Compte-rendu TP3

Aide à la décision – Tournée de véhicules

CHEVALIER Pierre – PIERREVAL Adrien

17/01/2016

COMPTE-RENDU TP3

Aide à la décision – Tournée de véhicules

SOMMAIRE

INTRODUCTION

Les problèmes de Tournées de véhicules sont des problèmes traitant du passage d'un ou plusieurs véhicules parmi différents points de livraison. Chaque livraison devait respecter des contraintes d'ouverture, de fermeture, et de capacité des véhicules. L'objectif est donc ici de trouver la ou les tournées les plus optimisées en terme de distance (qu'on assimilera comme étant la même notion que le temps ici), et de nombre de véhicules (aussi appelé nombre de tournées).

Le principe de ce TP3 d'Aide à la Décision consistait donc à mettre en œuvre plusieurs algorithmes de recherche locale permettant d'optimiser au mieux nos tournées de véhicules. Nous avons plus précisément cherché à réduire la distance totale parcourue plutôt qu'à réduire le nombre de véhicules. Notre programme devait s'appuyer sur des fichiers de données fournis, représentant des situations de tournées de véhicules à effectuer, et afficher à l'issue de ses calculs les itinéraires de chaque route, ainsi que le nombre de route et la distance totale parcourue, en tenant compte de la charge maximale du véhicule ainsi que des horaires d'ouvertures et de fermeture de chaque point de livraison.

L'exécution de ce programme se découpe donc en deux temps : Le premier consiste à créer une solution à l'aide d'heuristiques. Pour ce TP nous avions le choix entre deux heuristiques, l'heuristique d'insertion et l'heuristique de fusion. Nous avons choisi d'utiliser ici l'heuristique d'insertion. Puis, dans un second temps, il fallait lancer les algorithmes de recherche locale sur la solution créée par l'heuristique. Nous avons pour cela implémenté 4 algorithmes : 2-opt-* cas particulier, ot-opt cas particulier, ot-opt cas général, et 2-opt-* cas particulier. A la suite de ces algorithmes de recherche locale, nous devions obtenir des solutions optimisées.

1. LA STRUCTURE DE DONNEES

Une structure de données nous avait été fournie au démarrage du TP, permettant de lire les fichiers de données et d'initialiser les structures de données adéquates à notre travail. Nous avons ensuite complété cette structure avec nos propres classes afin de résoudre les différents algorithmes demandés.

1.1. Classe Customer

La classe `Customer` contient toutes les informations à propos d'un point de livraison, ou client. Elle est initialisée par la classe data, et chaque `customer` est défini par son id. Chaque instance de `customer` est initialisée grâce aux données contenues dans les fichiers de données, et ses informations restent inchangées tout le long du programme.

1.2. Classe Arc

La classe `Arc` contient toutes les informations pour les arcs. Elle relie en effet deux `Customers` et contient des informations en rapport avec cette liaison telle que la distance entre les deux points, l'id du client représentant l'origine de l'arc et l'id du client représentant la destination. On notera la présence de l'attribut `saving`, qui nous servira pour l'heuristique de fusion .

1.3. Classe Data

La classe Data contient toutes les informations obtenues du fichier de données. Elle permet ainsi le stockage permanent des données, mais son but n'est pas d'être modifiée. Son contenu principal sera donc un vecteur de Customer, un vecteur d'Arcs ainsi que toutes les constantes de contraintes de notre cas de tournées de véhicules (capacité des véhicules, positionnement du dépôt, etc…).

En général, la première instance de Data est recopié dans chaque `WorkingSolution` qui l'utilise ensuite pour avoir accès aux données décrites précédemment.

1.4. Classe WorkingSolution

La classe `WorkingSolution` organise les informations de la classe `data` de manière à mieux pouvoir les utiliser pour les algorithmes à exécuter. La classe `WorkingSolution` implémente donc des contextes de listes chaînés pour les routes et les clients, de manière à avoir une meilleure organisation des routes. De plus, `WorkingSolution` implémente des méthodes afin d'agir avec ces structures de données, tant à la fois pour y accéder que pour les modifier.

Le principal intérêt de cette classe est donc de contenir une liste chaînée de routes, elles-mêmes étant des listes chaînées de nœuds. De ce fait, chaque client contient des informations sur la charge cumulée du véhicule à son arrivée, son temps d'arrivée, le client livré ensuite et celui livré avant, la tournée à laquelle il appartient, etc. De la même façon, chaque tournée contient des références au premier client desservi (en l'occurrence le dépôt), au temps total mis par la tournée, etc.

`WorkingSolution` est donc en résumé le contenu des résultats des heuristiques permettant de composer une solution et des algorithmes de recherche locale.

1.5. Classe heuristique_insertion

`Heuristique_insertion` est une classe qui dérive de `WorkingSolution`. Celle-ci permet uniquement de créer une nouvelle solution à l'aide de l'heuristique d'insertion en profitant des attributs de la classe `WorkingSolution`. Cet héritage nous permet notamment de réutiliser la classe ensuite pour y appliquer les algorithmes de recherche locale.

