Fil Rouge ICO - v1 - Groupe ICOnique

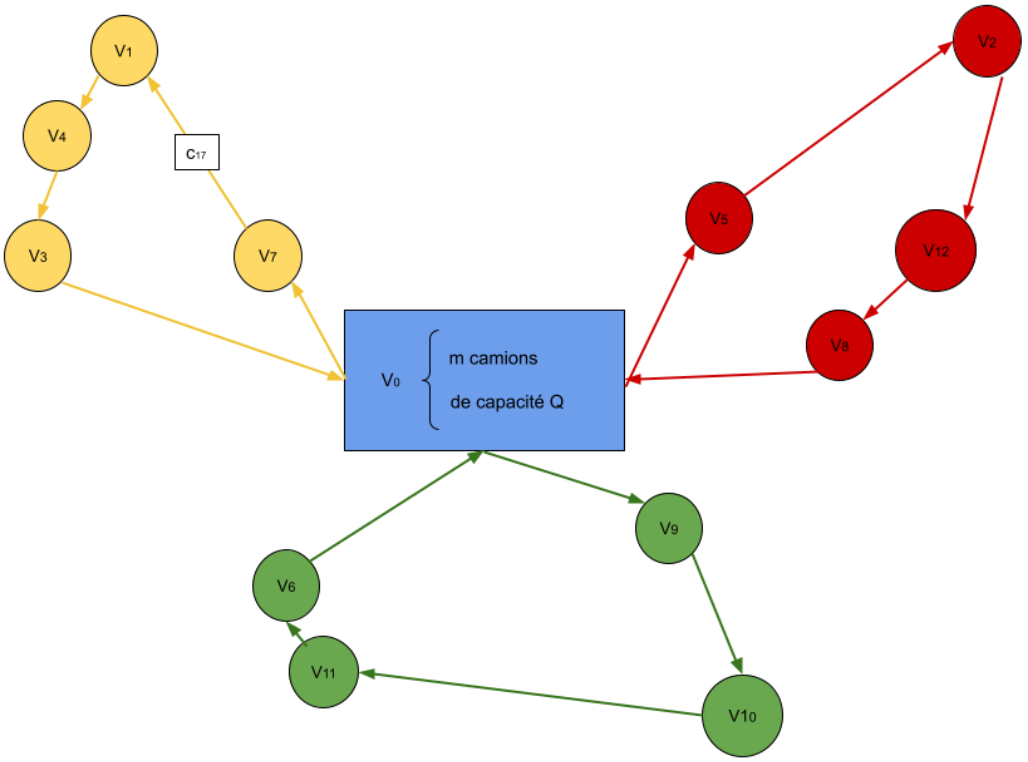
Yishan Sun, Simon Kurney, Pablo Aldana, Cédric Jung, Baptiste Deconihout, Zoé Poupardin

I - Introduction

Problématique :

Le Vehicle Routing Problem (VRP), en français problème de tournée de véhicule est un problème algorithmique d'optimisation NP-complet. Il s'agit d'une analogie généralisée du célèbre problème du voyageur de commerce. L'objectif du VRP est de déterminer l'ensemble des routes, recouvrant la totalité des nœuds clients à livrer, minimisant le coût total de la tournée , tout en respectant les contrainte de capacité des véhicules (le coût prend en compte le nombre de camions affrétés à la tourner, ainsi que le nombre de routes empruntées).

Plus concrètement, on peut considérer le problème en dessinant un graphe complet comportant (N + 1) nœuds ; un nœud central V_0 représente le dépôt où stationnent K véhicules identiques de capacité Q. Les autres nœuds V_i, i ∈ {1,2,…,N} représentent les clients i caractérisés par une demande de produit q_i, et attendant leur colis dans un intervalle de temps . Les arcs (i, j) représentent la possibilité d'un trajet direct du client i au client j avec un coût de transport de C_{ij}.



Méthode et approche de résolution :

Comme pour l'ensemble des problèmes NP-complet, obtenir de manière optimale une solution complète est quasi impossible sur un grand nombre de données. L'approche de résolution d'un tel problème est de trouver une solution qui soit la plus proche possible de la solution optimale. Pour parvenir à une telle solution, on fait appel à des méthodes approchées, par exemple des métaheuristiques.

Le VRP étant un problème d'optimisation combinatoire, la fonction de calcul du coût d'une solution prend de nombreux paramètres en considération. Nous avons décidé d'introduire progressivement les différents paramètres aux calculs du coût.
Nous avons pris en compte les paramètre dans l'ordre suivant :

1. Nombres de camions et coût des routes
2. Volumes des camions
3. Temps et fenêtre de livraison

Dans cette première partie de projet, nous allons programmer puis confronter trois métaheuristiques :

1. Méthode de recherche tabou
2. Recuit simulé
3. Algorithmes génétiques : méthode d'optimisation stochastique basés sur le mécanisme de la sélection naturelle

II - Algorithme tabou

a) Description de l'algorithme

On retrouve l'algorithme dans le fichier *taboue.ipynb*.

L'objectif est d'obtenir le meilleur ordonnancement pour desservir tous les clients dans le temps imparti. Un ordonnancement est meilleur qu'un autre si son coût total est plus faible que ce dernier.

Nous abordons ce problème en considérant une liste de camions, chacun défini par une liste de chiffres correspondant aux clients qui vont être livrés. Au début de l'algorithme, nous commençons avec une solution que nous cherchons à améliorer en modifiant l'ordre des clients livrés et le nombre de camions. Pour cela, nous avons recours à différentes fonctions de voisinage (exchange, inverse, exchange_inside) permettant d'inter-changer cet ordre.
Dans un premier temps, il est nécessaire de savoir si les listes trouvées peuvent correspondre à des solutions. Il s'agit du rôle de la fonction **acceptable** qui vérifie si le mouvement pour obtenir la liste n'a pas déjà été fait (appartenance à la liste tabou) et si la liste est meilleure que la précédente selon le critère d'aspiration.
Dans un second temps, le critère pour déterminer si la solution obtenue est meilleure que la celle déterminée jusqu'à présent est le critère du coût, obtenu par la fonction **coût** est calculé selon le trajet parcouru par l'ensemble des camions.

Les fonctions de voisinage permettent la création de nouvelles solutions. Elles se basent sur des opérations élémentaires telles que des inversions d'ordre de livraisons de clients, de nouveaux camions en divisant des camions préexistants, des fusions de camions et enfin des créations ou des suppressions de camions.

Ainsi, la fonction **exchange** permet de modifier l'ordre de livraison de clients livrés en ne modifiant pas le nombre de clients par camion. Par exemple, la solution [[4,2,5],[1,3]] peut devenir [[3,2,5],[4,1]]. De la même façon, la fonction **relocate** permet d'attribuer les clients à d'autres camions (existant ou non) sans tenir compte du nombre de clients par camion, en constituant des paires de clients. Enfin, la fonction **exchange_inside** permet la modification de l'ordre de livraison des clients au sein d'un même camion. Dans les trois cas, les fonctions vérifient si les solutions proposées sont acceptables (en appelant la fonction du même nom).

La structure de l'algorithme tabou est le suivant :

