POLYTECH LYON

Capacitated Vehicle Routing Problem

MÉTHODES À BASE DE VOISINAGE

OPTIMISATION DISCRÈTE

BENALI Myriam, NAAJI Dorian
POLYTECH LYON
4A INFO GROUPE 2

POLYTECH LYON

Table des matières

1. INTRODUCTION

Le Capacited Vehicle Routing Problem est un problème de recherche opérationnelle et d'optimisation combinatoire. L'objectif de ce TP est de déterminer un ensemble d'itinéraires, commençant et se terminant à un dépôt et couvrant un ensemble de clients. Le but est de **livrer les marchandises aux clients en minimisant la distance totale de parcours**. Un véhicule dessert un seul itinéraire et ne doit pas disposer de plus de 100 marchandises à livrer.

Notre objectif est alors de **réduire au minimum la distance totale parcourue** par les véhicules en utilisant la méthode du **recuit simul**é et la méthode **Tabou** : 2 métaheuristiques à base de voisinage.

2. ENVIRONNEMENT

2.1. Langage et technologies

Le langage utilisé pour développer le projet était le **Java**. Nous avons utilisé **IntelliJ**, un environnement de développement pratique et ergonomique. Nous avons développé en binôme et utilisions tous les deux des ordinateurs sous Windows. Pour le partage du code, **GitHub**, un gestionnaire de version, a été grandement efficace et nous a permis de se partager le code et nos versions rapidement. L'ensemble du code est par ailleurs disponible au sein du dépôt suivant : https://github.com/DorianNaaji/cvrp-voisinage. Des **instructions pas à pas pour l'import du projet sur un poste y sont disponibles**.

Pour pouvoir tester le bon fonctionnement de certains de nos algorithmes, nous avons utilisé **JUnit5**, qui nous a permis de créer des classes et méthodes de tests unitaires. Cela nous a permis par exemple d'éviter les régressions en cas d'ajout de nouvelles fonctionnalités, ou tout simplement de vérifier le comportement de certains algorithmes. Pour importer **JUnit5**, nous avons utilisé un projet **Maven** à base de dépendances.

2.2. Architecture du projet

Le projet est constitué de 6 packages Java principaux, chacun constitué de classe(s) :

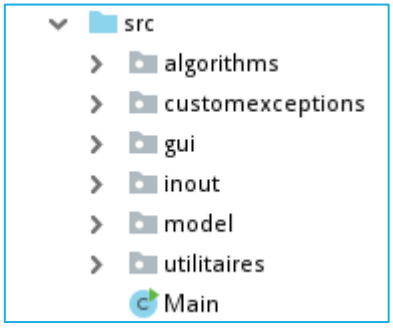


Figure 1 : Architecture globale

- Le package *model* contient l'ensemble des classes nécessaires pour la modélisation du problème : modélisation d'un Itinéraire, d'une Solution, etc.
- Le package *inout* permet le chargement de données externes.
- Le package *gui* contient les éléments liés à l'interface graphique, permettant d'ouvrir, fermer des fenêtres et de dessiner des itinéraires, solutions, clients, etc.
- Le package *customexceptions* contient des exceptions personnalisées qui traitent les cas de fonctionnement non attendus par notre programme.
- Le package *algorithms* contient l'implémentation des métaheuristiques tabou et recuit simulé, mais aussi des différentes méthodes de transformation (2-opt, échange, etc. sur lesquelles nous reviendrons par la suite).
- Finalement, le package *utilitaires* continent une unique classe utilitaire, permettant par exemple de calculer la distance euclidienne entre deux points (x1, y1) et (x2, y2) d'un plan.

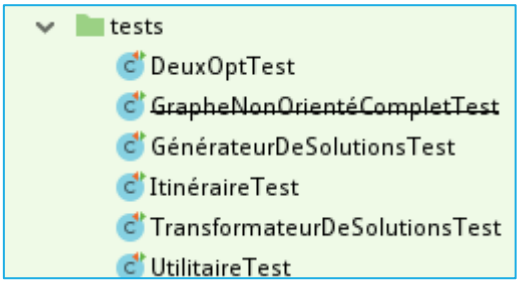


Figure 2 : Tests unitaires

Nous disposons également d'un dossier contenant l'ensemble de nos tests unitaires.

2.3. Interface graphique

Nous avons décidé de consacrer une partie de notre temps à l'implémentation d'une interface graphique, que nous avons jugée essentielle pour l'analyse de nos résultats. Celle-ci se situe dans le package *gui*.

