Factory*. Pour rappel, un *design pattern* est une solution générale à un problème de conception récurrent dans de nombreuses applications. Les concepteurs de l'application doivent adapter la solution du *design pattern* à leur projet spécifique. Ce problème de l'instanciation de classes concrètes qui héritent d'une même classe abstraite ou implémentent une même interface est très souvent rencontré. Le *design pattern Factory* apporte une solution à ce type de problème. Il s'agit en fait d'utiliser un objet (on parle de fabrique, *factory* en anglais) qui sera chargé de l'instanciation des objets. Ainsi, l'instanciation est centralisée en un seul endroit ce qui est beaucoup plus facile en terme de maintenance du code.

Ainsi, voici comment on peut procéder pour résoudre le problème évoqué dans notre exemple. Dans un premier temps, on crée un énumération avec un nom pour chaque opérateur.

```
1  public enum TypeOperateurLocal {
2     INTRA_DEPLACEMENT,
3     INTRA_ECHANGE;
4  }
```

Puis, on crée la fabrique. Ici, nous allons en faire une pour les constructeurs par défaut, et une autre pour les constructeurs par données. En fait, il faut une fabrique pour chaque signature de constructeur.

```
1  public static OperateurLocal getOperateur(TypeOperateurLocal type) {
2     switch(type) {
3        case INTRA_DEPLACEMENT:
4           return new IntraDeplacement();
5        case INTRA_ECHANGE:
6           return new IntraEchange();
7        default:
8           return null;
9     }
10 }
```

```
1  public static OperateurIntraTournee
       getOperateurIntra(TypeOperateurLocal type, Tournee tournee, int
       positionI, int positionJ) {
2     switch(type) {
3        case INTRA_DEPLACEMENT:
4           return new IntraDeplacement(tournee, positionI, positionJ);
5        case INTRA_ECHANGE:
6           return new IntraEchange(tournee, positionI, positionJ);
7        default:
8           return null;
9     }
10 }
```

Notez que chaque fabrique est en fait une méthode statique dans laquelle un **switch** permet de sélectionner le bon constructeur en fonction de la valeur du paramètre de la méthode. Une fabrique peut être implémentée dans une classe spécifique ou dans la classe abstraite mère des classes concrètes. Ici, on pourrait mettre les deux fabriques précédentes dans la classe **OperateurLocal**.

Ainsi, l'exemple précédent dans la classe **Tournee** peut être réécrit de la manière suivante.

```
1  public OperateurLocal getMeilleurOperateurIntra(TypeOperateurLocal
       type) {
2     OperateurLocal best = OperateurLocal.getOperateur(type);
3     for(int i=0; i<clients.size(); i++) {
4        for(int j=0; j<clients.size()+1; j++) {
5           OperateurIntraTournee op =
          OperateurLocal.getOperateurIntra(type, this, i, j);
6           if(op.isMeilleur(best)) {
7              best = op;
8           }
9        }
10    }
11    return best;
12 }
```

On n'a donc plus qu'une seule méthode dans la classe **Tournee** sans duplication de code. Si on veut ajouter un nouvel opérateur, on modifie seulement les fabriques.

Question 6. Dans le paquetage **operateur**, ajoutez un type énuméré **TypeOperateurLocal** qui liste les 4 opérateurs de recherche locale que nous implémenterons dans ce TP.

Question 7. Dans la classe **OperateurLocal**, ajoutez les trois fabriques suivantes :

- une fabrique pour les constructeurs par défaut des opérateurs de recherche locale ;

- une fabrique pour les constructeurs par données des opérateurs de recherche locale intra-tournée ;
- une fabrique pour les constructeurs par données des opérateurs de recherche locale inter-tournées.

Pour le moment, vous n'avez pas implémenté les classes concrètes des différents opérateurs de recherche locale, donc ces fabriques ne contiennent que la structure générale : il faut mettre les bons paramètres, et mettre en place le **switch**.

2 Opérateurs intra-tournée

Dans cette section nous allons nous concentrer sur l'optimisation d'une seule tournée de notre solution $\mathcal{S}$ obtenue grâce aux méthodes constructives du TP précédent. Pour optimiser une tournée de la solution, nous allons implémenter des opérateurs de recherche locale intra-tournée :

- l'opérateur de déplacement qui déplace un client à une position différente dans la tournée ;
- l'opérateur d'échange qui échange les positions de deux clients au sein de la tournée.

