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

TP 4 – Recherche locale - Partie 2

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, nous avons commencé à mettre en place un algorithme de recherche locale afin d'améliorer les solutions obtenues par des méthodes constructives. En particulier, nous avions mis en place l'architecture pour stocker les informations sur les opérateurs de recherche locale, et nous avons implémenté deux opérateurs intra-tournée.

L'objectif de ce TP est d'aboutir à un algorithme de recherche locale performant. Dans un premier temps, nous allons ajouter deux opérateurs inter-tournées : le déplacement et l'échange. Puis, enfin, nous implémenterons un algorithme de recherche locale qui utilise tous les opérateurs que nous aurons développés.

Ce TP permet d'illustrer les notions suivantes :

- les opérateurs d'amélioration *inter-tournées* ;
- la notion de recherche locale.

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 inter-tournées

Vous avez peut-être remarqué que les opérateurs vus dans le TP précédent peuvent être étendus au cas où l'on considère deux tournées différentes au lieu de considérer un mouvement dans une seule tournée. Notez bien, cependant, que dan le cas des opérateurs inter-tournées, nous devrons alors faire bien attention à ce que la capacité des tournées soit toujours bien respectée lorsque des clients changent de tournées.

Dans cette section nous allons nous concentrer sur l'optimisation d'un couple de tournées. Pour cela, nous allons implémenter des opérateurs de recherche locale inter-tournées :

- l'opérateur de déplacement qui déplace un client depuis une tournée vers une autre tournée ;
- l'opérateur d'échange qui échange les positions de deux clients situés dans des tournées différentes.

En ce qui concerne l'implémentation, nous allons suivre, pour chaque opérateur, le même schéma que nous avons appliqué jusqu'ici :

- évaluation du coût engendré par l'application de l'opérateur ;
- stockage des informations sur l'opérateur dans une classe héritant de **OperateurInterTournees** ;
- 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.

1.1 Opérateur de déplacement

L'opérateur de déplacement inter-tournées, indiqué par $\mathcal{O}_{inter-dep}$ considère chaque couple (t, u) de tournées de la solution courante $\mathcal{S}$ et évalue le déplacement de chaque client de la tournée t vers chaque position de la tournée u, à condition que l'ajout du client dans la tournée u n'entraine pas de dépassement de la capacité Q du véhicule. S'il existe un déplacement tel que la solution $\mathcal{S}$ est améliorée, alors ce déplacement est implémenté.

Le voisinage $\mathcal{O}_{inter-dep}(\mathcal{S})$ contient $\sum_{(k,k')\in\mathcal{K},k\neq k'} r_k \cdot (r_{k'} + 1)$ 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 $\mathcal{S}$.

Le mouvement lié à l'opérateur de déplacement inter-tournées correspond à déplacer le client i d'une tournée t avant le point j d'une autre tournée u. Le point j peut être un client ou le dépôt. La Figure 1 présente un exemple de déplacement d'un client i d'une tournée t avant un client j d'une tournée u. Vous pouvez remarquer que le déplacement inter-tournées du client i avant le point j correspond en fait à deux mouvements simultanés :

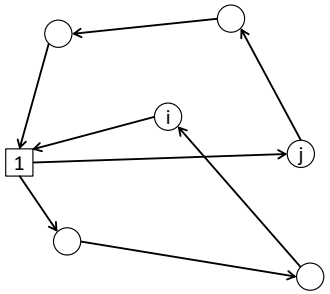
- la suppression du client i de sa position actuelle dans la tournée t (voir Figure 1c) : on retire les routes qui arrivaient et partaient de i et on rajoute la route qui va du prédécesseur de i au successeur de i ;
- l'insertion de i avant le point j dans la tournée u (voir Figure 1d) : on retire la 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.

Ainsi, pour évaluer le coût du mouvement, il faut sommer le coût de la suppression du client i dans la tournée t et le coût de l'insertion du client i avant le point j dans la tournée u. Notez que vous avez dû développer des méthodes qui calculent le coût de la suppression d'un client dans une tournée, et de l'insertion d'un client dans une tournée.