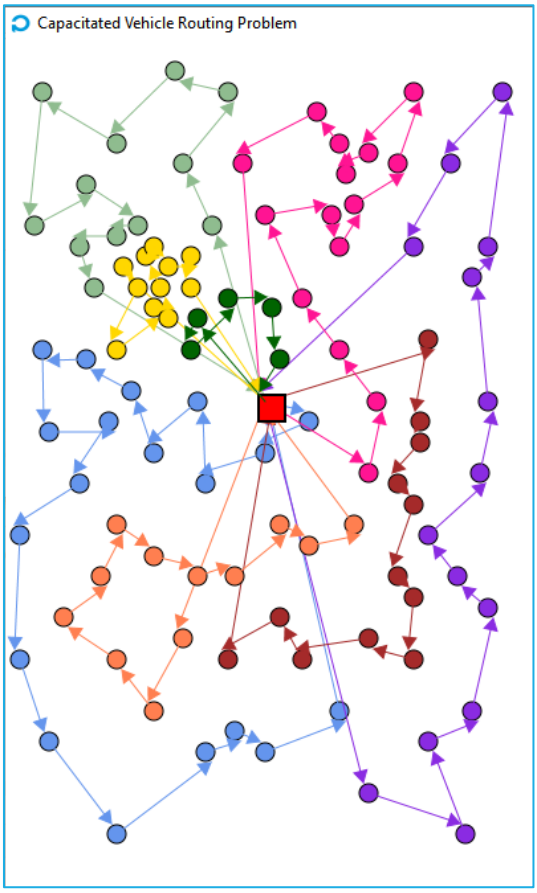


Figure 3 : Interface graphique

2.4. Tests unitaires

Nous avons créé un dossier dédié aux tests. En effet, nous avons réalisé des tests unitaires afin de tester le bon fonctionnement des parties importantes de notre programme. Ces tests nous permettent ainsi d'avoir une sécurité, si une partie du code ne fonctionne plus, nous nous en apercevons tout de suite.

Des tests ont été réalisés sur :

- la génération d'itinéraires respectant les contraintes métier,
- la génération de solutions,
- la transformation de solutions (transformation d'échange, insertion décalage, inversion et transformation 2-opt),
- les méthodes utilitaires (notamment le calcul de distance euclidienne entre deux éléments).

3. RÉSOLUTION DU PROBLÈME CVRP

3.1. Génération de solutions

3.1.1. Aléatoire classique

Nous avons tout d'abord implémenté une méthode permettant la génération d'une solution **aléatoire** à partir d'un fichier d'entrée. Pour cela, nous créons des itinéraires contenant des clients choisis aléatoirement. Chaque itinéraire ne dépasse pas une capacité maximum de livraison de 100 marchandises. Il est également possible de générer X de ces solutions aléatoires, X étant défini par l'utilisateur.

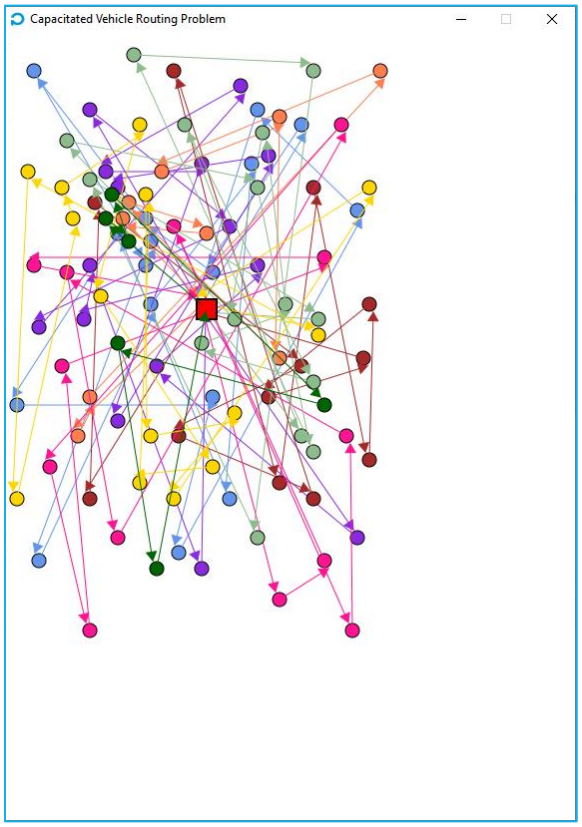


Figure 5 : Génération aléatoire sur le fichier "r101.txt"