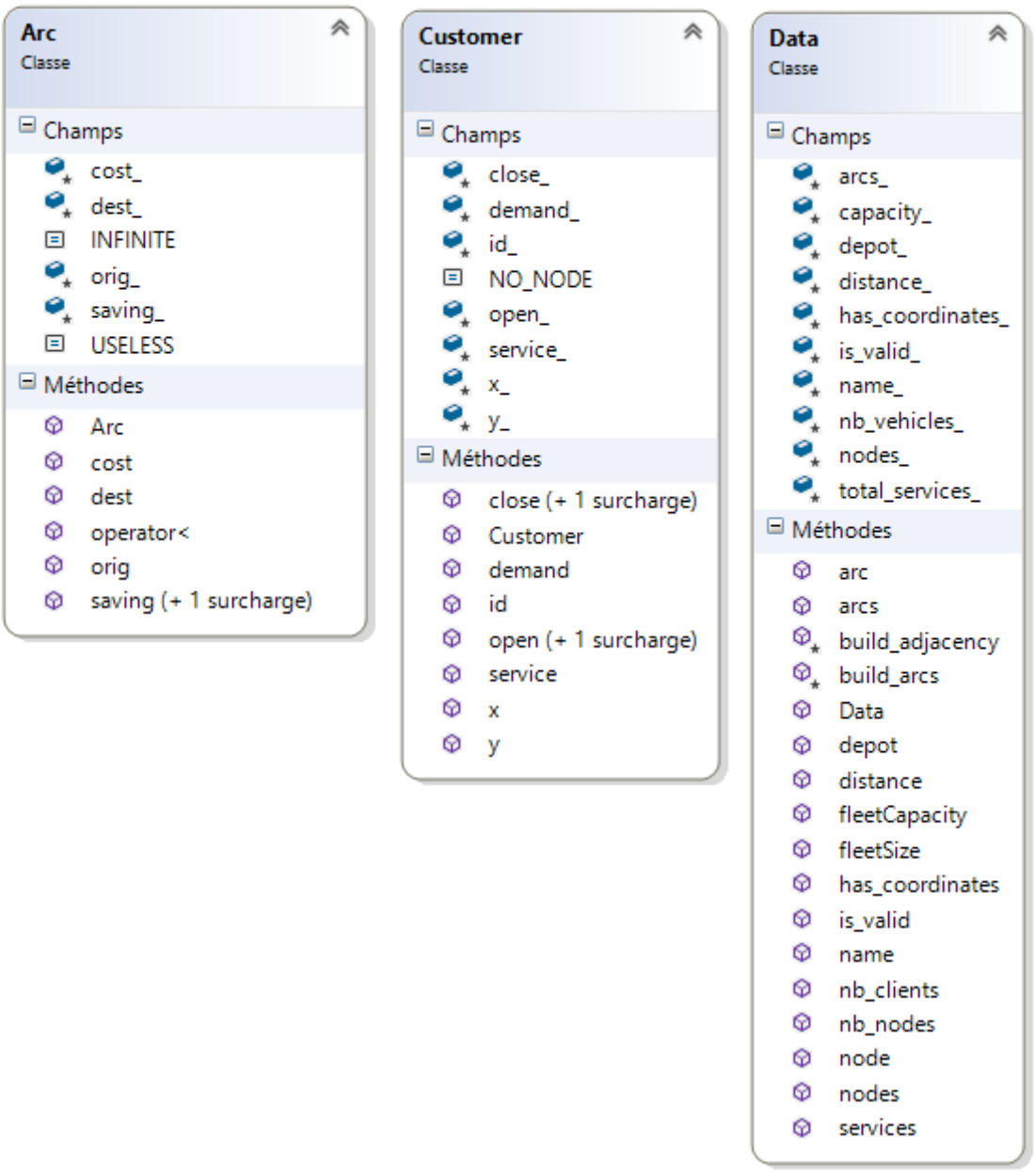
1.6. Classe heuristique_fusion

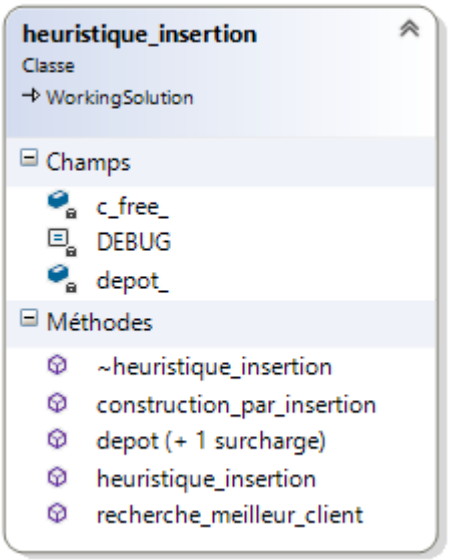
La classe `heuristique_fusion` est similaire à la classe `heuristique_insertion`, seule l'heuristique utilisée change pour créer la solution, ici on utilise l'heuristique de fusion. On notera que cette classe n'est pas terminée, et que des disfonctionnement sont présents.