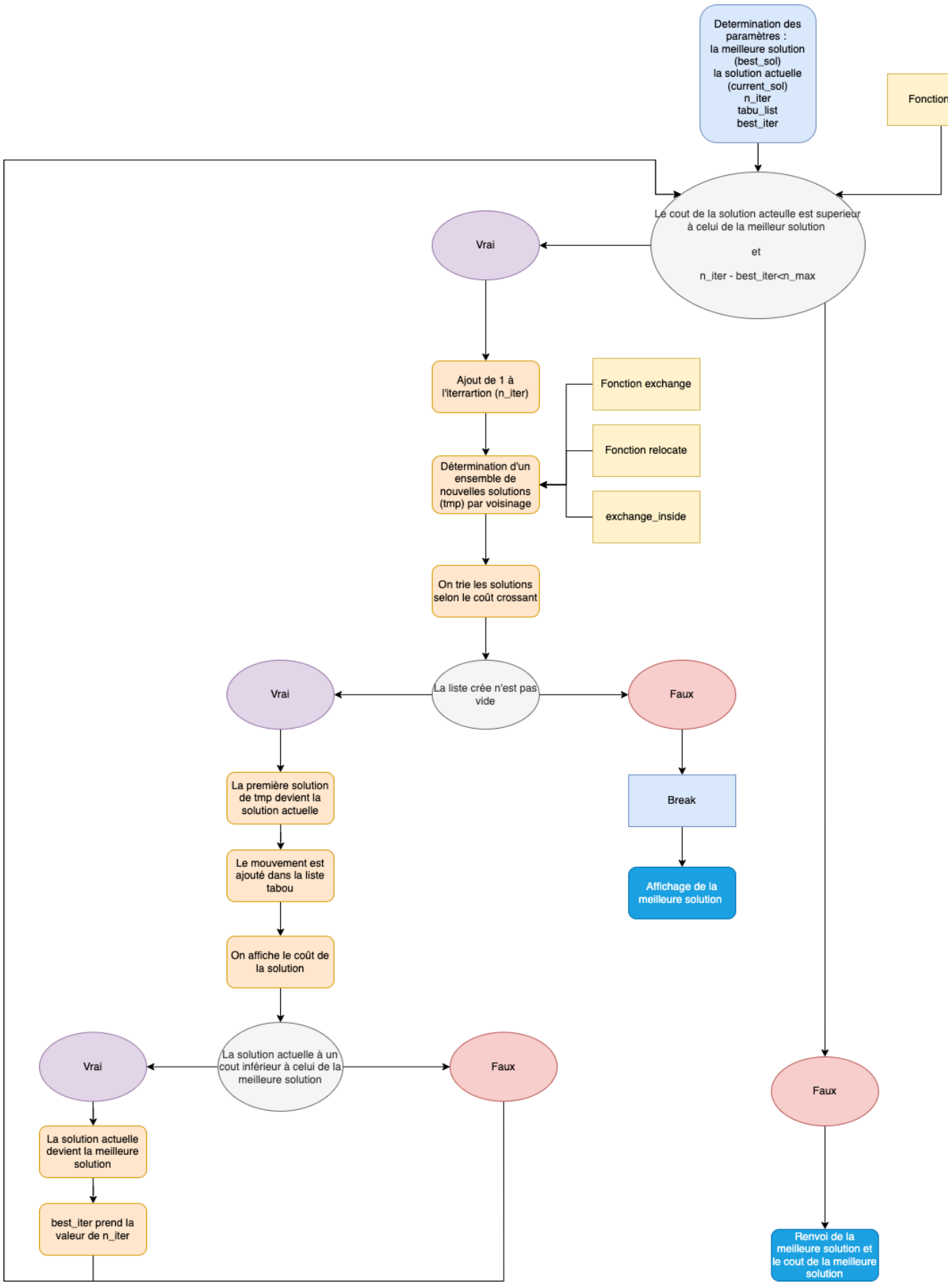
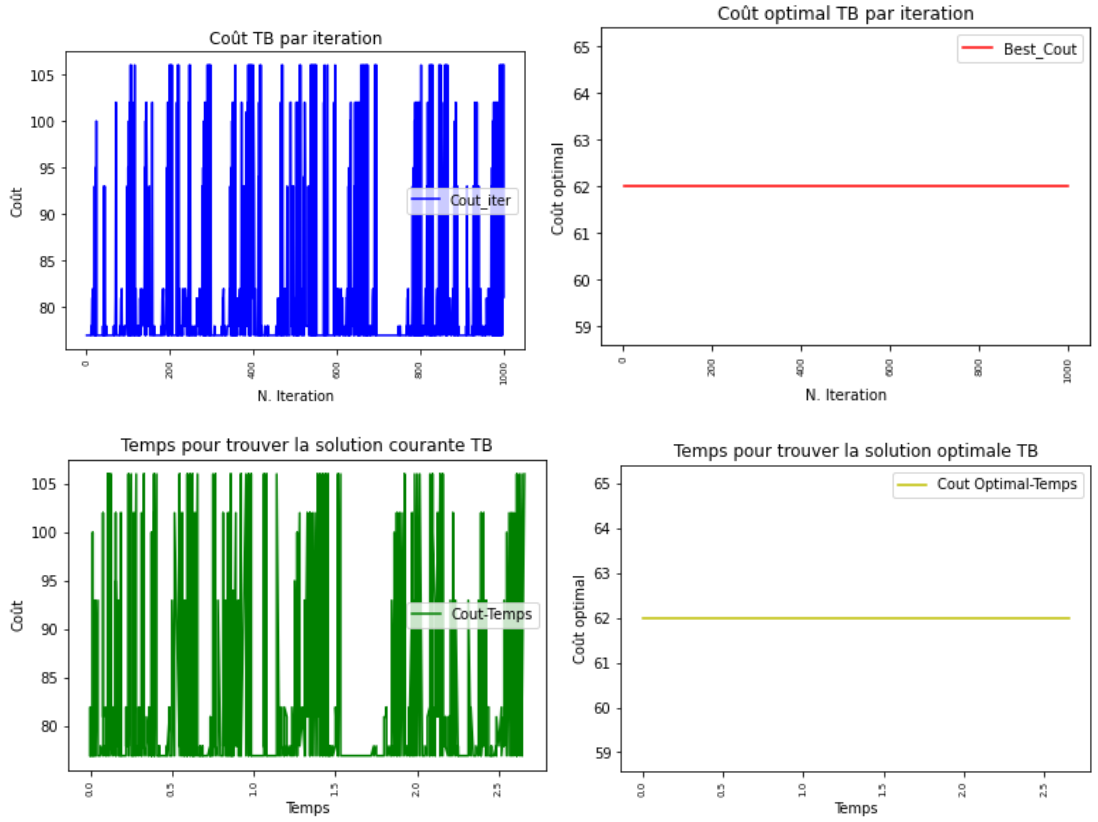

Determination des paramètres :
la meilleure solution (best_sol)
la solution actuelle (current_sol)
n_iter
tabu_list
best_iter
Fonction coût
Le cout de la solution acteulle est superieur à celui de la meilleur solution
et
n_iter - best_iter<n_max
Vrai
Faux
Ajout de 1 à l'iterrartion (n_iter)
Fonction exchange
Fonction relocate
exchange_inside
Détermination d'un ensemble de nouvelles solutions (tmp) par voisinage
On trie les solutions selon le coût crossant
La liste crée n'est pas vide
Vrai
Faux
La première solution de tmp devient la solution actuelle
Le mouvement est ajouté dans la liste tabou
On affiche le coût de la solution
Break
Affichage de la meilleure solution
La solution actuelle à un cout inférieur à celui de la meilleure solution
Vrai
Faux
La solution actuelle devient la meilleure solution
best_iter prend la valeur de n_iter
Faux
Renvoi de la meilleure solution et le cout de la meilleure solution

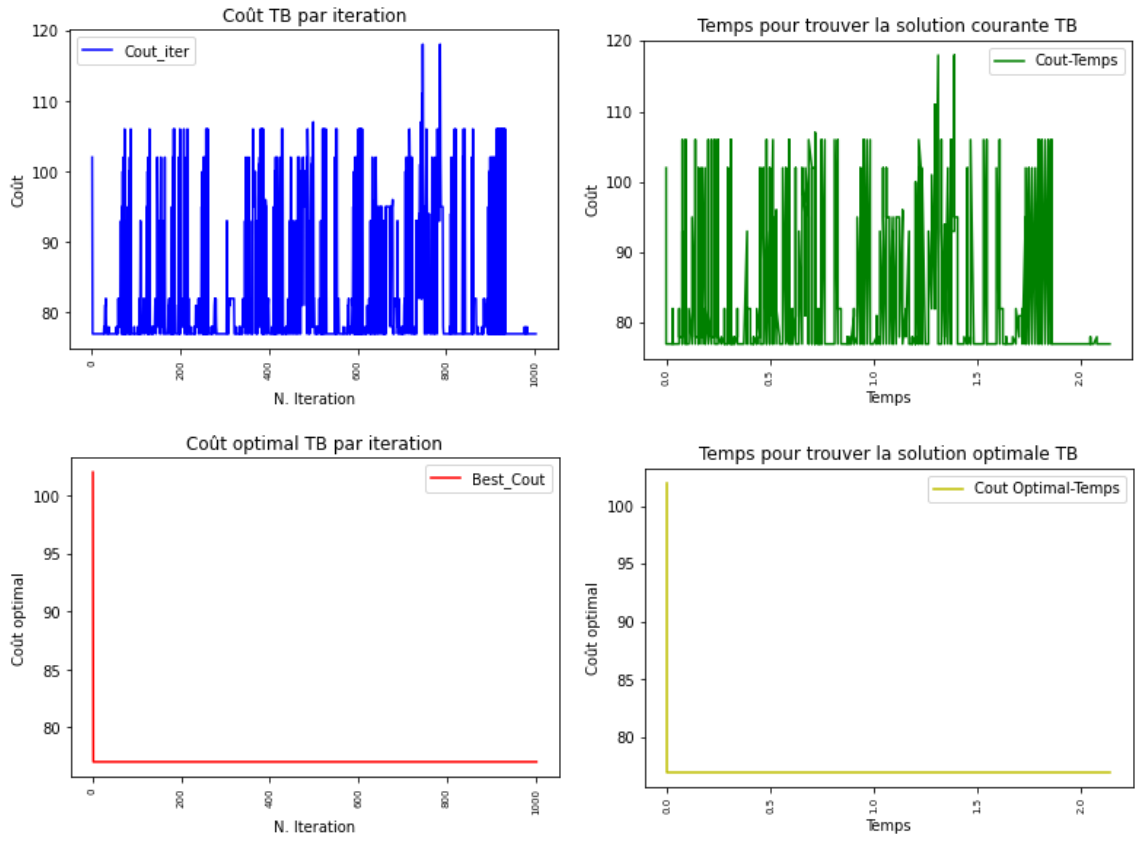
Figure 1 : Schéma du fonctionnement général de l'algorithme Tabou

b) Résultats et simulations

On a dessiné des graphiques pour afficher la variation de coût actuel et de coût optimal en fonction du nombre d'itération et de temps.
On est parti d'abord de notre exemple simple avec 5 clients noté [1, 2, 3, 4, 5] avec un poids ω=5 pour le calcul du coût, un nombre d'itération maximum de 100 et un facteur d'aspiration de 100.
En partant de la solution initiale [[1, 2, 5], [4, 3]] on obtient les courbes suivante et la solution [[1, 2, 5], [4, 3]] d'un coup de 62