En ce qui concerne l'implémentation, nous allons suivre, pour chaque opérateur, le même schéma que nous avons appliqué lors du TP précédent pour l'algorithme de meilleure insertion :

- évaluation du coût engendré par l'application de l'opérateur ;
- stockage des informations sur l'opérateur dans une classe héritant de **OperateurIntraTournee** ;
- algorithme pour la recherche du meilleur opérateur dans une solution ;
- implémentation du mouvement lié à l'opérateur ;
- algorithme itératif pour appliquer l'opérateur tant qu'il améliore la solution courante.

2.1 Opérateur de déplacement

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

Le voisinage $\mathcal{O}_{intra-dep}(\mathcal{S})$ contient $\sum_{k\in\mathcal{K}} r_k \cdot r_{k-1}$ solutions différentes, où r_k est le nombre de clients desservis par la tournée k, et $\mathcal{K}$ est l'ensemble des tournées dans la solution courante $\mathcal{S}$.

2.1.1 Évaluation du coût de l'opérateur de déplacement intra-tournée

Le mouvement lié à l'opérateur de déplacement intra-tournée correspond à déplacer le client i d'une tournée avant le point j de cette même tournée. Le point j peut être un client ou le dépôt. La Figure 2 présente trois exemples de déplacement d'un client i avant un client j d'une même tournée. Vous pouvez remarquer que le déplacement du client i avant le point j correspond en fait à deux mouvements simultanés :

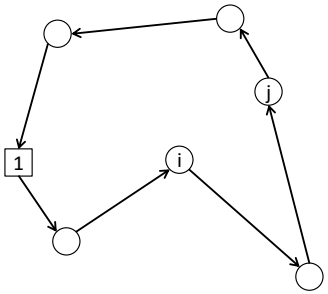
- la suppression du client i de sa position actuelle : on retire les routes qui arrivaient et partaient de i et on rajoute la route qui va du prédécesseur de i au successeur de i ;
- l'insertion de i avant le point j comme nous l'avons vu lors du TP précédent pour l'opérateur d'insertion : on retire la route qui arrivait en j et on rajoute la route qui va du prédécesseur de j vers i et la route qui va de i vers j.

La Figure 2 ne présente pas le cas où le point j est le dépôt. N'hésitez pas à faire votre propre dessin afin de bien visualiser ce qui se passe dans ce cas.
Notez bien que si le client i est en position $posI$, on ne tentera pas de l'insérer avant le client j qui serait en position $posI$ ou $posI + 1$ (le client i ne changerait pas de place). On ne tentera pas non plus de l'insérer avant le client j qui serait en position $posI - 1$ (cela reviendrait à échanger les positions des clients i et j ce qui sera traité par la suite). Là encore, n'hésitez pas à faire un petit dessin pour vous en convaincre.

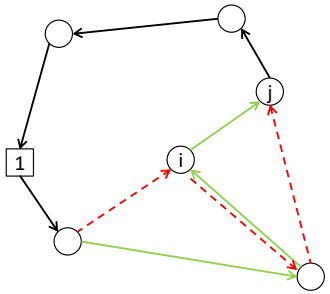
Comme l'opérateur de déplacement intra-tournée consiste en fait en une suppression du client i et une insertion du client i, nous allons calculer le coût engendré par cet opérateur en ajoutant le coût de suppression du client i et le coût d'insertion du client i avant le point j. Un des gros avantages est que nous avons déjà développé, lors du TP précédent, une méthode **deltaCoutInsertion(int position, Client clientToAdd)** qui calcule le coût engendré par l'insertion d'un client.

Question 8. Dans la classe **Tournee**, ajoutez les deux méthodes suivantes avec une visibilité **private** :

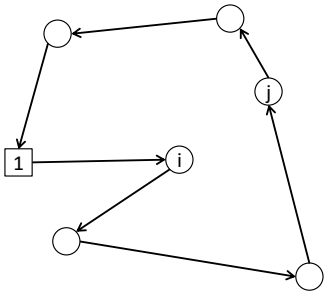
- **getNext(int position)** qui renvoie le point de la tournée qui succède la position **position** : si **position** correspond à la position du dernier client de la tournée, alors il s'agit du dépôt ;
- **isPositionValide(int position)** qui renvoie un booléen indiquant si **position** correspond à une position valide d'un client dans la tournée (donc entre 0 et le nombre de clients -1).