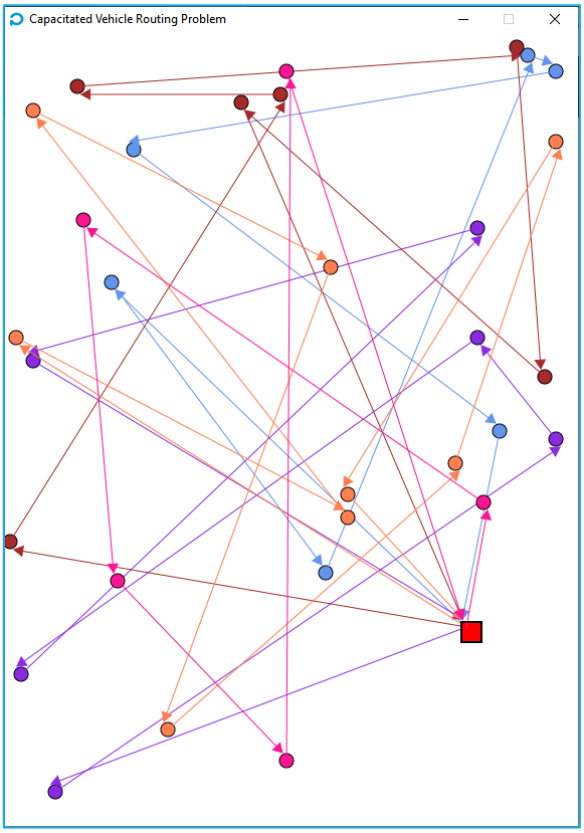


Figure 4 : Génération aléatoire sur le fichier "A3205.txt"

Un autre type de génération **aléatoire avec seuil** implémentée au sein du programme permet par exemple d'empêcher un itinéraire de dépasser une quantité Q de marchandises à livrer.

3.1.2. Aléatoire unique

De plus, nous avons implémenté une méthode permettant de générer aléatoirement **un itinéraire unique par fichier**. Cela permet donc d'obtenir une unique solution aléatoire, constituée d'un unique itinéraire. Cette solution devra ensuite être optimisée. Nous le détaillerons par la suite, mais cette méthode de génération couplée aux méthodes Tabou et Recuit simulé permet d'obtenir de très bons résultats.

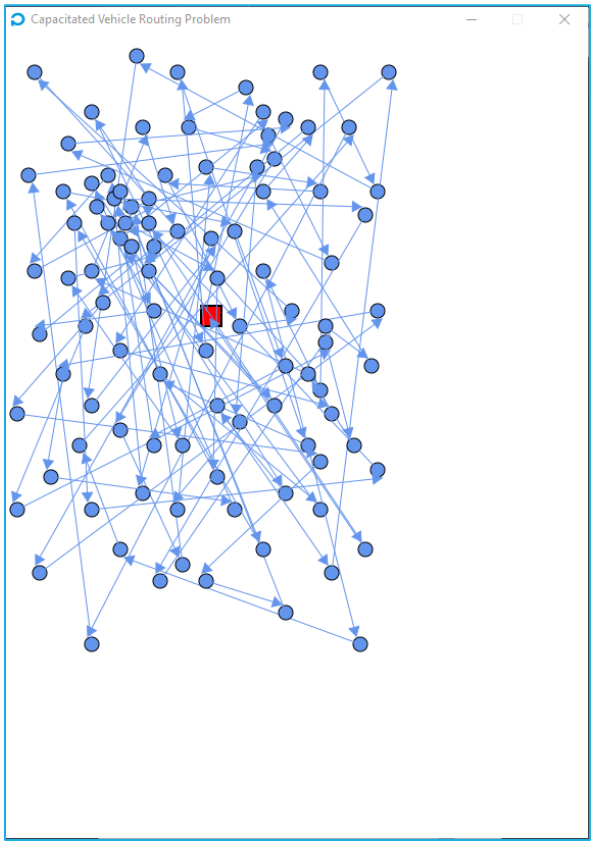


Figure 7 : Génération aléatoire sur le fichier "r101.txt"

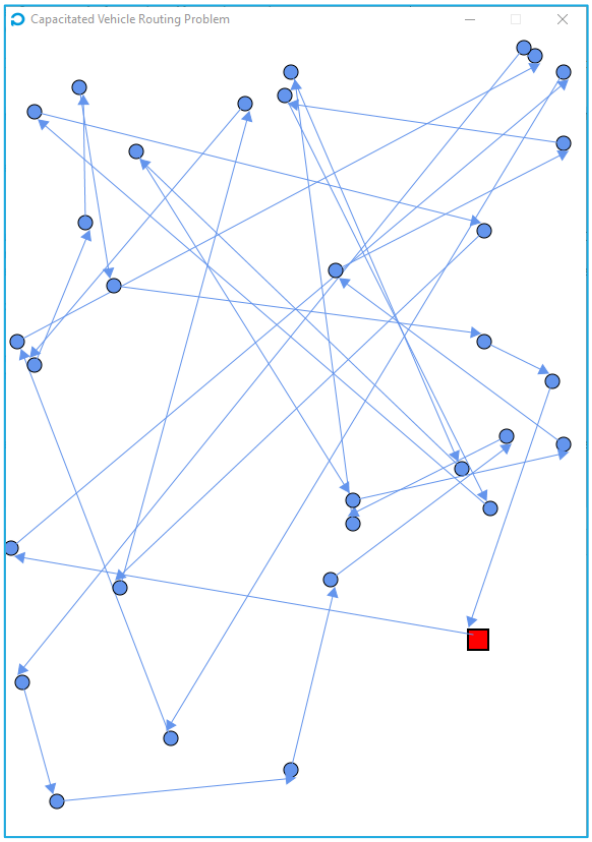


Figure 6 : Génération aléatoire unique sur le fichier "A3205.txt"