En partant de la solution initiale [[1, 2, 5], [4, 3]] on obtient les courbes suivante et la solution [[1, 5], [4], [2, 3], []] d'un coup de 77



c) Raffinement: ajout de la notion de capacité

On retrouve cela dans le fichier *taboue-with-capacities.ipynb*.

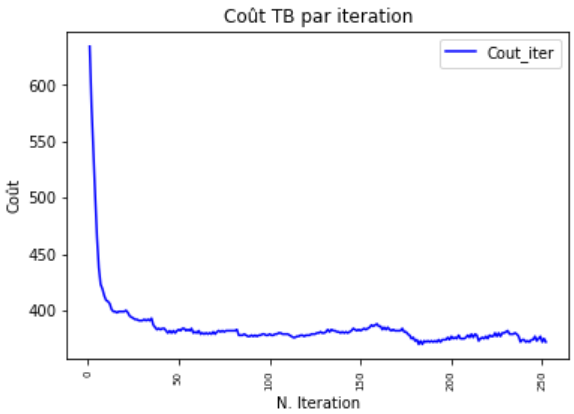
Nous avons décidé d'implémenter la notion de capacité dans une deuxième version de l'algorithme tabou, en implémentant une condition dans la recherche de voisinage. C'est-à-dire dans les fonctions, **relocate**, **exchange** et **exchange_inside**.

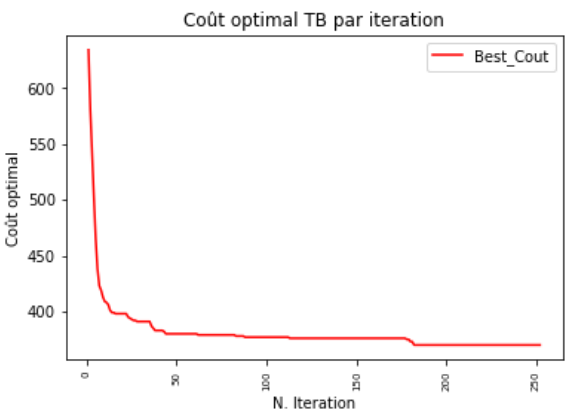
Nous avons implémenté la fonction **filter_on_capacities** qui permet de ne garder que les solutions qui répondent à une capacité maximale de chaque camion pour chaque route.

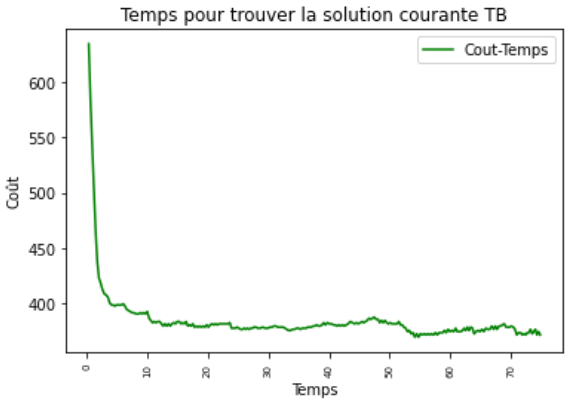
Dans le cas la fonction **relocate**, si la fonction **filter_on_capacities** retourne une liste de solution vide, on va tenter de rajouter une route à la solution donnée en paramètre à **relocate** et d'y déplacer un client afin de faire en sorte que d'avoir un voisinage. Cette façon de faire nécessite d'être amélioré

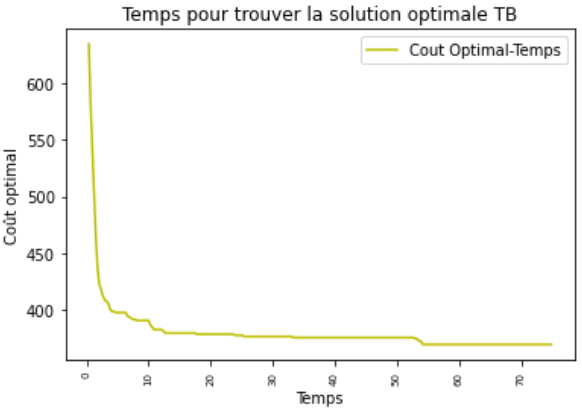
On est parti d'abord de notre exemple simple avec 51 clients noté avec un poids ω=5 pour le calcul du coût, un nombre d'itération maximum de 100 et un facteur d'aspiration de 3.

Avec exemple avec 51 clients, avec capacité maximale des camion de 100, et chaque client a une demande d'une taille 1, on obtient un coup de 370:

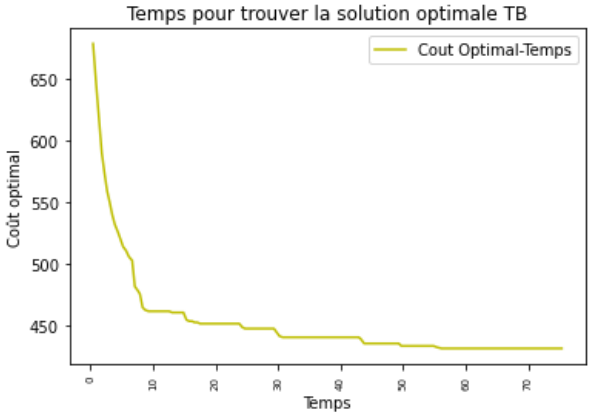


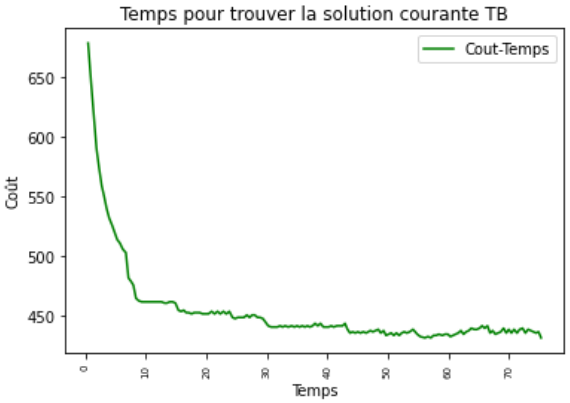


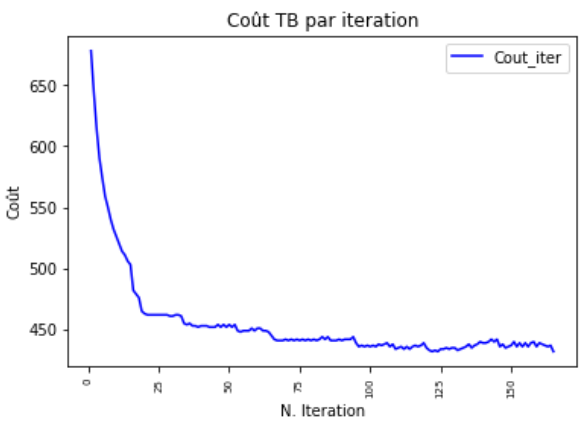


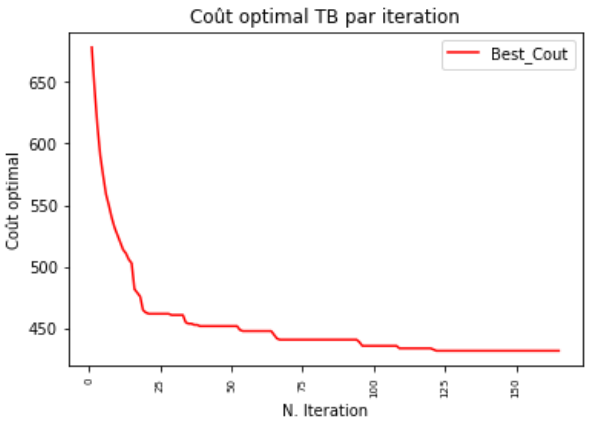


Avec exemple avec 51 clients, avec capacité maximale des camion de 100, et chaque client a une demande d'une taille 3, on obtient un coût de 432:









III - Recuit simulé

a) Description de l'algorithme

On retrouve l'algorithme dans le fichier *rs.ipynb.*

Une seconde approche peut être celle du recuit simulé. Bien que cette méthode se base elle aussi sur la modification d'une solution acceptable par l'utilisation de fonctions de voisinage (décrites ci-dessus), celle-ci diffère en acceptant de garder une solution qui n'est pas meilleure que la précédente en supposant que cette dernière correspond à maximum locale. Ainsi, cette méthode fait appel à une probabilité liée au temps de recherche de la solution (voir figure 2).