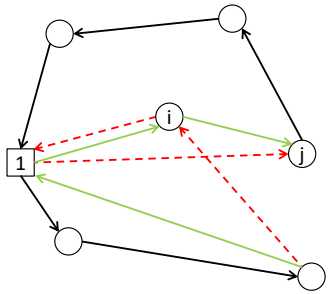
La Figure 1 ne présente pas le cas où le point j dans la tournée u est le dépôt. N'hésitez pas à faire votre propre dessin, si nécessaire, afin de bien visualiser ce qui se passe dans ce cas.

Question 1. Dan la classe **Tournee**, vérifiez que vous avez bien implémenté les deux méthodes suivantes, avec une visibilité **public**.

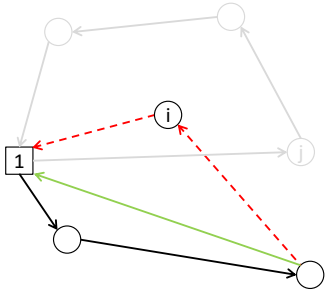
- **deltaCoutSuppression(int position)** qui renvoie le coût engendré par la suppression du client à la position **position** de la tournée. Cette méthode renvoie l'infini si la position passée en paramètre est incorrecte.
- **deltaCoutInsertion(int position, Client clientToAdd)** qui renvoie le coût engendré par l'insertion du client **clientToAdd** à la position **position** de la tournée. Cette méthode renvoie l'infini si la position passée en paramètre est incorrecte.



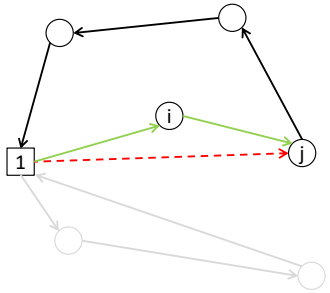
(a) On souhaite déplacer le client i en position 2 de la tournée t avant le client j en position 0 de la tournée u



(b) Le client i a été déplacé en position 0 dans la tournée u avant le client j



(c) Le client i a été supprimé de la tournée t



(d) Le client i a été ajouté avant le client j en position 0 de la tournée u

Figure 1: Modifications du coût de deux tournées t et u lors du déplacement du client i de la tournée t avant le client j de la tournée u. Les coûts à rajouter sont ceux des routes en vert, et les coûts à retirer sont ceux des routes en pointillés rouges.

Si vous avez bien suivi les sujets des TP précédents, les deux seules choses que vous avez à faire sont :

- modifier la visibilité de la méthode **deltaCoutSuppression(int position)**, qui doit maintenant avoir une visibilité **public** ;
- modifier si besoin la méthode **deltaCoutSuppression(int position)** pour qu'elle renvoie bien une valeur infinie si la position passée en paramètre en incorrecte. En effet, maintenant que la visibilité est **public**, il y a plus de risques que cette méthode soit appelée avec de mauvaises valeurs de paramètres.

Vous avez dû remarquer que l'on possède déjà une méthode **deltaCoutInsertion(int position, Client clientToAdd)** dans la classe **Tournee**, mais comme elle a été développée pour les opérateurs intra-tourné, il n'y a pas de vérification de la validité de la capacité. Deux solutions peuvent être mises en place :

- ajouter un booléen dans la liste des paramètres qui indique s'il faut vérifier la validité de la capacité ;
- ajouter une méthode **deltaCoutInsertionInter(int position, Client clientToAdd)** qui fera la vérification de la capacité (est-ce que le client peut être ajouté ?), et si le client peut être ajouté, appellera ensuite la méthode **deltaCoutInsertion(int position, Client clientToAdd)** .

Nous proposons ici d'implémenter la seconde méthode afin de modifier le moins possible le code existant.

Question 2. Dans la classe **Tournee**, ajoutez une méthode **deltaCoutInsertionInter(int position, Client clientToAdd)** qui calculera le coût d'insertion du client **clientToAdd** si celui-ci peut être ajouté dans la tournée. Si le client ne peut pas être ajouté dans la tournée, à cause de la capacité, alors la méthode renverra l'infini.

Réutilisez le code déjà développé. Si vous avez déjà une méthode pour savoir si un client peut être ajouté dans une tournée, c'est le moment de la réutiliser.