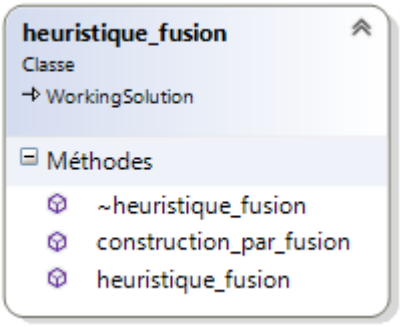
1.7. Classe recherche_locale

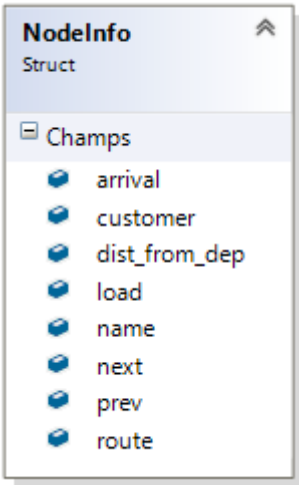
La classe `recherche_locale` contient les méthodes associées aux algorithmes de recherche locale. Elle a en attribut une instance de `WorkingSolution` sur laquelle elle exécute les algorithmes qu'elle contient.

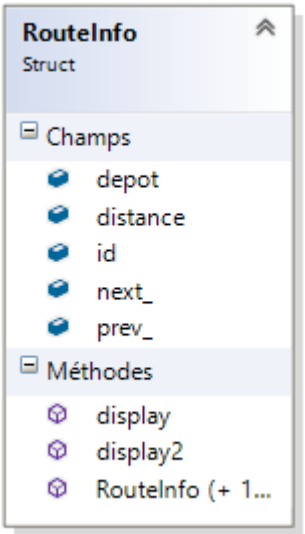
1.8. Résumé : diagrammes UML

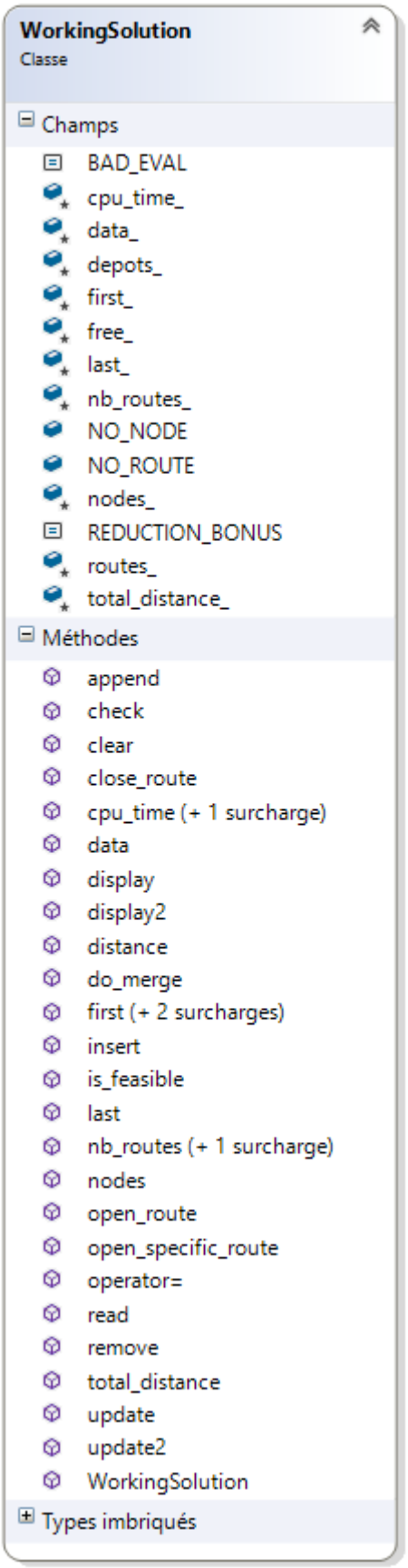



heuristique_insertion
Classe
WorkingSolution
Champs
c_free_
DEBUG
depot_
Méthodes
~heuristique_insertion
construction_par_insertion
depot (+ 1 surcharge)
heuristique_insertion
recherche_meilleur_client


heuristique_fusion
Classe
WorkingSolution
Méthodes
~heuristique_fusion
construction_par_fusion
heuristique_fusion


NodeInfo
Struct
Champs
arrival
customer
dist_from_dep
load
name
next
prev
route


RouteInfo
Struct
Champs
depot
distance
id
next_
prev_
Méthodes
display
display2
RouteInfo (+ 1...