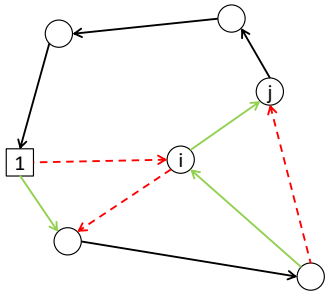
(a) On souhaite déplacer le client i en position 1 avant le client j en position 3



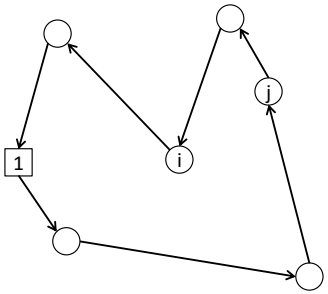
(b) Le client i a été déplacé en position 2, avant le client j



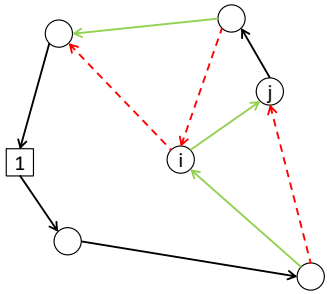
(c) On souhaite déplacer le client i en position 0 avant le client j en position 3



(d) Le client i a été déplacé en position 2, avant le client j



(e) On souhaite déplacer le client i en position 4 avant le client j en position 2



(f) Le client i a été déplacé en position 2, avant le client j

Figure 2 – Modifications du coût d'une tournée lors du déplacement du client i avant le client j. Les coûts à rajouter sont ceux des routes en vert, et les coûts à retirer sont ceux des routes en pointillés rouges.

Notez que ces deux méthodes viennent compléter les méthodes **getPrec(int position)**, **getCurrent(int position)** et **isPositionInsertionValide(int position)** que vous avez développées lors du TP précédent.

Question 9. Dans la classe **Tournee**, ajoutez une méthode **deltaCoutSuppression(int position)** qui renvoie le coût engendré par la suppression du client à la position **position** de la tournée. Cette méthode aura une visibilité **private**.

N'hésitez pas à regarder les dessins de la Figure 2 ou de faire vos propres dessins afin de bien voir comment calculer le coût engendré par la suppression d'un client.
N'oubliez pas de traiter le cas particulier où il n'y a qu'un seul client dans la tournée.

Question 10. Dans la classe **Tournee**, ajoutez une méthode **deltaCoutDeplacement(int positionI, int positionJ)** qui renvoie le coût engendré par le déplacement dans la même tournée du client à la position **positionI** avant le point à la position **positionJ**. Cette méthode renverra l'infini si une des positions passées en paramètre est incorrecte, ou si les deux positions ne sont pas compatibles pour un déplacement.

Pour rappel, un déplacement intra-tournée équivaut à une suppression et une insertion.
Aussi, faites bien attention aux valeurs que peuvent prendre les paramètres : **positionI** correspond à la position d'un client, **positionJ** correspond à la position d'une insertion possible ; et les deux valeurs ne doivent pas être égales ni avoir une différence de 1.
N'hésitez pas à insérer une méthode avec une visibilité **private** qui indique si les positions passées en paramètres sont valides.

Question 11. Dans le paquetage **test**, ajoutez une classe **TestIntraDeplacement** dans laquelle vous ferez un test pour vérifier que le calcul du coût de déplacement d'un client dans une tournée est correct.

Vous pouvez créer une petite instance avec 5 clients qui sont alignés (par exemple toutes les ordonnées sont 0). Ainsi, il est facile pour vous de calculer les distances. Vous insérez ensuite tous les clients dans une tournée mais de sorte que le déplacement d'un ou plusieurs clients puisse améliorer le coût de la tournée. Vous pouvez vous aider du code ci-dessous par exemple.

```
int id = 1;
Depot d = new Depot(id++, 0, 0);
Instance inst = new Instance("test", 100, d);
Client c1 = new Client(10, id++, 5, 0);
Client c2 = new Client(10, id++, 10, 0);
Client c3 = new Client(10, id++, 40, 0);
Client c4 = new Client(10, id++, 5, 0);
Client c5 = new Client(10, id++, 20, 0);
inst.ajouterClient(c1);
inst.ajouterClient(c2);
inst.ajouterClient(c3);
inst.ajouterClient(c4);
inst.ajouterClient(c5);
Tournee t = new Tournee(inst);
t.ajouterClient(c1);
t.ajouterClient(c2);
t.ajouterClient(c3);
t.ajouterClient(c4);
t.ajouterClient(c5);
```

Ensuite, vous affichez le coût de déplacement de quelques clients à une autre position, et vous vérifiez que les coûts sont correctement calculés.