Question 3. Dans le paquetage **operateur**, ajoutez une classe **InterDeplacement** qui hérite de **OperateurInterTournees**. Implémentez les méthodes **evalDeltaCoutTournee()** et **evalDeltaCoutAutreTournee()** qui renvoient le surcoût engendré par l'application de l'opérateur dans chacune des tournées. Pour le moment, pour implémenter la méthode **doMouvement()** un simple **return false;** suffira.

Ajoutez un constructeur par défaut, un constructeur par la donnée des deux tournées, et des deux positions des clients dans ces tournées. Ajoutez une redéfinition de la méthode **toString**.

Pour les évaluations des coûts, rappelez-vous bien que dans la première tournée, il s'agit d'une suppression, et dans l'autre tournée, il s'agit d'une insertion du client **clientI**.
En principe, pour les constructeurs, vous avez juste à faire appel aux constructeurs correspondants dans la classe mère.

Question 4. Dans la classe **OperateurLocal**, modifiez les fabriques pour les constructeurs par défaut et pour les constructeurs par données des opérateurs de recherche locale inter-tournées afin de prendre correctement en compte le cas de l'opérateur de déplacement inter-tournées.

Question 5. Dans le paquetage **test**, ajoutez une classe **TestInterDeplacement** dans laquelle vous ferez un test pour vérifier que :

- vous arrivez bien à créer des objets de type **InterDeplacement** ;
- les valeurs des attributs de ces objets sont correctes (en particulier le coût, et notamment dans le cas où le déplacement engendre une violation de la capacité) ;
- les méthodes **isMeilleur(Operateur op)**, **isMouvementRealisable()** et **isMouvementAmeliorant()** renvoient des résultats corrects.