WorkingSolution
Classe
Champs
BAD_EVAL
cpu_time_
data_
depots_
first_
free_
last_
nb_routes_
NO_NODE
NO_ROUTE
nodes_
REDUCTION_BONUS
routes_
total_distance_
Méthodes
append
check
clear
close_route
cpu_time (+ 1 surcharge)
data
display
display2
distance
do_merge
first (+ 2 surcharges)
insert
is_feasible
last
nb_routes (+ 1 surcharge)
nodes
open_route
open_specific_route
operator=
read
remove
total_distance
update
update2
WorkingSolution
Types imbriqués

2. IMPLEMENTATION

2.1. Heuristique d'insertion

L'heuristique que nous avons utilisé pour créer notre solution est l'heuristique d'insertion. Nous avons donc utilisé l'algorithme qui consiste à parcourir chaque client et à les insérer, si possible, dans une tournée vide jusqu'à que celle-ci ne puisse plus accepter de nouveaux clients ou qu'il n'y ait plus de clients éligibles à insérer.

Pour optimiser nos résultats nous avons dans un premier temps trié nos clients suivant leur distance du dépôt et ainsi nos premières insertions dans les nouvelles tournées sont optimales puis nous trions également à chaque itération les clients éligibles en fonction de leur distance avec le dernier point inséré pour que le client le plus proche soit celui inséré dans la tournée.

Ainsi nous avons obtenue une heuristique d'insertion efficace qui nous permet d'obtenir des résultats satisfaisant avant même la recherche locale.

2.2. Recherche Locale : 2-opt-*

Le principe de cet algorithme de recherche locale est de voir si l'on peut échanger deux pointeur `next` de deux points et ainsi obtenir un meilleur résultat en échangeant les "queues" de tournées. Pour ce faire il faut tester toutes les solutions possibles, vérifiées qu'elles soient réalisables (contrainte de temps et de charge) et quelles apportent un gain. Si oui on garde les modifications, si non on retourne à la solution de départ.

Dans le cas particulier du 2-opt-*, on cherche à concaténer les routes qui peuvent l'être. Pour ce faire, on va prendre chaque route, et voir si elles peuvent se concaténer. Auparavant, on a vérifié les conditions suivantes : que les routes ne soient pas identiques, que les fenêtres de temps, et que les contraintes de charges soient respectées. Après vérification que la solution optimisée est toujours valable, on remplace l'ancienne solution par la nouvelle.

2.3. Recherche locale : ot-opt

Le but de cette algorithme est d'insérer un point à la suite d'un autre. On parcours donc tous les points et l'on regarde s'il est possible de l'ajouter après un autre point et si on obtient un gain positif avec cet ajout. Dans ce cas on exécute l'algorithme : on enlève le point à insérer de sa route puis on l'insère à son nouvel emplacement.

Les cas particuliers sont :

- l'insertion de point unique sur la tournée
- l'insertion sur une même tournée (non implémenté de manière individuelle)
- swap entre voisin (géré dans cross)

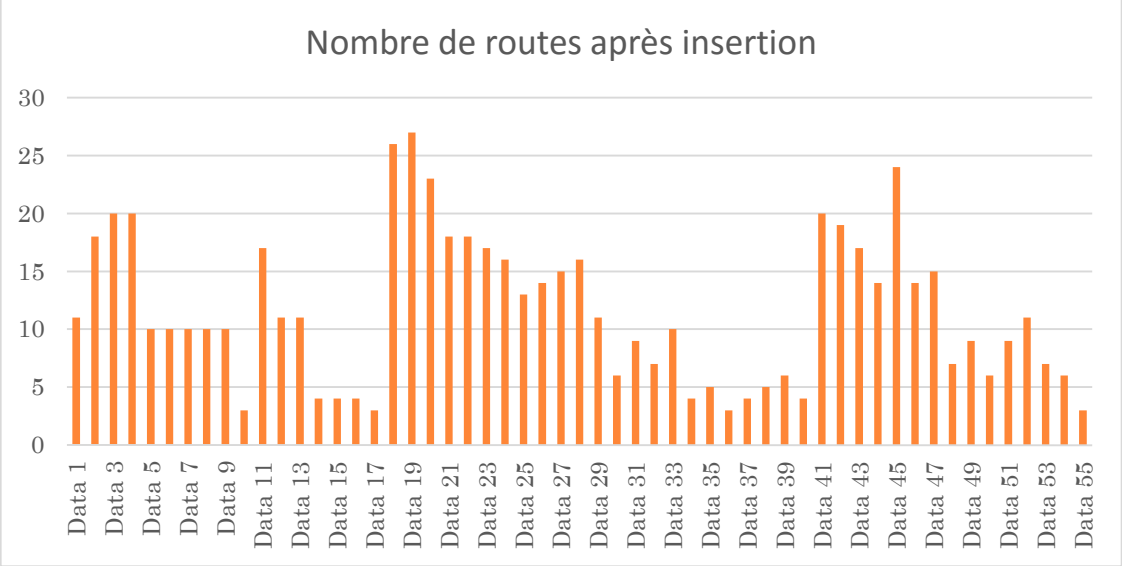
2.4. Recherche locale : cross

La fonction cross consiste à échanger deux points de place (faire un swap). On parcours donc chaque client et on tente de l'échanger avec un autre client. Si l'on obtient un gain alors on garde l'échange, si l'on en obtient pas alors on fait l'échange inverse.