2.1.2 Stockage des informations sur l'opérateur de déplacement intra-tournée

Question 12. Dans le paquetage **operateur**, ajoutez une classe **IntraDeplacement** qui hérite de **OperateurIntraTournee**. 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 des deux positions des clients dans cette tournée. Ajoutez une redéfinition de la méthode **toString**.

En principe, pour les constructeurs, vous avez juste à faire appel aux constructeurs correspondants dans la classe mère.

Question 13. Dans la classe **OperateurLocal**, modifiez les fabriques pour les constructeurs par défaut et pour les constructeurs par données des opérateurs de recherche locale intra-tournée afin de prendre correctement en compte le cas de l'opérateur de déplacement intra-tournée.

Question 14. Modifiez votre test de la classe **TestIntraDeplacement** pour vérifier que :

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

N'hésitez pas à utiliser la fabrique pour construire des objets du type **IntraDeplacement**. Ceci vous permettra de vérifier que les fabriques sont correctement implémentées.

2.1.3 Recherche du meilleur opérateur de déplacement intra-tournée

Question 15. Dans la classe **Tournee**, ajoutez une méthode **getMeilleurOperateurIntra(TypeOperateurLocal type)** qui renvoie le meilleur opérateur intra-tournée de type **type** pour cette tournée.

N'oubliez pas ce que nous avons fait dans la Section 1.3. Cette méthode ne doit pas être spécifique à l'opérateur de déplacement intra-tournée, et il faut donc utiliser les fabriques pour construire les objets du bon type d'opérateur passé en paramètre.
Par ailleurs, faites bien attention aux valeurs prises par les positions pour le déplacement, n'en oubliez pas.

Question 16. Modifiez votre test de la classe **TestIntraDeplacement** pour vérifier que vous arrivez bien à trouver le meilleur opérateur de déplacement intra-tournée dans une tournée.

Question 17. Dans la classe **Solution**, ajoutez une méthode **getMeilleurOperateurIntra(TypeOperateurLocal type)** qui renvoie le meilleur opérateur intra-tournée de type **type** dans la solution. Cette méthode aura une visibilité **private**.

Question 18. Dans la classe **Solution**, ajoutez une méthode **getMeilleurOperateurLocal(TypeOperateurLocal type)** qui renvoie le meilleur opérateur de recherche locale de type **type** dans la solution.

Notez bien que cette méthode aura une visibilité **public** et permettra de décider, en fonction du type passé en paramètre, si l'on fait appel :

- à une méthode de recherche du meilleur opérateur intra-tournée (méthode **getMeilleurOperateurIntra(TypeOperateurLocal type)** que vous venez d'implémenter) ;
- ou à une méthode de recherche du meilleur opérateur inter-tournées (méthode que vous implémenterez par la suite).

Le gros avantage de développer une telle méthode est par la suite de ne pas devoir faire de traitements spécifiques selon que l'on cherche un meilleur opérateur intra-tournée ou inter-tournées dans une solution. Seul le type de l'opérateur passé en paramètre suffit à sélectionner la méthode adéquate dans la classe **Solution**.
Par ailleurs, notez que pour savoir si le type passé en paramètre correspond à un opérateur intra-tournée, vous pouvez, par exemple, vous inspirer du code suivant :

```
1  OperateurLocal.getOperateur(type) instanceof
       OperateurIntraTournee
```

2.1.4 Implémentation du mouvement de déplacement intra-tournée

Question 19. Dans la classe **Tournee**, ajoutez une méthode **doDeplacement(IntraDeplacement infos)** qui implémente le mouvement lié à l'opérateur de déplacement intra-tournée **infos**. Cette méthode renvoie un booléen qui indique si le mouvement a bien été implémenté.

Il faut donc modifier la liste des clients afin de réaliser le mouvement de déplacement d'un client, et mettre correctement à jour l' attribut coût total de la tournée. Pour cela on se sert des informations contenues dans le paramètre **infos**.
Faites extrêmement attention à l'ordre dans lequel vous retirez et vous insérer le client dans la liste des clients, et aux indices auxquels vous faites ces opérations de suppression et d'insertion. N'hésitez pas à faire des dessins pour avoir quelques exemples.
Vous pouvez faire appel à la méthode **check()** de la classe **Tournee** à la fin de la méthode **doDeplacement(IntraDeplacement infos)** afin de vérifier que les attributs ont bien été mis à jour.