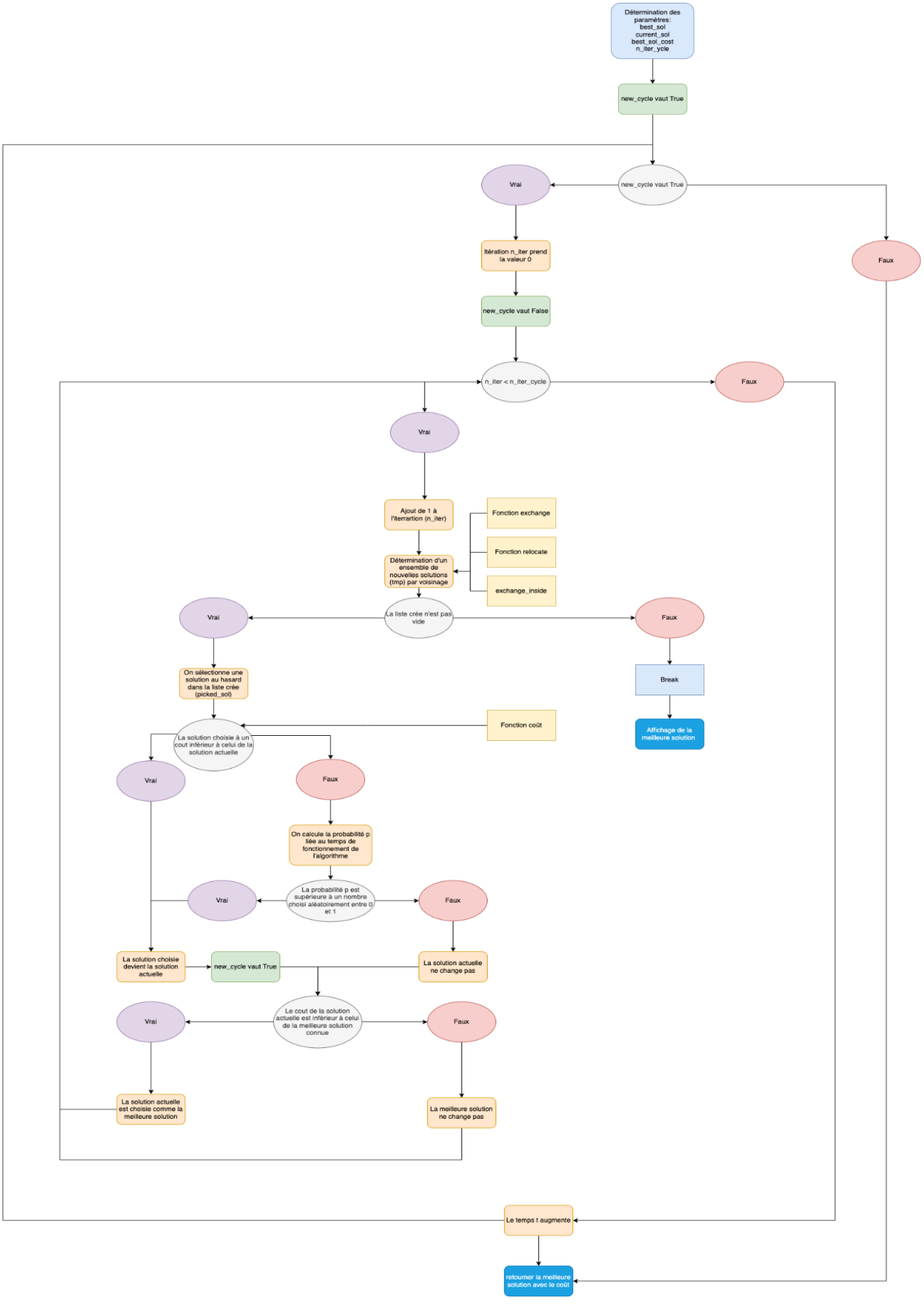
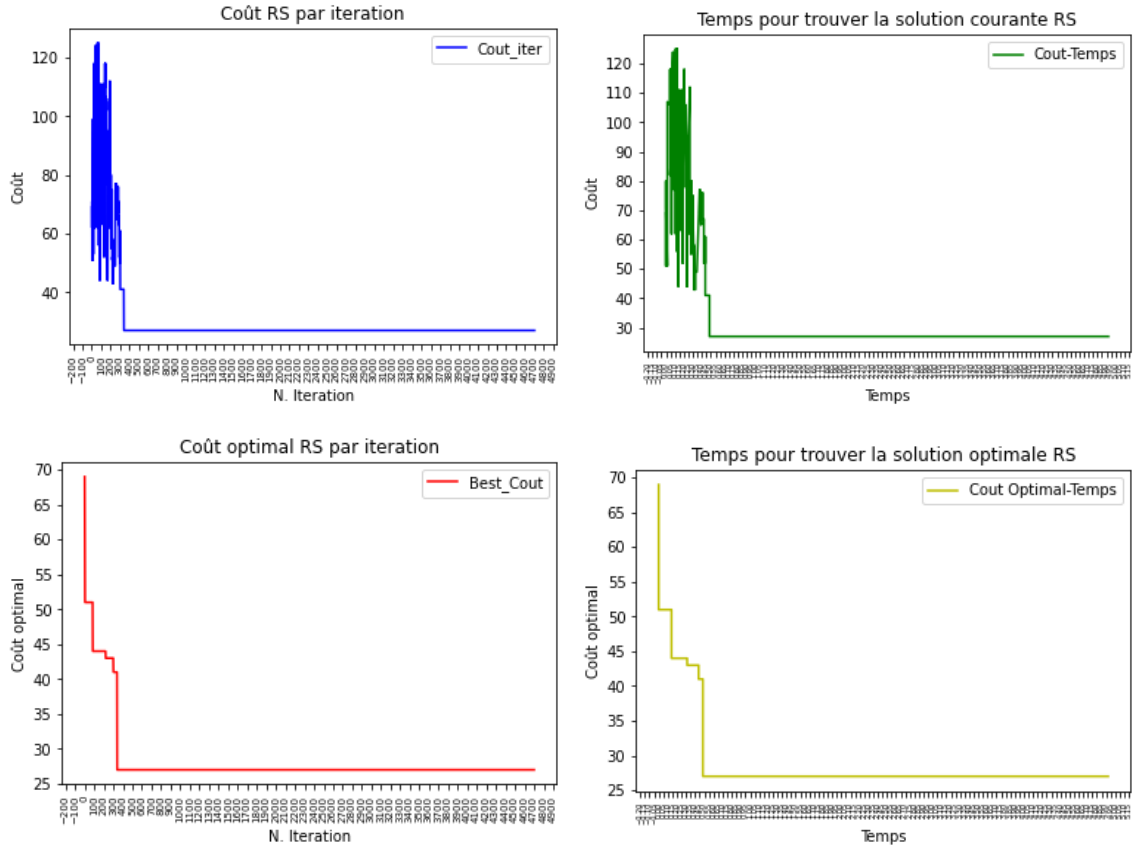