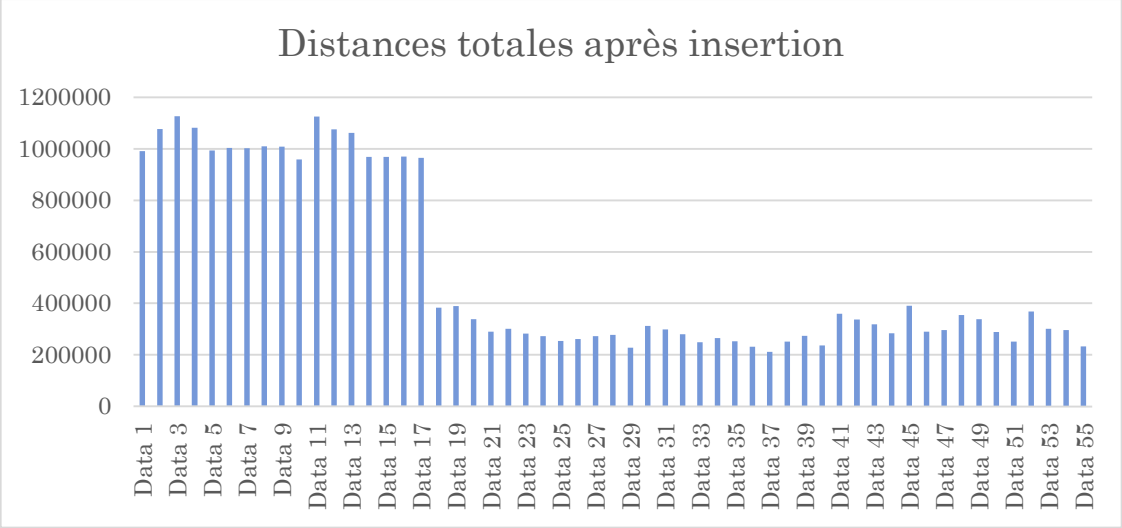
3. RESULTATS

3.1. Nombre de routes et distance totale

Pour tester notre programme, nous avons implémenter une fonction dans le main qui permet de lancer l'heuristique d'insertion ainsi que la recherche locale sur tous les fichiers de données que nous avions à notre disposition. Nous noterons que compte-tenu des difficultés éprouvées, seuls nos algorithmes ot-opt ont été éprouvés. Néanmoins, malgré l'absence des algorithmes de type two-opt-*, nous avons été en mesure d'obtenir les résultats suivants :