3.1.3. Génération proche en proche

Cette méthode de génération de solution gloutonne consiste **à prendre un client aléatoirement**, le placer dans un itinéraire, **puis de prendre tous les clients le plus proche de ce dernier et les placer au sein du même itinéraire**. Ce jusqu'à avoir rempli au maximum l'itinéraire concerné. On réitère jusqu'à avoir placé tous les clients dans des itinéraires. La solution ainsi générée est constituée de l'ensemble des itinéraires créés.

Cette méthode n'a pas vraiment été utilisée pour les tests que nous détaillerons au sein de ce rapport puisqu'elle consiste déjà à optimiser les solutions dès leur génération, avec une méthode gloutonne. En revanche, elle permet d'obtenir d'excellents résultats, peu importe la métaheuristique utilisée.

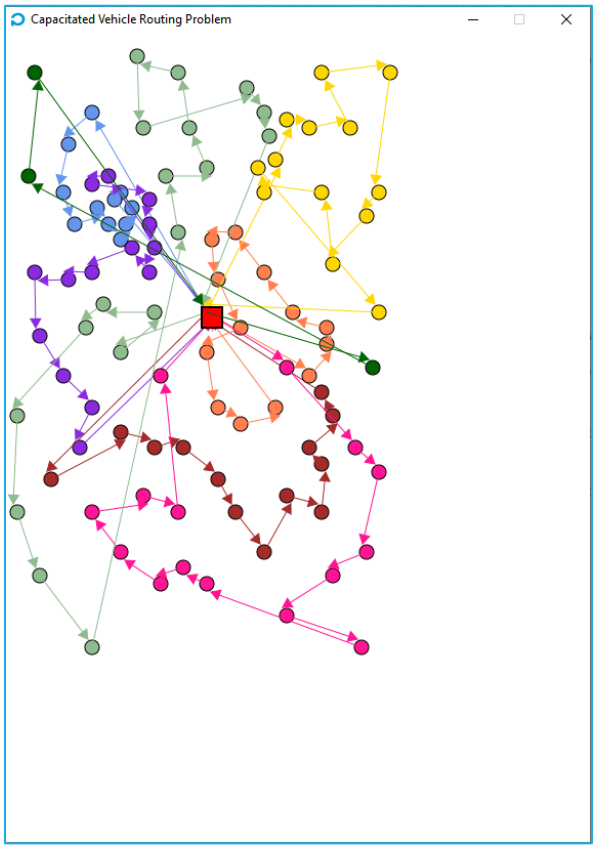


Figure 9 : Génération proche en proche sur le fichier "r101.txt"

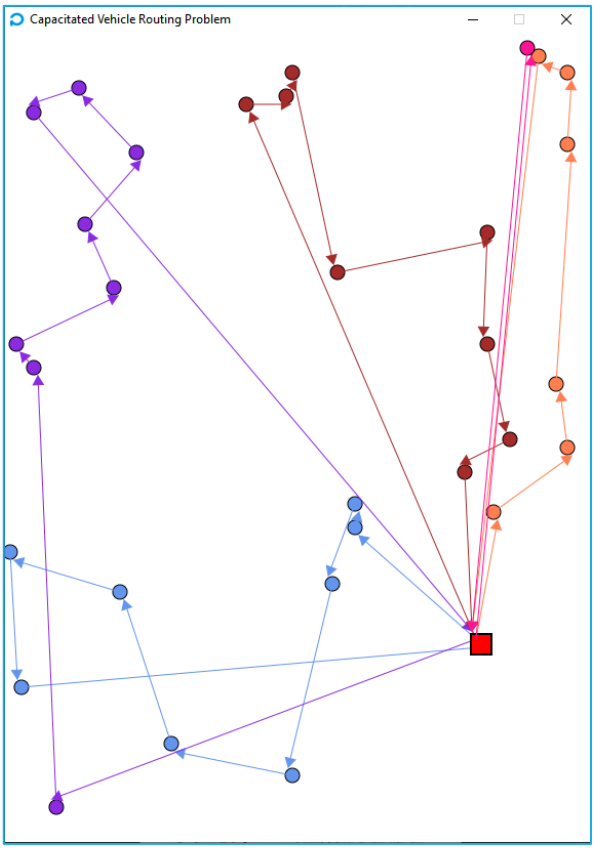


Figure 8 : Génération proche en proche sur le fichier "A3205.txt"

On remarque que dès la génération et avant même l'optimisation, les solutions sont déjà très convaincantes. Il sera donc plus pertinent d'appliquer des métaheuristiques de recherche des meilleures solutions (qui est l'enjeu principal du TP) grâce aux autres méthodes de génération de solutions citées précédemment.