Question 20. Dans la classe **IntraDeplacement**, implémentez correctement la méthode **doMouvement()** qui réalise le mouvement lié à l'opérateur de déplacement intra-tournée.

Question 21. Modifiez votre test de la classe **TestIntraDeplacement** 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 22. Dans la classe **Solution**, ajoutez une méthode **doMouvementRechercheLocale(OperateurLocal infos)** qui implémente le mouvement lié à l'opérateur de recherche locale **infos** s'il est réalisable. Cette méthode renvoie un booléen qui indique si le mouvement 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**.
Vous pouvez faire appel à la méthode **check()** de la classe **Solution** à la fin de cette méthode afin de vérifiez que la solution a été correctement mise à jour.
En outre, notez bien ici l'intérêt d'avoir utilisé l'héritage pour les classes des opérateurs de recherche locale. Cette méthode est commune à tous les opérateurs, et vous n'aurez donc pas à la réimplémenter pour les autres opérateurs de recherche locale que vous développerez par la suite.

2.1.5 Test des performances de l'opérateur de déplacement intra-tournée

Nous avons mis en place tous les éléments nécessaires à l'application de l'opérateur de déplacement intra-tournée. Afin de tester les performances de ce seul opérateur, nous allons réaliser un solveur qui fonctionnera selon l'Algorithme 1. Dans un premier temps, on obtient une solution réalisable avec une méthode constructive, puis, on essaye d'améliorer cette solution en implémentant le meilleur opérateur de déplacement intra-tournée. On arrête lorsque le mouvement lié au meilleur opérateur de déplacement intra-tournée ne permet pas d'améliorer la solution.

Algorithme 1 : Algorithme de test de l'opérateur de déplacement intra-tournée.

```
S = solution après application d'une méthode constructive
improve = vrai
while improve est vrai do
    improve = faux
    best = meilleur opérateur de déplacement intra-tournée dans S
    if le mouvement lié à best est améliorant then
        implémenter le mouvement lié à best sur la solution S
        improve = true
    end if
end while
return S
```

Dans un premier temps, nous nous intéressons seulement à évaluer un seul opérateur. Nous verrons dans le TP suivant comment combiner plusieurs opérateurs au sein d'un algorithme de recherche locale.

Dans les premiers tests, nous proposons d'utiliser comme méthode constructive l'algorithme d'insertion simple pour générer une solution initiale. L'avantage est que l'algorithme d'insertion simple fournit une solution de qualité médiocre, ce qui permet de bien se rendre compte des performances de l'opérateur de déplacement intra-tournée, et qu'il soit appliqué de nombreuses fois pour bien le tester. Notez également qu'on applique seulement le meilleur opérateur de déplacement intra-tournée sur l'ensemble de la solution. Or, comme la solution contient plusieurs tournées, et que l'opérateur ne modifie qu'une seule tournée, on pourrait appliquer le meilleur opérateur améliorant sur chaque tournée de la solution. Dans le cadre du TP, nous ne retenons pas cette option par soucis de garder un code assez générique, et de ne pas alourdir le code.

Question 23. Dans le paquetage **solveur**, ajoutez une classe **RechercheLocale** qui implémente l'interface **Solveur**. Ajoutez un attribut qui représente le solveur utilisé pour la méthode constructive qui permet d'obtenir une solution initiale. Ajoutez un constructeur par données. Implémentez les méthodes de l'interface **Solveur**. En particulier, la méthode **solve** devra avoir le comportement de l'Algorithme 1 afin de tester l'opérateur de déplacement intra-tournée.

Testez que votre code fonctionne correctement.

Pour le test, vous pouvez utiliser le solveur d'insertion simple pour la solution initiale. Par ailleurs, 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 24. Afin de bien tester et évaluer les performances de l'opérateur de déplacement intra-tournée, nous allons ajouter le solveur développé à la question précédente 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 **RechercheLocale**.

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

2.2 Opérateur d'échange

Dans cette section, nous allons voir comment implémenter un autre opérateur intra-tournée : l'opérateur d'échange intra-tournée. Nous allons utiliser le même schéma que celui utilisé pour l'opérateur de déplacement intra-tournée. Vous verrez que le développement sera plus rapide car certaines méthodes communes aux opérateurs intra-tournée ont déjà été développées précédemment.