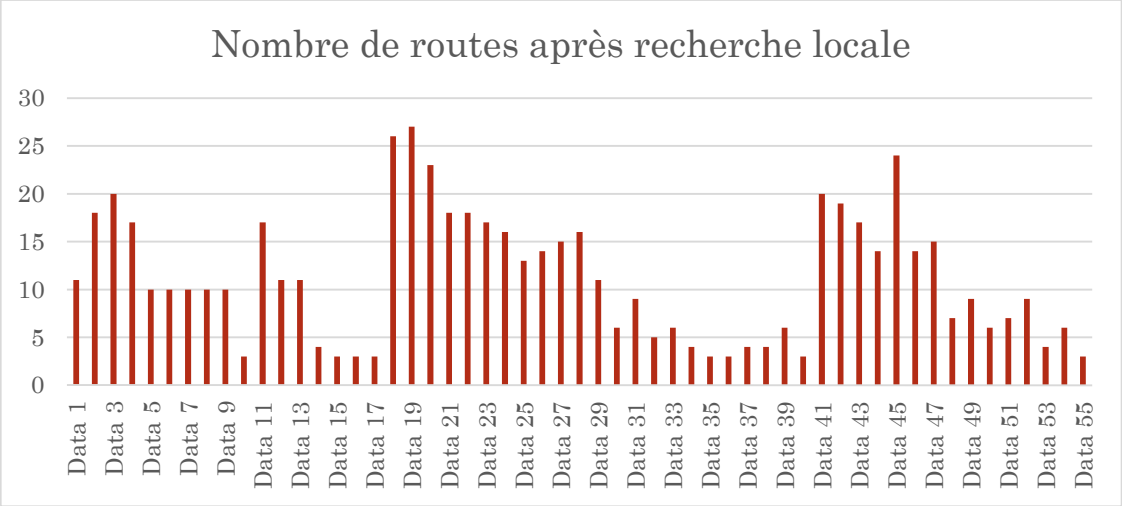


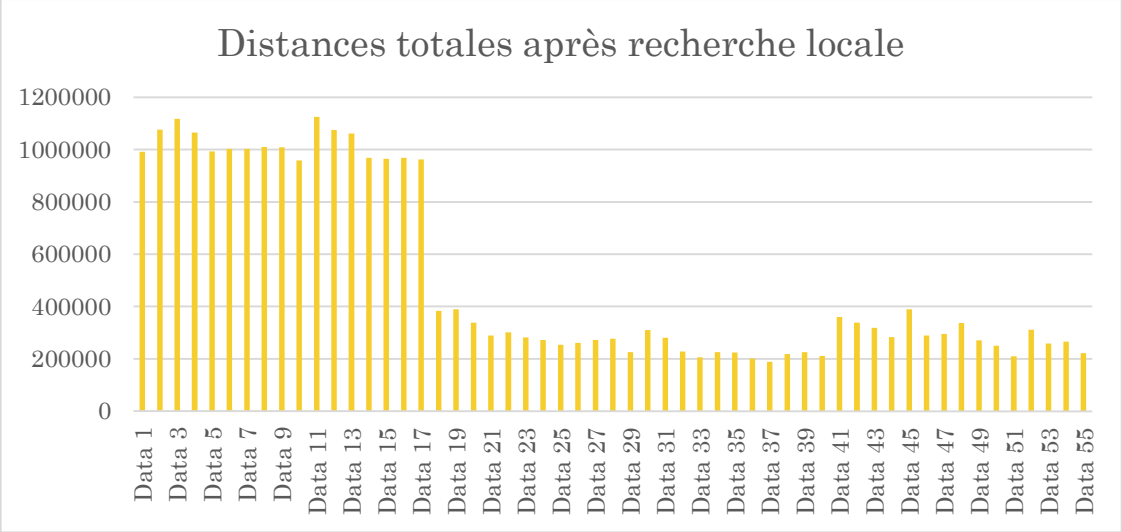
Ce graphique représente le nombre de tournées pour les solutions créées pour chaque fichier après l'exécution de notre heuristique d'insertion. Ici les résultats restent assez cohérents, et nous pouvons donc remarquer que notre heuristique permet de bons résultats de base.



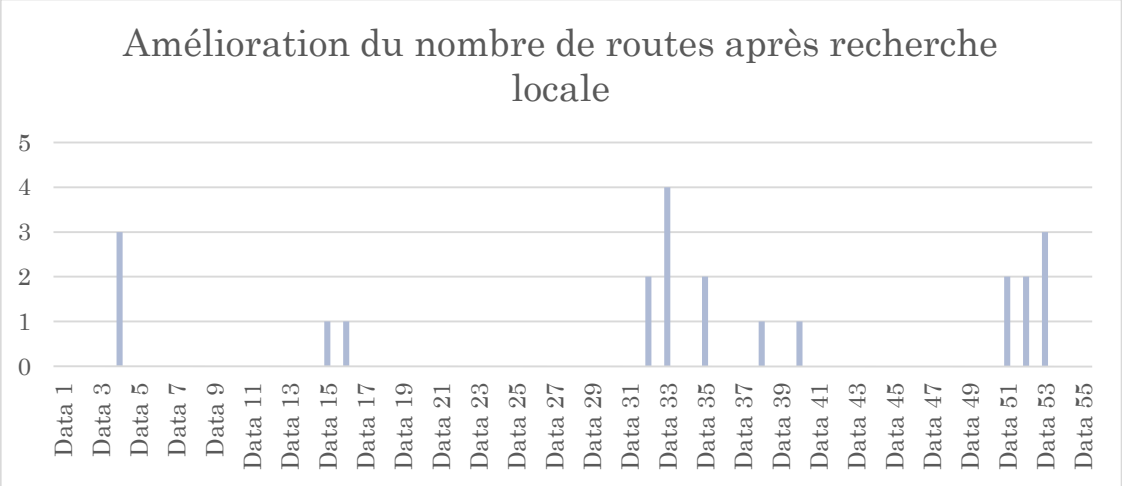
Pour ce graphique-ci, nous avons cherché à représenter la distance totale de toutes les tournées après l'exécution de l'heuristique d'insertion.