Détermination des paramètres:
best_sol
current_sol
best_sol_cost
n_iter_cycle
new_cycle vaut True
new_cycle vaut True
Vrai
Faux
Itération n_iter prend la valeur 0
new_cycle vaut False
n_iter < n_iter_cycle
Faux
Vrai
Ajout de 1 à l'iterrartion (n_iter)
Fonction exchange
Fonction relocate
exchange_inside
Détermination d'un ensemble de nouvelles solutions (tmp) par voisinage
La liste crée n'est pas vide
Vrai
Faux
Break
Affichage de la meilleure solution
On sélectionne une solution au hasard dans la liste crée (picked_sol)
Fonction coût
La solution choisie à un cout inférieur à celui de la solution actuelle
Vrai
Faux
On calcule la probabilité p liée au temps de fonctionnement de l'algorithme
La probabilité p est supérieure à un nombre choisi aléatoirement entre 0 et 1
Vrai
Faux
La solution choisie devient la solution actuelle
new_cycle vaut True
La solution actuelle ne change pas
Le cout de la solution actuelle est inférieur à celui de la meilleure solution connue
Vrai
Faux
La solution actuelle est choisie comme la meilleure solution
La meilleure solution ne change pas
Le temps t augmente
retourner la meilleure solution avec le coût

Figure 2 : Schéma du fonctionnement général de l'algorithme recuit simulé

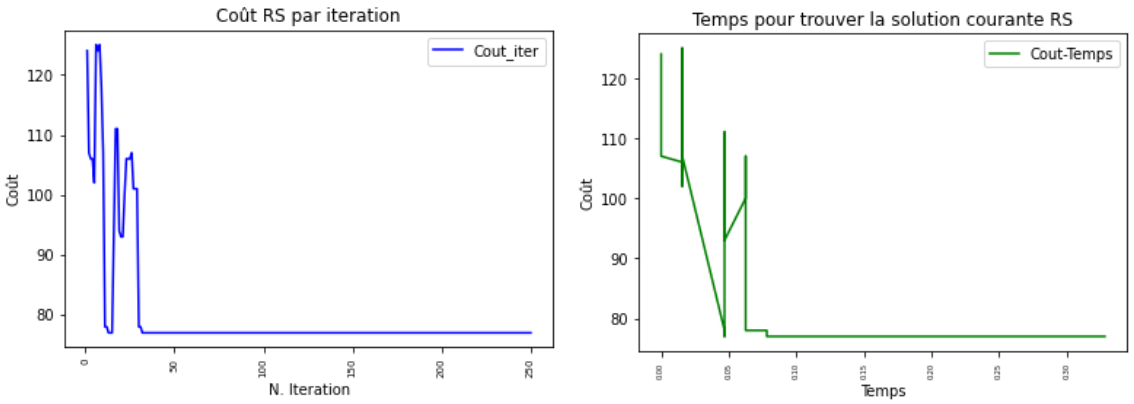
On retrouve l'algorithme avec le même raffinement que celui présenté en II)c) dans le fichier *rs-with-capacities.ipynb*.

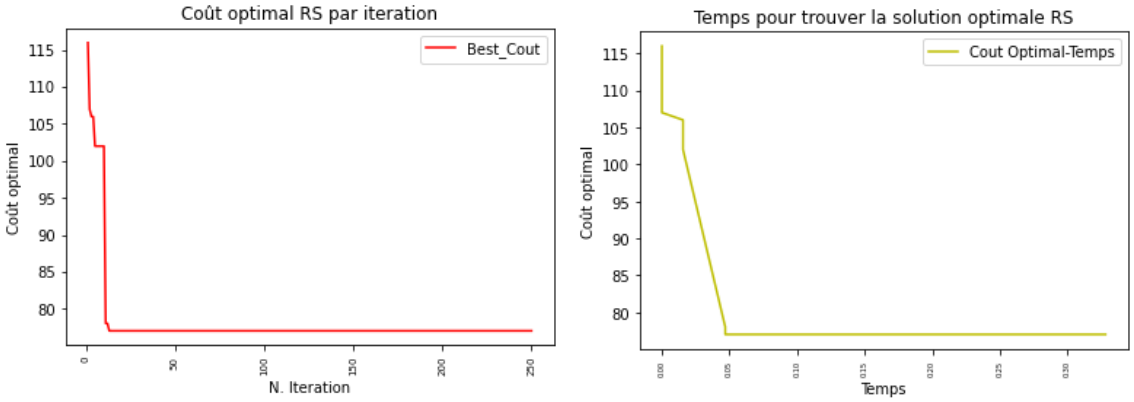
b) Résultats et simulations

On considère toujours un poids ω=5 pour le calcul du coût.
A partir de même exemple que tabou, on a aussi dessiné les graphes dans ce cas. On peut constater qu'il y a déjà beaucoup d'itérations.

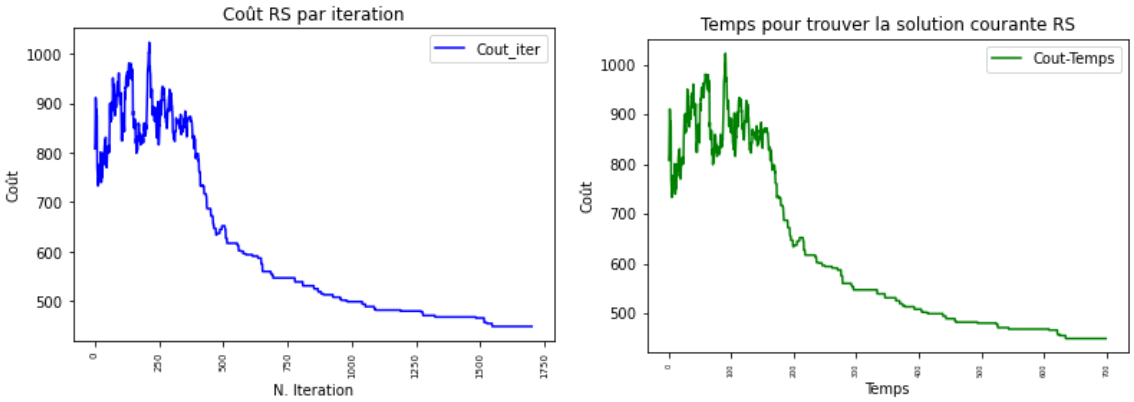


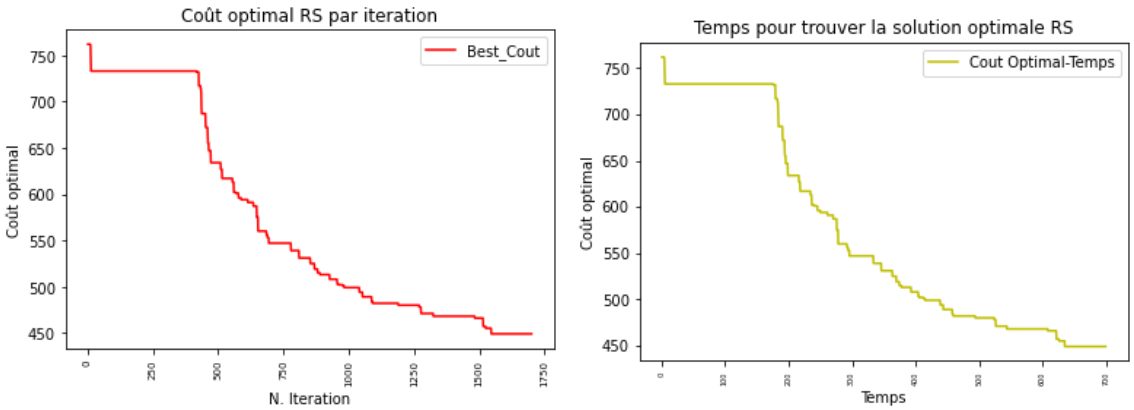
Après avoir ajouté la notion de capacité et avoir modifié nombre de l'itération, on a obtenu les simulations :





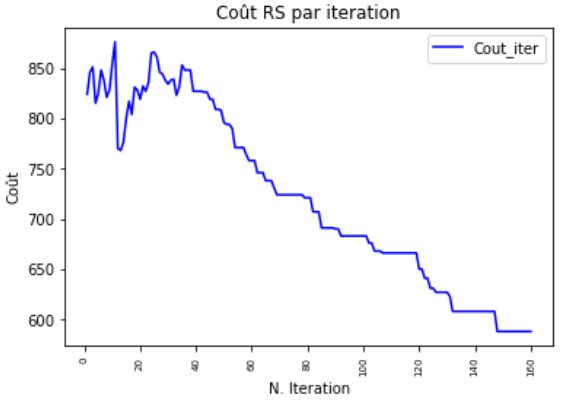
Avec le même exemple complexe, on a obtenu les simulations suivantes. Nous avons été forcé d'interrompre le calcul car trop d' itérations, l'algorithme a pris 11 minutes 45 secondes pour trouver la solution.