Vous pouvez créer une petite instance avec quelques clients qui sont alignés (par exemple toutes les ordonnées sont 0), et d'autres clients qui sont également alignés sur une autre ligne (par exemple toutes les ordonnées sont 10). Ainsi, il est assez facile pour vous de calculer les distances. Vous pouvez également faire en sorte qu'au moins un client ne puisse pas être déplacé à cause de la capacité de la tournée dans laquelle on souhaite le déplacer. Vous insérez ensuite tous les clients dans 2 tournées de telle sorte que le déplacement d'un client dans une autre tournée puisse améliorer la solution. 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++, 0, 5);
Client c2 = new Client(10, id++, 0, 10);
Client c3 = new Client(10, id++, 0, 10);
Client c4 = new Client(10, id++, 0, 15);
Client c5 = new Client(60, id++, 10, 0);
Client c6 = new Client(10, id++, 10, 10);
Client c7 = new Client(10, id++, 10, 15);
inst.ajouterClient(c1);
inst.ajouterClient(c2);
inst.ajouterClient(c3);
inst.ajouterClient(c4);
inst.ajouterClient(c5);
inst.ajouterClient(c6);
inst.ajouterClient(c7);
Tournee t = new Tournee(inst);
t.ajouterClient(c1);
t.ajouterClient(c2);
t.ajouterClient(c6);
t.ajouterClient(c3);
t.ajouterClient(c4);
Tournee u = new Tournee(inst);
u.ajouterClient(c5);
u.ajouterClient(c7);
```

Question 6. Dans la classe **Tournee**, ajoutez une méthode **getMeilleurOperateurInter(Tournee autreTournee, TypeOperateurLocal type)** qui renvoie le meilleur opérateur inter-tournées de type **type** pour cette tournée et la tournée **autreTournee**.

Cette méthode ne doit pas être spécifique à l'opérateur de déplacement inter-tournées, et il faut donc utiliser les fabriques pour construire les objets du bon type d'opérateur passé en paramètre, comme nous avons pu le faire dans le cas intra-tournée.
Par ailleurs, faites bien attention aux valeurs prises par les positions : les valeurs pour le client supprimé, ainsi que les valeurs pour l'insertion.
Enfin, pensez-bien à traiter le cas où la tournée passée en paramètre **autreTournee** serait la tournée courante **this**.

Question 7. Modifiez votre test de la classe **TestInterDeplacement** pour vérifier que vous arrivez bien à trouver le meilleur opérateur de déplacement inter-tournées.

Question 8. Dans la classe **Solution**, ajoutez une méthode avec une visibilité **private**, **getMeilleurOperateurInter(TypeOperateurLocal type)** qui renvoie le meilleur opérateur inter-tournées de type **type** dans la solution.

Question 9. Dans la classe **Solution**, modifiez la méthode **getMeilleurOperateurLocal(TypeOperateurLocal type)** qui renvoie le meilleur opérateur de recherche locale de type **type** dans la solution. Il faut à présent correctement en compte le cas où le type passé en paramètre correspond à un opérateur inter-tournées.

Question 10. Dans la classe **Tournee**, ajoutez une méthode **doDeplacement(InterDeplacement infos)** qui implémente le mouvement lié à l'opérateur de déplacement inter-tournées **infos**. Cette méthode renvoie un booléen qui indique si le mouvement a bien été implémenté.

Il faut donc modifier les listes des clients de la tournée courante **this** ainsi que de l'autre tournée (à laquelle on doit pouvoir accéder depuis le paramètre **infos**).
Par ailleurs, il faut modifier en conséquence les attributs coût total et demande totale des deux tournées.
Vous pouvez faire appel à la méthode **check()** de la classe **Tournee** à la fin de la méthode **doDeplacement(InterDeplacement infos)** afin de vérifier que les attributs ont bien été mis à jour dans les deux tournées.

Question 11. Dans la classe **InterDeplacement**, implémentez correctement la méthode **doMouvement()** qui réalise le mouvement lié à l'opérateur de déplacement inter-tournées.

Question 12. Modifiez votre test de la classe **TestInterDeplacement** 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 13. Dans la classe **RechercheLocale**, modifiez la méthode **solve(Instance instance)** afin de tester l'opérateur de déplacement inter-tournées (au lieu des opérateurs intra-tournée). Le principe de l'algorithme mis en place lors du TP précédent reste le même, seul l'opérateur à tester change.

Testez que votre code fonctionne correctement.

Question 14. Afin de bien tester et évaluer les performances de l'opérateur de déplacement inter-tournées, 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.

1.2 Opérateur d'échange inter-tournées

L'opérateur d'échange inter-tournées, indiqué par $\mathcal{O}_{inter-ech}$ considère chaque couple (t, u) de tournées de la solution courante $\mathcal{S}$ et évalue l'échange de chaque client de la tournée t avec chaque client de la tournée u, à condition que cet échange n'entraine pas de dépassement de la capacité Q du véhicule, dans aucune des tournées t et u. S'il existe un échange tel que la solution $\mathcal{S}$ est améliorée, alors cet échange est implémenté.

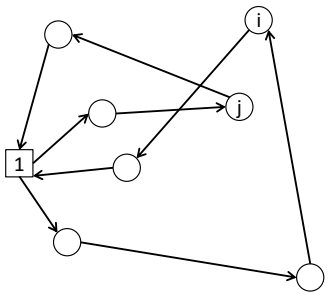