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

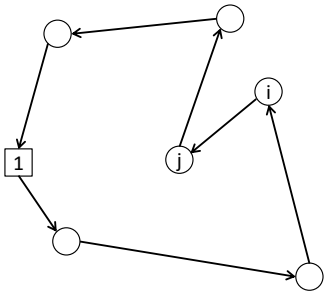
Le voisinage $\mathcal{O}_{intra-ech}(\mathcal{S})$ contient $\sum_{k\in\mathcal{K}} \frac{r_k \cdot r_{k-1}}{2}$ solutions différentes, où r_k est le nombre de clients desservis par la tournée k, et $\mathcal{K}$ est l'ensemble des tournées dans la solution courante $\mathcal{S}$.

Le mouvement lié à l'opérateur d'échange intra-tournée correspond à échanger les positions des clients i et j dans une même tournée. La Figure 3 présente deux exemples d'échange des positions de deux clients i et j dans une tournée. Vous pouvez remarquer que les deux cas sont différents.

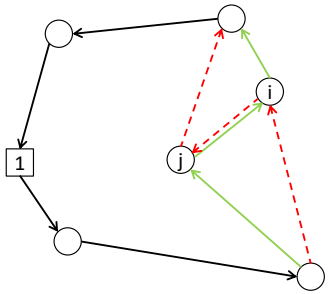
- Dans les Figures 3a et 3b, le client i est le prédécesseur du client j. Dans ce cas, il faut retirer les coûts de la route qui arrive au client i, de la route entre i et j, et de la route qui part du client j ; et ajouter les coûts de la route qui va du prédécesseur de i vers j, de j vers i et de i vers le successeur de j.
- Dans les Figures 3c et 3d, le client i n'est pas le prédécesseur direct du client j. Dans ce cas, il faut remplacer le client i par le client j, et remplacer le client j par le client i. Remplacer un client par autre implique de changer les routes qui arrivent et qui partent du *client à remplacer* par les routes qui arrivent et qui partent du *client remplaçant*.

Notez que l'on peut se limiter au cas où le client i est avant le client j dans la tournée, car échanger i avec j est strictement identique à échanger j avec i. N'hésitez pas à faire un dessin pour vous en convaincre.

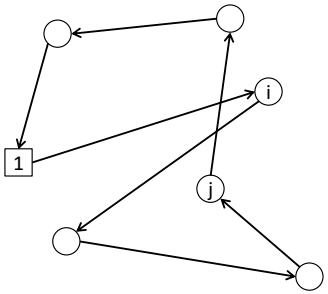
Question 25. Dans la classe **Tournee**, ajoutez une méthode **deltaCoutEchange(int positionI, int positionJ)** qui renvoie le coût engendré



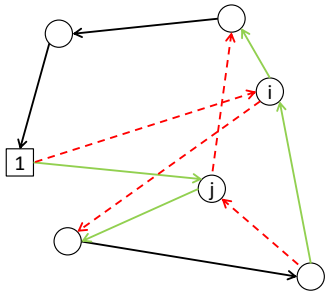
(a) On souhaite échanger le client i en position 2 avec le client j en position 3



(b) Le client i a été échangé avec le client j



(c) On souhaite échanger le client i en position 0 avec le client j en position 3



(d) Le client i a été échangé avec le client j

Figure 3 – Modifications du coût d'une tournée lors de l'échange des clients i et j. Les coûts à rajouter sont ceux des routes en vert, et les coûts à retirer sont ceux des routes en pointillés rouges.

par l'échange dans la même tournée du client à la position **positionI** avant le client à la position **positionJ**. Cette méthode renverra l'infini si une des positions passées en paramètre est incorrecte, ou si les deux positions ne sont pas compatibles pour un échange.

Pour rappel, un échange intra-tournée a un coût différent selon que les deux positions se succèdent ou non. Si les deux positions ne se succèdent pas, alors le coût équivaut au coût de remplacement de i par j plus le coût de remplacement de j par i. Ainsi, vous êtes fortement encouragés à développer des méthodes avec une visibilité **private** :

- **deltaCoutEchangeConsecutif(int positionI)** qui renvoie le coût d'un échange entre les clients aux positions **positionI** et **positionI** +1 ;
- **deltaCoutRemplacement(int position, Client client)** qui renvoie le coût du remplacement du client actuellement à la position **position** par le client **client**.