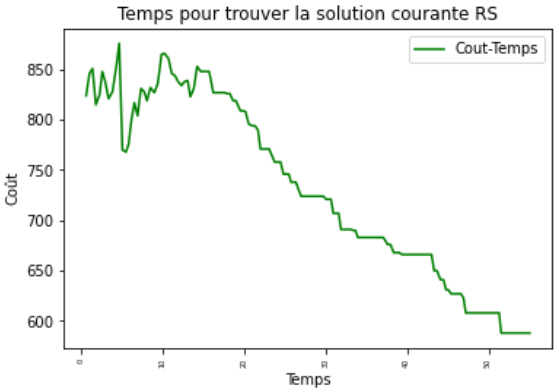


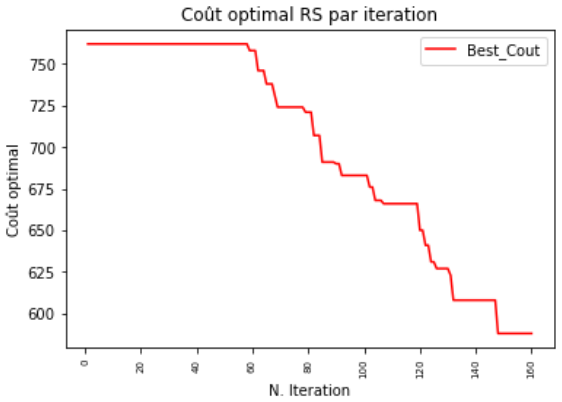


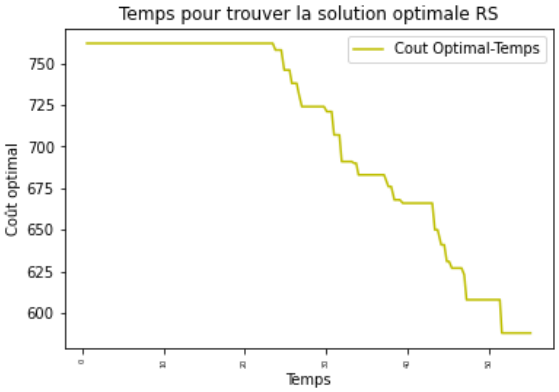
Après avoir ajouté la notion de capacité et avoir modifié nombre de l'itération, on a obtenu les simulations :

Avec la capacité 51 clients ayant chacun une demande de volume 1:

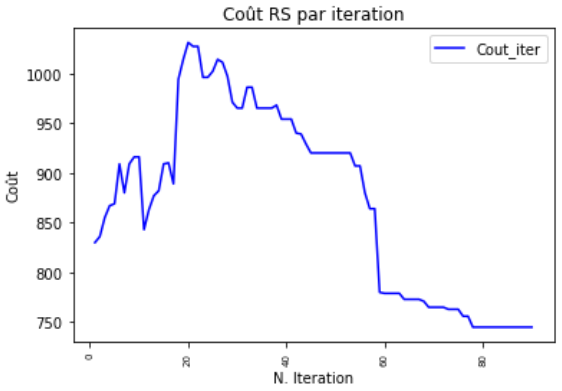


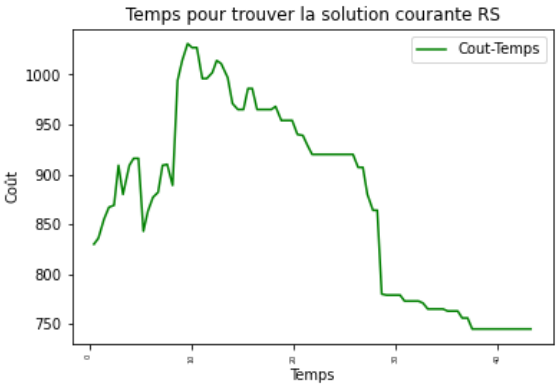


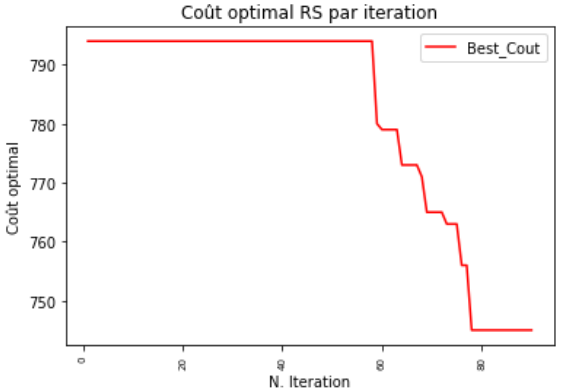


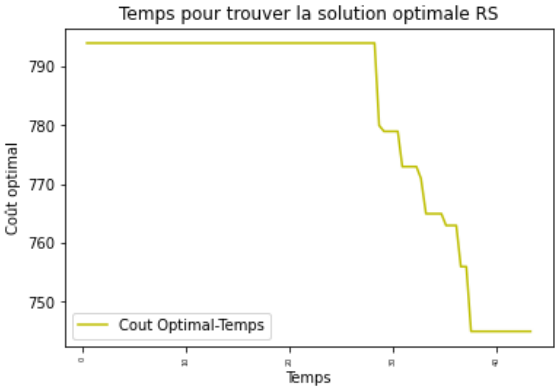


Avec la capacité 51 clients ayant chacun une demande de volume 3:









IV - Algorithme génétique

a) Description de l'algorithme

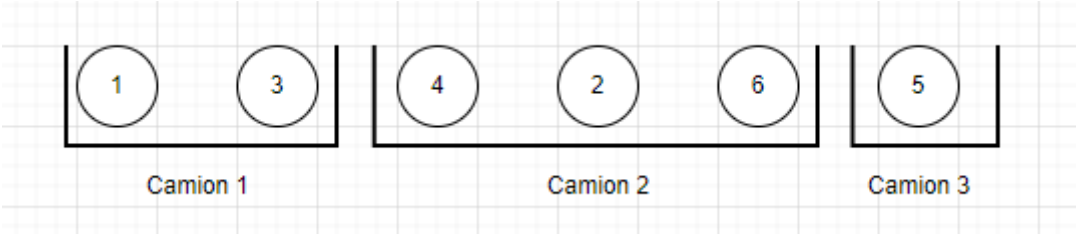
Dans ce cas, on applique l'algorithme génétique vu dans le cours.

- Etape 0 : Définir un codage du problème
- Étape 1 : t:=0, créer une population initiale de N individus P(0) = x1,x2,...,xN
- Etape 2 : Evaluation
    - Calculer la force F(xi) de chaque individu xi, i=1…N
- Etape 3 : Sélection
    - Sélectionner N individus de P(t) et les ranger dans un ensemble S(t). Un même individu de P(t) peut apparaître plusieurs fois dans S(t)
- Etape 4 : Recombinaison Grouper les individus de S(t) par paire, puis, pour chaque paire d'individus :
    - avec la probabilité Pcross, appliquer le croisement à la paire et recopier la progéniture dans S(t+1) (la paire d'individus est éliminée, elle est remplacée par sa progéniture),
    - avec la probabilité 1-Pcross, recopier la paire d'individus dans S(t+1)

  Pour chaque individu de S(t+1) :

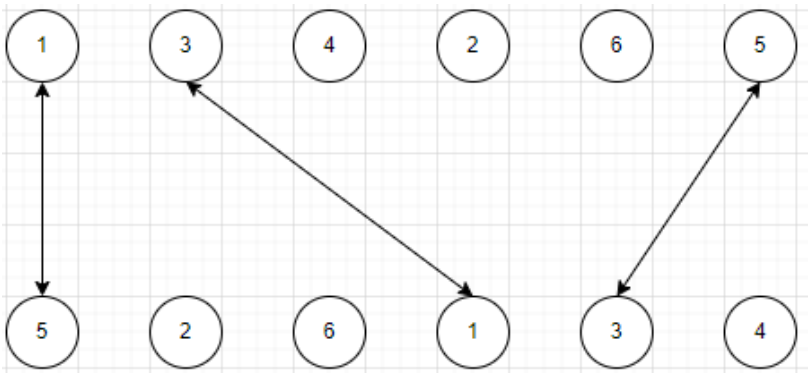
    - avec la probabilité Pmut, appliquer la mutation à l'individu le recopier dans P(t+1)
    - avec la probabilité 1-Pmut, recopier l'individu dans P(t+1)
- Etape 5 : Incrémenter t et reprendre à l'étape 2 jusqu'à un critère d'arrêt

On démarre avec l'ordonnancement, qu'on a utilisé pour les clients. On a choisi d' utiliser une chaîne de clients, qui représente la répartition dans le camion. Principalement, la chaîne sera lue à la fin du algorithmique et on va remplir chaque camion avec la capacité maximale c'est-à-dire lorsque le paquet du client suivant ne passe plus dans le camion, on le met dans le prochain. Pour éviter des trajets qui ne sont pas logiques, on donne en plus la possibilité pour l'algorithme de choisir un nouveau camion, si ce soit moins cher.