3.2. Transformation de solutions

Nous avons implémenté quatre méthodes de transformation locale d'une solution :

- la transformation d'échange : elle permet l'échange de place entre deux clients choisis arbitrairement parmi un itinéraire. Sur notre exemple ci-dessous, les clients 1 et 4 ont été échangés.

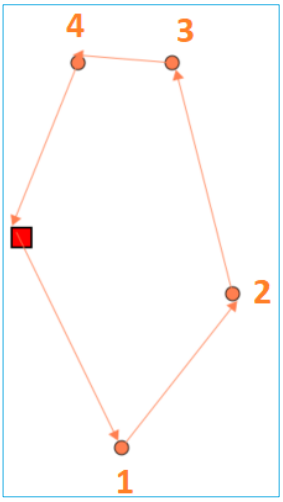


Figure 10 : Itinéraire avant la transformation d'échange

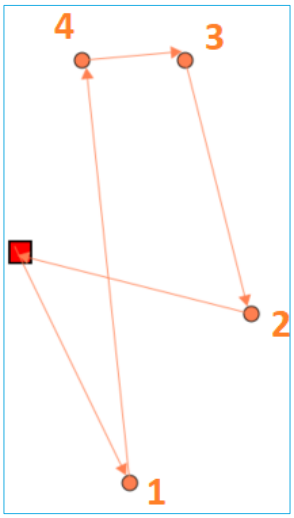


Figure 11 : Itinéraire après la transformation d'échange

- l'insertion décalage : elle permet d'effectuer une opération d'insertion décalage d'un client sur un itinéraire. Sur notre exemple ci-dessous, le client 4 s'est inséré avant le client 1 et tous les autres clients se sont décalés.

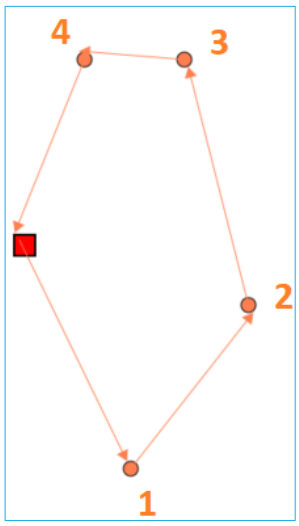


Figure 13 : Itinéraire avant l'insertion décalage

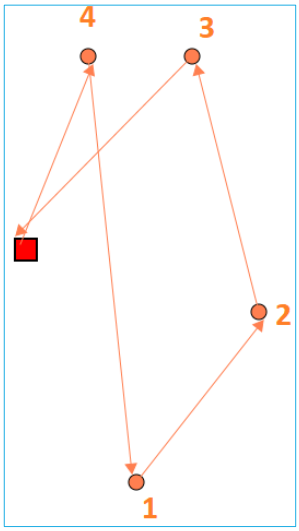


Figure 12 : Itinéraire après l'insertion décalage

- l'inversion : elle permet d'inverser une portion entière de k éléments appartenant à un itinéraire. Sur notre exemple ci-dessous, les clients 1, 2 et 3 ont été inversés.

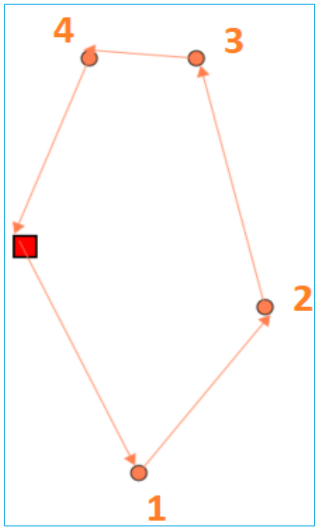


Figure 15 : Itinéraire avant l'inversion

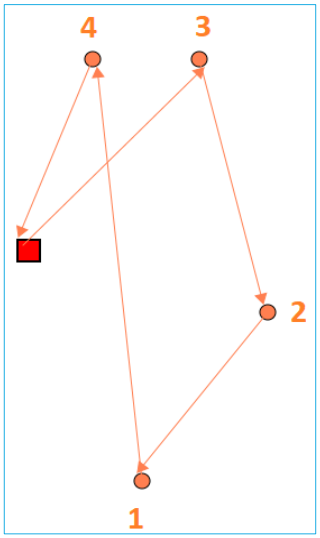


Figure 14 : Itinéraire après l'inversion

- la transformation 2-opt : elle permet d'effectuer d'échanger deux arêtes disjointes d'un itinéraire :