Aussi, faites bien attention aux valeurs que peuvent prendre les paramètres : **positionI** et **positionJ** correspondent à la position d'un client ; et **positionI** doit être strictement inférieur à **positionJ**.

Question 26. Dans le paquetage **test**, ajoutez une classe **TestIntraEchange** dans laquelle vous ferez un test pour vérifier que le calcul du coût de l'échange de deux clients dans une tournée est correct.

Vous pouvez créer une petite instance avec quelques clients qui sont alignés (par exemple toutes les ordonnées sont 0). Ainsi, il est facile pour vous de calculer les distances. Vous insérez ensuite tous les clients dans une tournée mais de sorte que l'échange d'un ou plusieurs clients puisse améliorer le coût de la tournée.
Ensuite, vous affichez le coût d'échange de quelques paires de clients, et vous vérifiez que les coûts sont correctement calculés.

Question 27. Dans le paquetage **operateur**, ajoutez une classe **IntraEchange** qui hérite de **OperateurIntraTournee**. 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 des deux positions des clients dans cette tournée. Ajoutez une redéfinition de la méthode **toString**.

Question 28. Dans la classe **OperateurLocal**, modifiez les fabriques pour les constructeurs par défaut et pour les constructeurs par données des opérateurs de recherche locale intra-tournée afin de prendre correctement en compte le cas de l'opérateur d'échange intra-tournée.

Question 29. Modifiez votre test de la classe **TestIntraEchange** pour vérifier que :

- vous arrivez bien à créer des objets de type **IntraEchange** ;

- les valeurs des attributs de ces objets sont correctes (en particulier le coût) ;
- les méthodes **isMeilleur(Operateur op)**, **isMouvementRealisable()** et **isMouvementAmeliorant()** renvoient des résultats corrects.

N'hésitez pas à utiliser la fabrique pour construire des objets du type **IntraEchange**. Ceci vous permettra de vérifier que les fabriques sont correctement implémentées.

Question 30. Modifiez votre test de la classe **TestIntraEchange** pour vérifier que vous arrivez bien à trouver le meilleur opérateur d'échange intra-tournée dans une tournée.

Remarquez bien ici l'avantage d'avoir réalisé le travail dans la Section 1 : nous n'avons pas besoin de redéfinir de méthodes pour trouver le meilleur opérateur d'échange intra-tournée dans une tournée ou une solution. Tout le travail a été réalisé précédemment.

Question 31. Dans la classe **Tournee**, ajoutez une méthode **doEchange(IntraEchange infos)** qui implémente le mouvement lié à l'opérateur d'échange intra-tournée **infos**. Cette méthode renvoie un booléen qui indique si le mouvement a bien été implémenté.

Il faut donc modifier la liste des clients afin de réaliser le mouvement d'échange des clients, et mettre correctement à jour l'attribut coût total de la tournée. Pour cela on se sert des informations contenues dans le paramètre **infos**.
Pour rappel, un échange correspond à remplacer les clients situés à deux positions différentes de la liste.
Vous pouvez faire appel à la méthode **check()** de la classe **Tournee** à la fin de la méthode **doDeplacement(IntraDeplacement infos)** afin de vérifier que les attributs ont bien été mis à jour.

Question 32. Dans la classe **IntraEchange**, implémentez correctement la méthode **doMouvement()** qui réalise le mouvement lié à l'opérateur d'échange intra-tournée.

Question 33. Modifiez votre test de la classe **TestIntraEchange** 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).

Remarquez ici que nous n'avons pas besoin de ré-implémenter une méthode dans la classe **Solution** pour implémenter un mouvement de recherche locale. La méthode **doMouvementRechercheLocale(OperateurLocal infos)** peut resservir pour le cas de l'opérateur d'échange intra-tournée.

Question 34. Dans la classe **RechercheLocale**, modifiez la méthode **solve(Instance instance)** afin de tester l'opérateur d'échange intra-tournée (au lieu de l'opérateur de déplacement intra-tournée). Le principe de l'Algorithme 1 reste le même, seul l'opérateur à tester change.

Testez que votre code fonctionne correctement.

Question 35. Afin de bien tester et évaluer les performances de l'opérateur d'échange intra-tournée, nous allons utiliser à nouveau la classe **TestAllSolveur**, toujours avec le solveur **RechercheLocale** que vous venez de modifier.

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