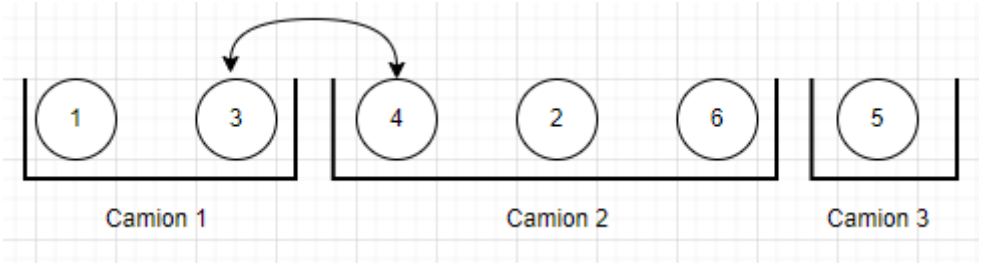
i. Le croisement

Vu que la coupage de la manière de faire le croisement présentée dans le cours n'est pas réalisable, on choisit trois clients aléatoires, dont l'ordonnancement sera changé. Plus particulier, on change le premier de trois dans chaque liste et on les échange, puis les deuxièmes et à la fin les troisièmes, comme on peut voir dans la graphique affichée au-dessous.



ii. La mutation

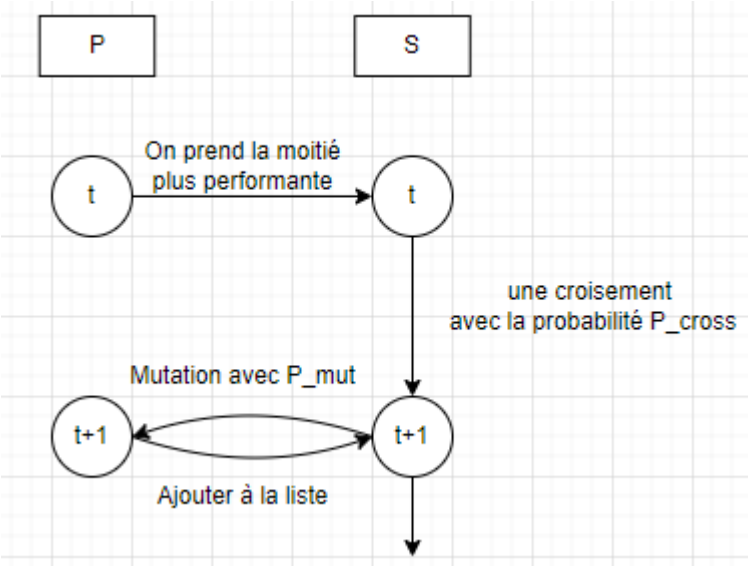
Pour la mutation on change l'ordonnance de deux clients côté à côté. On prend un client aléatoire de 1 à N-1 et change sa position avec son voisin à droite.



iii. Organisation

Alors on organise la structure de notre algorithme. On prend d'abord la moitié de la population P(t) qui performe le mieux comme S(t). Puis, on les met en couple et on effectue un croisement avec la probabilité P_cross et on garde tout dans S(t+1). Maintenant les

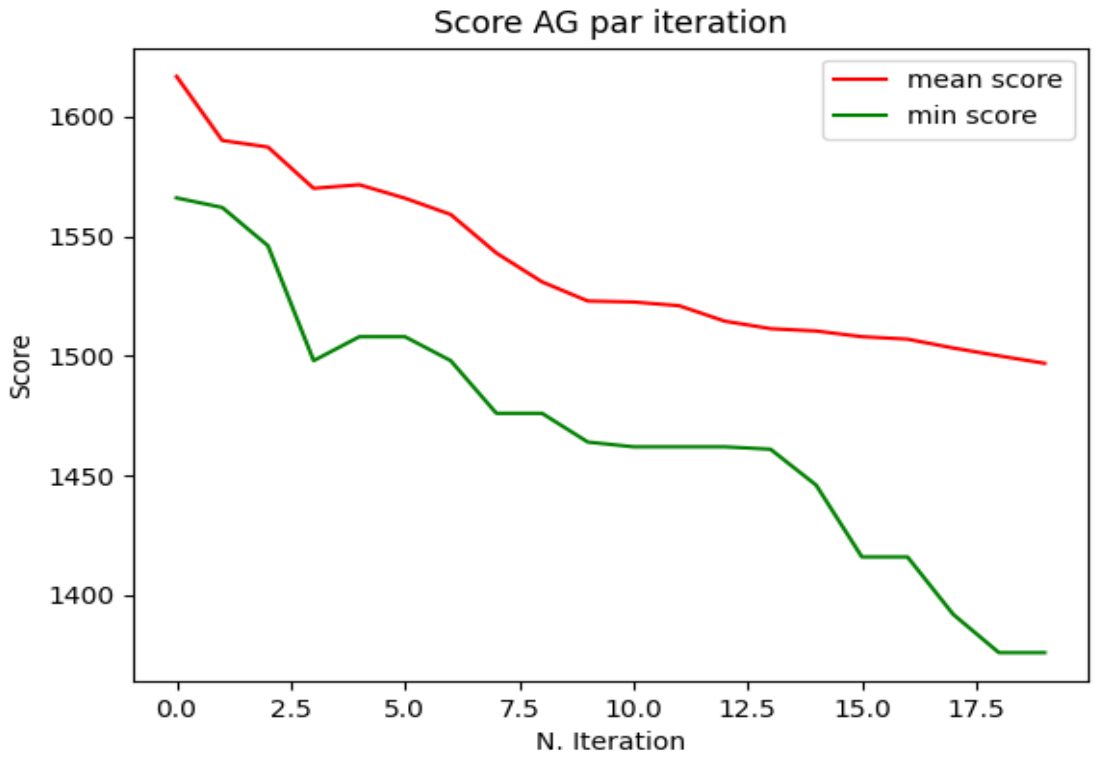
gènes mutent avec la probabilité P_mut et seront stockés en P(t+1). Enfin, on les rejoint avec les gènes en S(t+1) et on peut recommencer le processus pour t+2.