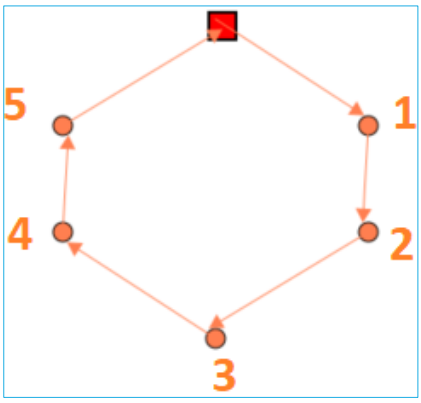


Figure 17 : Itinéraire avant la transformation 2-opt

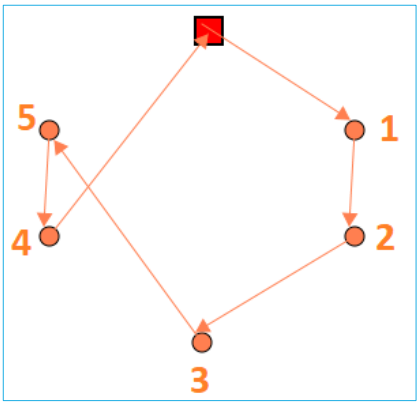


Figure 16 : Itinéraire après la transformation 2-opt

De plus, nous avons également implémenté **la transformation d'échange entre deux itinéraires d'une solution** afin **d'échanger des clients entre les itinéraires**. C'est ce que l'on a appelé au sein du code une **« méta transformation »**, afin de réaliser de **l'optimisation inter-itinéraires**. En raison des contraintes de temps, aucune autre méta transformation n'a été implémentée.

Pour prolonger notre travail sur les transformations, il aurait été intéressant de **réaliser des transformations locales « intelligentes » et non aléatoires**, en prenant **compte de certaines contraintes** comme la distance entre deux clients. Également pour nos méta transformations, de par exemple échanger de place un client trop excentré d'un itinéraire et de le placer dans un autre itinéraire, dans lequel il sera plus proche des autres clients.

3.3. Optimisation de solutions

3.3.1. Méthode du recuit simulé

La méthode du recuit simulé est une métaheuristique mise au point par trois chercheurs, S. Kirkpatrickn C.D. Gelatt et M.P. Vecchi en 1983 et indépendamment par V Černy en 1985.

Cette méthode est inspirée de pratiques issues de la thermodynamique, de la façon dont les métaux sont chauffés puis refroidis.

En optimisation combinatoire, le recuit simulé consiste en la recherche d'une configuration pour minimiser la fitness (fonction objectif) donnée. Son objectif est de sortir des minima locaux en acceptant des solutions possédant une fitness moins bonne.

Dans le cas de ce projet, la fitness représente la distance totale de la solution. Nous cherchons donc à la minimiser.

A chaque pas de température, nous générons N solutions voisines dans le but de trouver la meilleure solution. La température est multipliée par un coefficient de refroidissement de la température lorsque les N solutions voisines de la température courante ont été générés.

La **température initiale**, le **nombre d'itérations pour lequel la température est constante** (nombre de solution voisines par pas de température) et le **coefficient de refroidissent** sont paramétrables.

3.3.2. Méthode Tabou

La méthode de recherche Tabou est une métaheuristique présentée par Fred W. Glover en 1986 [1].

Le principe de cette méthode est de partir d'une solution initiale, d'explorer le voisinage de cette solution (c'est-à-dire des solutions qui lui ressemblent, que l'on peut créer en transformant la solution initiale) pour trouver celles qui minimisent une *fitness* (fonction objectif) donnée.

Dans notre cas, la *fitness* correspond à la longueur totale parcourue pour la solution. C'est la fonction objectif à minimiser. La recherche Tabou doit donc se charger de cela.

Pour chaque élément du voisinage, la méthode tabou recherche la meilleure solution voisine, puis garde cette solution en mémoire. Elle réitère ensuite en recherchant les nouveaux voisins de cette meilleure solution voisine. Le **nombre de voisins** devant être explorés à chaque itération est paramétrable.

Cependant, à chaque itération, on ajoute l'ensemble des voisins trouvés dans une liste dite « de tabou », qui contient l'ensemble des éléments que l'on **s'interdit** de reparcourir. Ainsi, on ne revient pas en arrière. Cette liste de tabou, paramétrable, dispose d'une **taille fixe**. Avec une logique FIFO, une fois que cette liste est pleine, on retire les plus anciens voisins (donc les plus anciennes transformations/solutions).

On continue ce processus jusqu'à une condition de sortie, en général définie par **un nombre d'itérations maximales**, paramétré au préalable.

4. DISCUSSIONS DES RÉSULTATS

Nous avons décidé d'exposer nos résultats sur le jeu de données **A5509**, constitué de **54 clients.** Pour chaque méthode et pour chaque type de transformation, nous faisons varier les paramètres afin de mesurer l'influence des paramètres pour chaque méthode.

Nos résultats ont été obtenus grâce **au type de génération aléatoire unique** (une solution contenant un seul itinéraire unique).