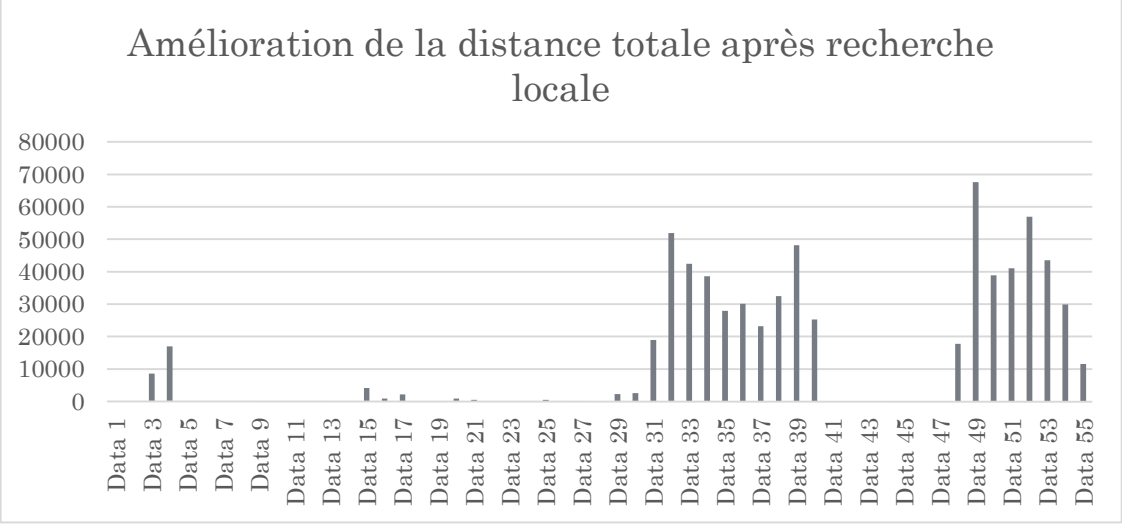
Nous avons cherché ensuite à comparé ses données après l'exécution de la recherche locale :





Après la recherche locale, nos résultats restent similaires, et l'absence de l'algorithme two-opt-* se fait ressentir, néanmoins certains fichiers de données ont pu être améliorés de façon remarquable, comme le montre les graphiques suivants :

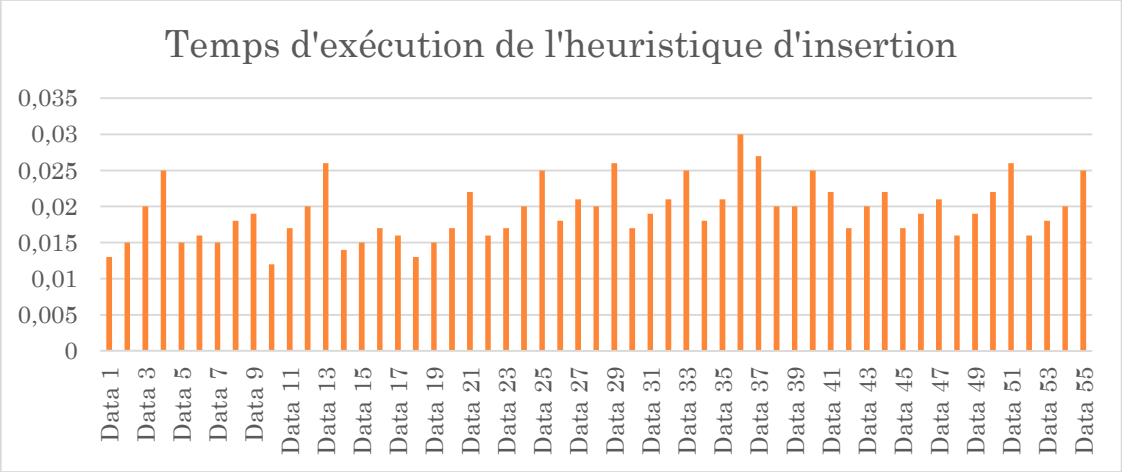




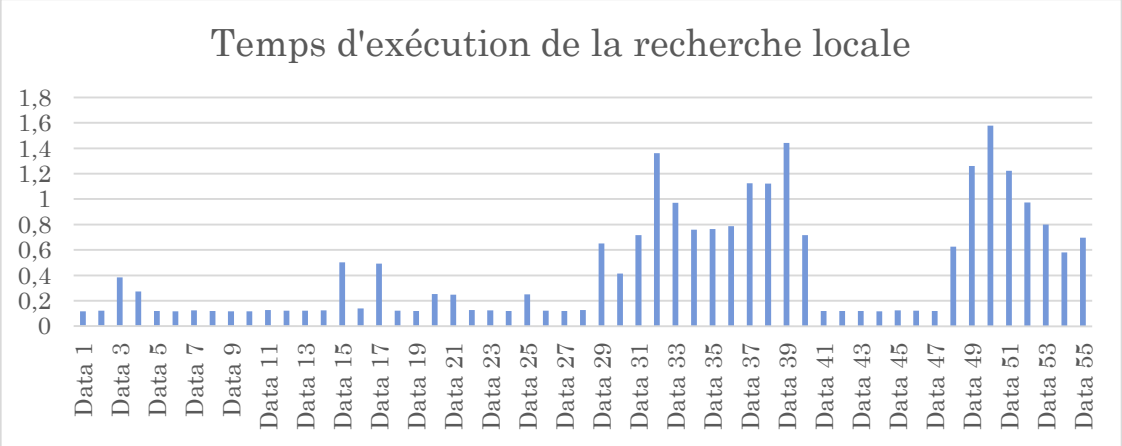
Ces graphiques sont ceux qui nous montrent le mieux à quel point notre algorithme de recherche locale est efficace. Ici, nous pouvons voir que nous améliorons beaucoup certains jeux de données, mais très peu certains autres. Nous pouvons en déduire que les heuristiques et les algorithmes de recherche locale utilisés jouent grandement dans l'amélioration du nombre de routes et de la distance, et que seul l'ensemble de ces algorithmes permet d'obtenir le résultat le plus optimal.