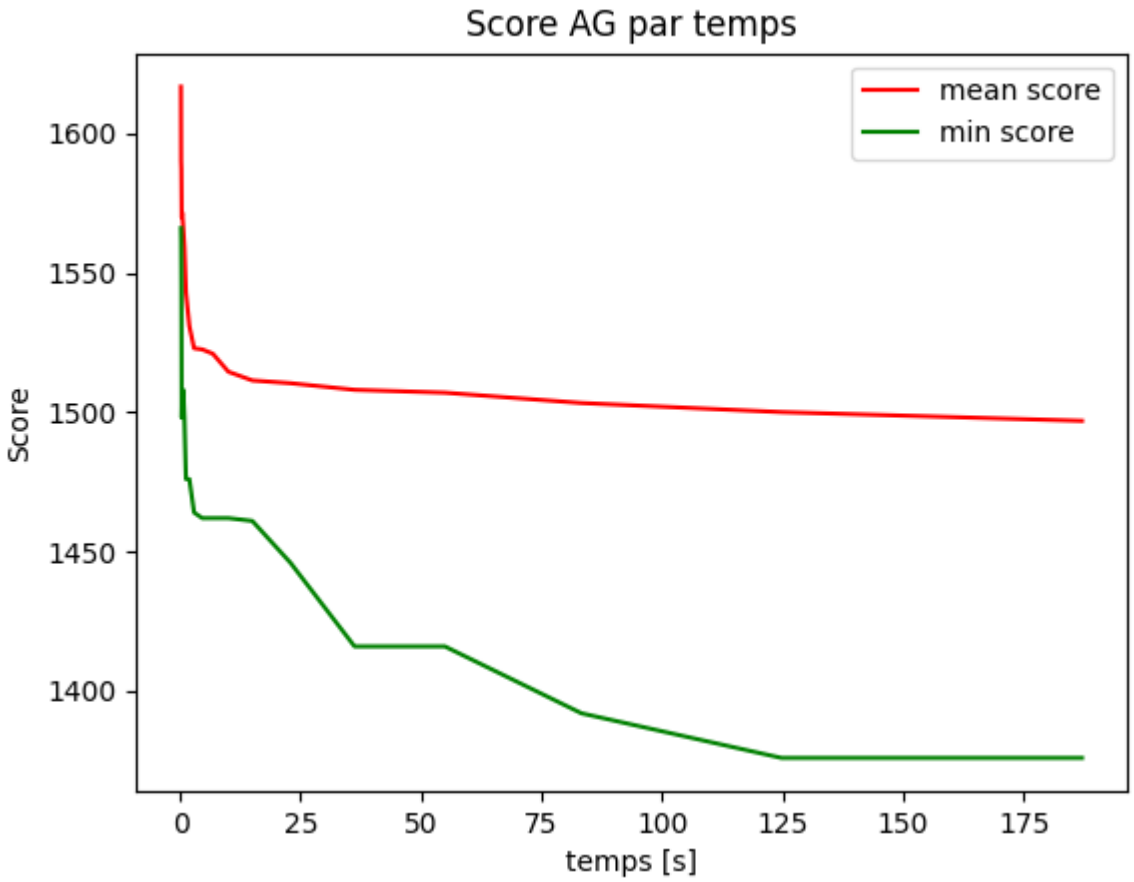


b) Résultats et simulations

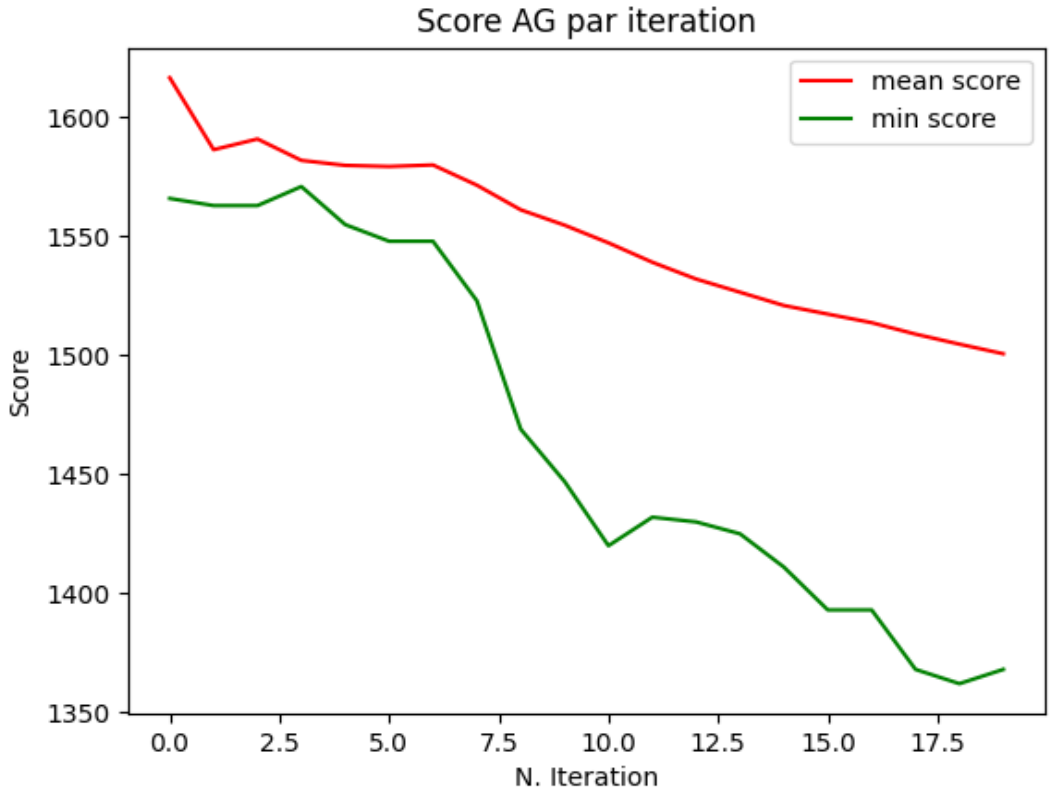
Simulation avec la matrice créée comportant 52 clients. Chaque client a une demande aléatoire entre 1 et 100. Et les camions ont une capacité totale de 100. La pénalité d'utiliser un nouveau camion est de 5. Nous lançons une simulation de 20 générations avec une population initiale de 4 chromosomes.

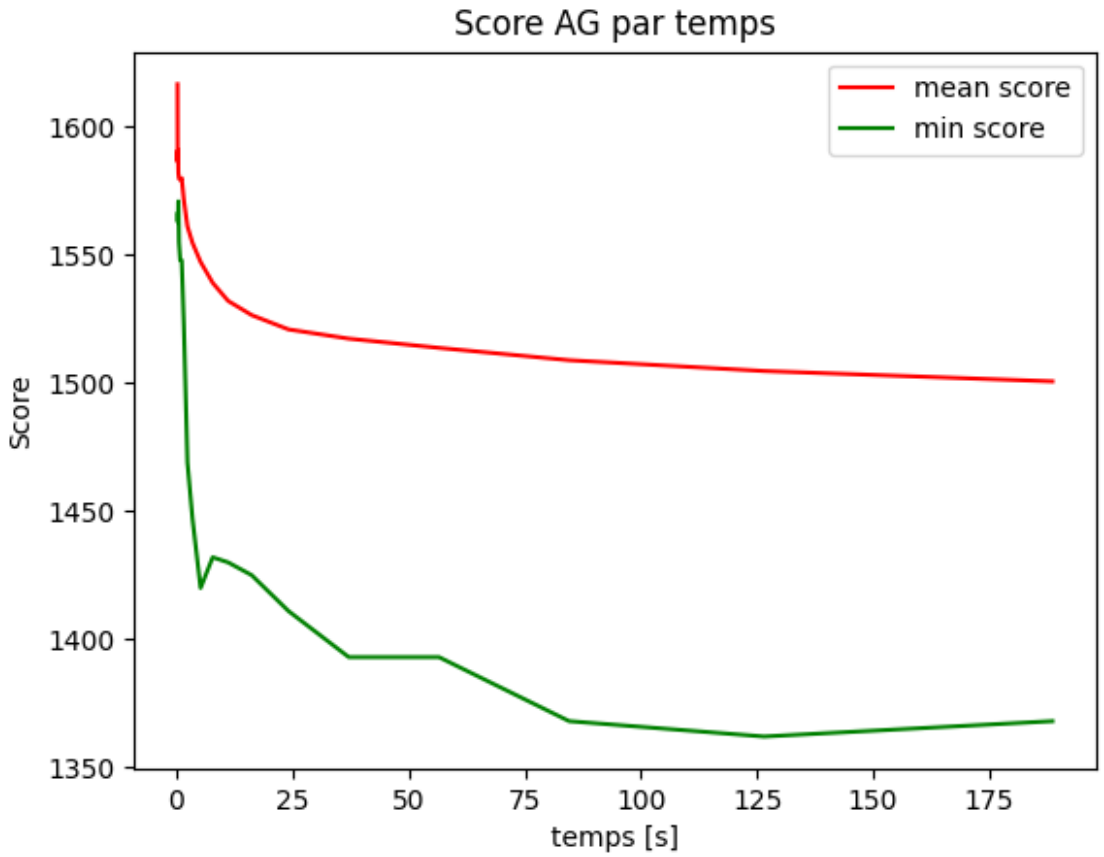
En premier temps on utilise comme **P_mut = 30%, P_cross = 30%**





Une deuxième simulation avec **P_mut = 60%, P_cross = 60%**





Après simulation de 20 itérations, on observe que les scores des dernières populations ont diminué en moyenne. Et il y a une diminution pour l'individu qui comporte la solution avec le score minimal. Avec une probabilité plus grande de mutation, nous observons une chute plus grande dans le minimum du score qu'il trouve.

V - Comparaisons intra et inter groupes

Pour un exemple similaire, avec une cinquantaine de clients, une capacité des camions de 100, une demande des clients de 3 (dans les algorithmes Tabou et Recuit Simulé) et une demande aléatoire des clients dans la méthode AG, et un coût de pénalité fixe de 5 par ajout de camion.

	Méthode Tabou	Recuit Simulé	AG
Coût optimal atteint	$C = 432$	$C = 730$ **La simulation étant trop longue le coût est un coût optimal obtenu en temps "raisonnable"*	$C_{min} = 1380$ **La demande ayant été demandé aléatoirement la moyenne des demandes des clients est plus élevée que celle des deux méthodes précédentes : 3 Il est donc normal et prévisible d'obtenir un coût bien plus élevé*
Nombre d'itérations	$Nb \approx 150$	$Nb \approx 80$ **pour un C de 730 (nombre d'itération moins élevées pour la méthode Tabou dans les mêmes conditions)*	$Nb \approx 17$
Temps de simulation	$t = 50/60\ sec$	$t = 40\ sec$ **pour un C de 730 (temps moins élevé pour la méthode Tabou dans les mêmes conditions)*	$t = 150\ sec$

Bien qu'il est difficile de comparer les deux premières méthodes et la troisième puisque le nombre total de commandes n'est pas le même (et donc le coût de la méthode 3 sera *de facto* nécessairement plus élevé), après cette première approche on peut tout de même arriver au constat que les méthodes Tabou et AG semble plus efficace en temps et en nombre d'itérations que la méthode du Recuit Simulé.