Cet itinéraire unique est ensuite optimisé à l'aide du recuit simulé ou de la méthode tabou, puis l'itinéraire est **divisé en plusieurs itinéraires respectant les consignes métiers** (un itinéraire ne doit pas contenir plus de 100 marchandises à livrer). *Éventuellement, (mais ce n'est pas réalisé pour ces tests), il est possible dans le programme d'appliquer un tabou ou un recuit une nouvelle fois sur cette solution dont l'itinéraire unique a été découpé en plusieurs itinéraires.*

Pour chaque méthode, nous avons recueilli de nombreuses données. Pour chaque type de transformation (transformation d'échange, insertion décalage, inversion, 2-opt) nous avons 9 tests par méthode. En effet, pour chacune des méthodes, nous avons 3 paramètres pouvant changer : **nous avons alors fait varier chaque paramètre en fixant les autres afin de pouvoir analyser l'influence de chaque paramètre.**

Les tests ont été réalisés sur une solution aléatoire et sur 50 solutions aléatoires. Lorsque l'on réalise des tests sur une unique solution aléatoire, on applique l'algorithme de recuit simulé ou de tabou sur la solution unique, puis on récupère les résultats. Lorsque l'on réalise cela sur 50 solutions aléatoires, on boucle sur les 50 solutions aléatoire et l'algorithme de recuit simulé ou de tabou est exécuté 50 fois, une fois par solution. On récupère ensuite la meilleure fitness parmi toutes ces solutions aléatoires et toutes ces exécutions.

4.1. Méthode du recuit simulé

4.1.1. Données brutes

L'ensemble des tests qui suivent on été réalisés avec un ordinateur doté d'un processeur Intel Core i7-8550U 1.80GHz – 1.99 GHz et 8,00Go de mémoire RAM.

4.1.1.1. Transformation d'échange :

Température initiale	Nombre de voisins par température	Coefficient de diminution de la température	Temps d'exécution (sur une solution)	Temps d'exécution (sur 50 solutions)	Distance totale parcourue (sur une solution)	Distance totale parcourue (sur 50 solutions)
1	500	0,95	0,051 s	0,068 s	3116,80	2704,11
10	500	0,95	0,22 s	5,03 s	1505,96	1371,65
30	500	0,95	0,317 s	7,3 s	1941,87	1732,60
10	100	0,95	0,124 s	1,282 s	1554,30	1421,58
10	300	0,95	0,167 s	3,259 s	1612,99	1350,23
10	800	0,95	0,282 s	8,387 s	1479,25	1308,60
10	500	0,5	0,098 s	0,898 s	1905,30	1516,53
10	500	0,8	0,179 s	1,574 s	1612,67	1381,04
10	500	0,99	0,737	23,9 s	1356,30	1298,52

4.1.1.2. Transformation d'insertion décalage :

Température initiale	Nombre de voisins par température	Coefficient de diminution de la température	Temps d'exécution (sur une solution)	Temps d'exécution (sur 50 solutions)	Distance totale parcourue (sur une solution)	Distance totale parcourue (sur 50 solutions)
1	500	0,95	0,045 s	0,086 s	3245,87	2829,40
10	500	0,95	0,27 s	6,816 s	1370,21	1280,21
30	500	0,95	0,386 s	10,15 s	2016,09	1742,06
10	100	0,95	0,18 s	1,793 s	1522,05	1418,03
10	300	0,95	0,194 s	4,337 s	1550,85	1323,48
10	800	0,95	0,384 s	12,04 s	1326,31	1230,24
10	500	0,5	0,111 s	0,943 s	1531,35	1449,90
10	500	0,8	0,141 s	2,663 s	1434,60	1357,54
10	500	0,99	0,996 s	35,72 s	1370,74	1259,14

4.1.1.3. Transformation d'inversion :

Température initiale	Nombre de voisins par température	Coefficient de diminution de la température	Temps d'exécution (sur une solution)	Temps d'exécution (sur 50 solutions)	Distance totale parcourue (sur une solution)	Distance totale parcourue (sur 50 solutions)
1	500	0,95	0,041 s	0,077 s	3122,68	2760,14
10	500	0,95	0,316 s	7,778 s	1484,35	1319,11
30	500	0,95	0,429 s	11,27 s	1882,79	1792,02
10	100	0,95	0,146 s	2,12 s	1541,78	1343,34
10	300	0,95	0,227 s	4,788 s	1502,40	1326,15
10	800	0,95	0,481 s	11,928 s	1339,11	1289,45
10	500	0,5	0,135 s	1,04 s	1631,01	1398,83
10	500	0,8	0,162 s	2,289 s	1530,67	1329,64
10	500	0,99	1,058 s	37,70 s	1312,73	1309,27