3.2. Temps d'exécution

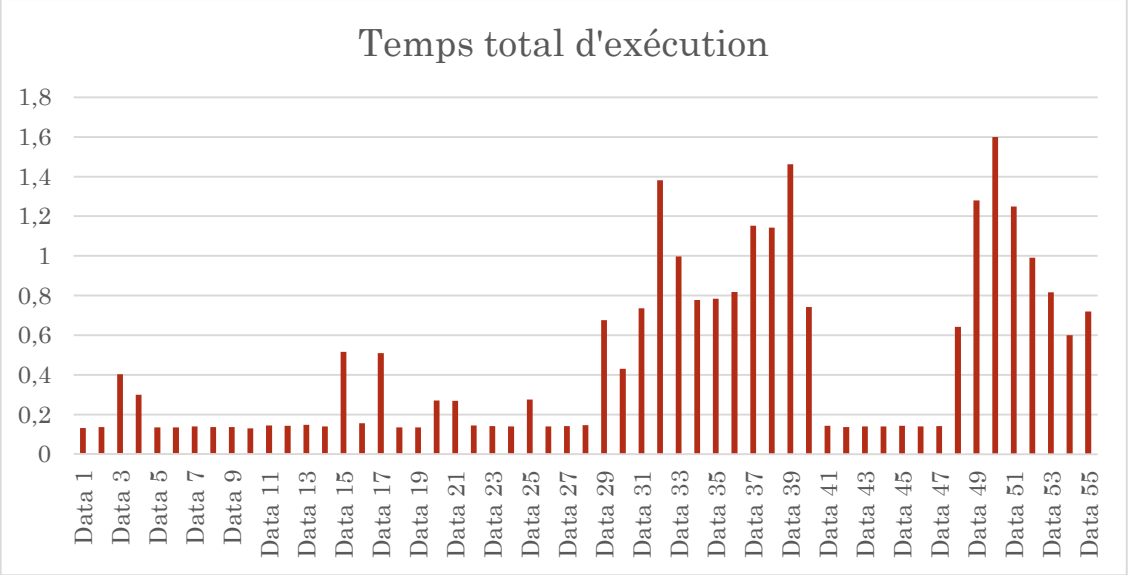
Nous avons également cherché à mesurer les temps d'exécution de notre programme. Toujours à l'aide des mêmes fonctions que précédemment, nous avons établi que notre programme mettait 6.611s à charger les différents fichiers de données puis à initialiser les structures de données.



S'exécutent ensuite l'heuristique d'insertion. Le graphique précédent permet d'établir en secondes le temps mis par chaque heuristique d'insertion selon les fichiers de données.



Concernant la recherche locale, nous pouvons constater que les temps dépendent de si la recherche locale effectue une action ou non (cf. les graphiques de la section précédente).



Nous obtenons donc les temps totaux suivants, pour chaque fichier de données. Au total, tous fichiers confondus, le programme met 25s94, ce qui même compte tenu du fait des algorithmes de recherche local manquants reste tout à fait convenable. Il sera noté que les mesures ont été faites en lançant le programme depuis Visual Studio 2015, avec un i5 cadencé à 3,7Ghz.

4. Conclusion

Nous avons implémenter un algorithme de tournée de véhicule en utilisant l'heuristique d'insertion. Nous avons ensuite implémenter un algorithme de recherche locale le ot opt mais nous avons rencontré des difficultés pour les autres.

Difficultés rencontrées:

Les algorithmes du 2opt* et du cross nous ont posé problème du fait qu'il fallait créer une nouvelle `Working Solution` à partir de la première (constructeur par copie) puis simuler pour chaque points (qui possédaient un gain potentiel) un changement de route mais les modifications sur la `Working solution` construite par le constructeur par copie modifiées aussi la `Working solution` originelle et rendez impossible un retour en arrière simple car le dit constructeur par copie copié les adresses de la première dans sa nouvelle `Working Solution` rendant impossible des modification sans effet de bord pour la `Working Solution` originelle.

Les principales difficultés finalement ont été l'utilisation du code déjà implémenté : en effet les fonctions n'étaient pas commentées, aussi nous avons dû passé une dizaine d'heure à comprendre quels paramètres donnés, sous quel condition etc.

S'il fallait refaire ce TP nous aurions commencé par créer notre propre structure pour avoir une meilleure vision de nos possibilité et de ce qu'il reste à implémenter.