Le voisinage $\mathcal{O}_{inter-ech}(\mathcal{S})$ contient $\sum_{(k,k')\in\mathcal{K},k\neq k'} r_k \cdot r_{k'}$ solutions différentes, où r_k est le nombre de clients desservis par la tournée k, et $\mathcal{K}$ est l'ensemble des tournées dans la solution courante $\mathcal{S}$.

Le mouvement lié à l'opérateur d'échange inter-tournées correspond à échanger le client i d'une tournée t avec le client j d'une autre tournée u. La Figure 2 présente un exemple d'échange d'un client i d'une tournée t avec un client j d'une tournée u. Vous pouvez remarquer que l'échange inter-tournées du client i avec le client j correspond en fait à deux mouvements simultanés :

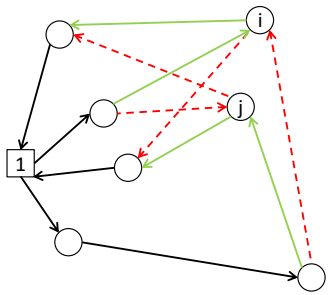
- dans la tournée t, le remplacement du client i par le client j (voir Figure 2c) ;
- dans la tournée u, le remplacement du client j par le client i (voir Figure 2d).

Ainsi, pour évaluer le coût du mouvement, il faut sommer le coût du remplacement du client i par le client j dans la tournée t et le coût du remplacement du client j par client i dans la tournée u. Notez que vous avez dû développer une méthode qui calcule le coût du remplacement d'un client par un autre dans une tournée.

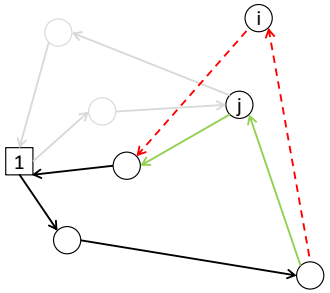
N'oubliez pas que dans le cas des opérateurs inter-tournées, il faut vérifier, pour chaque remplacement de clients, si la capacité du véhicule est toujours satisfaite.



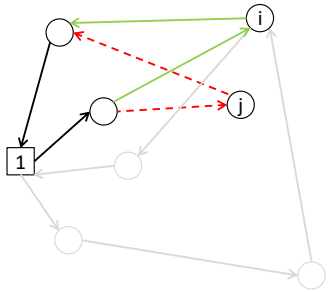
(a) On souhaite échanger le client i en position 2 de la tournée t avant le client j en position 1 de la tournée u



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



(c) Le client i a été remplacé dans la tournée t par le client j



(d) Le client j a été remplacé dans la tournée u par le client i

Figure 2: Modifications du coût de deux tournées t et u lors de l'échange du client i de la tournée t avec le client j de la tournée u. Les coûts à rajouter sont ceux des routes en vert, et les coûts à retirer sont ceux des routes en pointillés rouges.

Question 15. Ajoutez dans votre code tous les éléments nécessaires au développement et au test de l'opérateur d'échange inter-tournées.

Comme il s'agit du quatrième opérateur de recherche locale que nous implémentons, vous devez maintenant pouvoir réaliser cela sans aide supplémentaire.
Rappelez-vous bien des éléments suivants :

- ne pas faire de code redondant ;
- faire des tests ;
- faire des dessins ou des exemples numériques.

2 Recherche locale

À présent que nous développé 4 opérateurs de recherche locale, et que nous avons vérifié l'efficacité de chacun des opérateurs, nous allons développer un algorithme de recherche locale qui combine ces différents opérateurs.

Considérons un problème d'optimisation dont la fonction objectif est à minimiser (l'objectif est de trouver la solution qui minimise le coût). Étant donnée une solution $\mathcal{S}$, on notera $c_{\mathcal{S}}$ son coût.

L'idée de la recherche locale est d'améliorer une solution donnée $\mathcal{S}$ en construisant une suite de solutions $(\mathcal{S} = \mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_n)$ telle que la solution $\mathcal{S}_{i+1}$ est une solution voisine de la solution $\mathcal{S}_i$ et que $c_{\mathcal{S}_{i+1}} < c_{\mathcal{S}_i}$. La solution $\mathcal{S}_n$ finale de la suite est dite *minimum local*. Un minimum local est une solution $\mathcal{S}^*$ telle qu'il n'existe pas de solution voisine $\bar{\mathcal{S}}$ telle que $c_{\bar{\mathcal{S}}} < c_{\mathcal{S}^*}$. La recherche locale s'arrête alors.

Plusieurs manières d'implémenter une recherche locale existent. Ici, nous proposons d'enchainer la visite des voisinages associés aux opérateurs codés précédemment. Notre recherche locale termine quand l'exploration de la suite de tous ces voisinages ne produit pas une meilleure solution par rapport à la solution courante. Le pseudo code est donné dans l'Algorithme **??**.

Algorithme 1 : Algorithme de recherche locale.

```
S = solution initiale (e.g. après application d'une méthode constructive)
OPER = {O_intra-dep, O_intra-ech, O_inter-dep, O_inter-ech}
improve = vrai
while improve est vrai do
    improve = faux
    for all O ∈ OPER do
        best = meilleur opérateur de type O 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 for
end while
return S
```

Question 16. Dans la classe **RechercheLocale**, modifiez la méthode **solve(Instance instance)** afin d'implémenter un algorithme de recherche locale comme décrit dans l'Algorithme 1.

Testez que votre code fonctionne correctement.

Pour la solution initiale, vous pouvez utiliser dans un premier temps l'insertion simple pour tester les performances de l'algorithme de recherche locale. Dans un second temps, vous pouvez utiliser votre meilleur méthode constructive afin d'essayer d'obtenir les meilleurs résultats.

Question 17. Afin de bien tester et évaluer les performances de la recherche locale, 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 et avec votre meilleure méthode constructive.