4.1.1.4. Transformation 2-opt :

Température initiale	Nombre de voisins par température	Coefficient de diminution de la température	Temps d'exécution (sur une solution)	Temps d'exécution (sur 50 solutions)	Distance totale parcourue (sur une solution)	Distance totale parcourue (sur 50 solutions)
1	500	0,95	0,044 s	0,103 s	3295,80	2774,35
10	500	0,95	0,338 s	8,575 s	1440,77	1312,47
30	500	0,95	0,435 s	12,566 s	1954,27	1792,60
10	100	0,95	0,191 s	2,051 s	1423,58	1379,61
10	300	0,95	0,258 s	5,414 s	1379,79	1307,38
10	800	0,95	0,469 s	14,206 s	1417,12	1278,93
10	500	0,5	0,171 s	1,174 s	1568,62	1411,43
10	500	0,8	0,173 s	2,62 s	1487,57	1337,12
10	500	0,99	1,179 s	42,544 s	1320,21	1273,70

4.1.2. Influence des paramètres

Le choix des paramètres est très important pour le bon fonctionnement de l'algorithme. De plus, nous remarquons qu'il n'y a pas de différence notable entre les différents types de transformation pour la méthode du recuit simulé.

4.1.2.1. Influence de la température initiale

Pour analyser l'influence de la température initiale, il faut regarder les 3 premières lignes de chaque tableau. Nous avons réalisé 3 tests pour chaque transformation : une température initiale à T = 1, puis T = 10 et enfin T = 30. La température est un critère permettant de contrôler l'acceptation des solutions moins bonnes. Si la **température est grande**, les **solutions moins bonnes ont une plus grande probabilité d'être acceptées**. Si la température initiale est trop grande, toutes les solutions voisines sont acceptées : le début de la recherche ne sert donc à rien. A l'inverse, si elle est égale à 0, les solutions moins bonnes ne sont jamais acceptées. Ces paramètres extrêmes nous permettent d'avoir de mauvais résultats.

Grâce à nos cas de tests, nous pouvons voir que la **température initiale, doit être suffisamment élevée pour accepter plus de moins bonnes solutions mais elle ne doit ni être trop basse, ni trop élevée**. Parmi nos cas de tests, la température T = 10 nous donne de meilleurs résultats. *(Sur des jeux de données avec plus de clients, une température initiale encore plus basse donne des meilleurs résultats (voir partie 4.1.3. Discussions des résultats).*

De plus, nous remarquons également que le **temps d'exécution de l'algorithme augmente dans le même sens que l'augmentation de la température initiale**. En effet, plus la température initiale est élevée plus d'itérations seront faites sur notre algorithme et donc plus le temps d'exécution sera élevé.

4.1.2.2. Influence du nombre d'itérations

Pour analyser l'influence du nombre d'itérations pour lequel la température est constante (nombre de solutions à chaque pas de température), il faut regarder les lignes 4 à 6 de chaque tableau. Nous avons réalisé 3 tests pour chaque transformation : nIter = 100, nIter = 300 et nIter = 800. Ce paramètre est à déterminer en fonction du temps que nous avons pour effectuer les tests. **Plus ce paramètre est important, plus l'algorithme va créer de nouvelles solutions voisines et va donc permettre d'augmenter la fitness**. Cependant, **au bout d'une certaine valeur, l'augmentation du nombre de solutions voisines ne permet plus d'augmenter la valeur de la fitness**. Parmi nos cas de tests, nIter = 800 nous donne de meilleurs résultats, néanmoins le temps d'exécution de l'algorithme est plus élevé avec cette valeur. *(Sur des jeux de données avec plus de clients, un nombre d'itération encore plus bas donne des meilleurs résultats (voir partie 4.1.3. Discussions des résultats).*

Il est évident que **plus le nombre de solutions voisines que l'on génère par température est élevée, plus le temps d'exécution de l'algorithme est élevé**.

4.1.2.3. Influence du coefficient de refroidissement

Pour analyser l'influence du coefficient de refroidissement, il faut regarder les lignes 7 à 9 de chaque tableau. Nous avons réalisé 3 tests pour chaque transformation : coeff = 0,5, coeff = 0,8 et coeff = 0,99. Ce paramètre **spécifie l'évolution de la température**, température qui décroit par palier, et doit avoir une valeur entre 0 et 1. **Plus ce coefficient est petit, plus la température diminue rapidement et inversement, plus il est grand, plus la température diminue moins rapidemen**t. Il est donc **essentiel d'avoir un coefficient de refroidissement proche de 1 afin de trouver une meilleure fitness**. Parmi nos cas de tests, coeff = 0,99 nous donne de meilleurs résultats, néanmoins le temps d'exécution de l'algorithme est plus élevé avec cette valeur.

Le **temps d'exécution de l'algorithme augmente dans le même sens que l'augmentation du coefficient de refroidissement**. On remarque qu'il augmente considérablement lorsque le coefficient est très proche de 1 (0,99 dans notre cas *(cf tableau)*).

4.1.3. Quelques résultats

Voici quelques résultats que nous avons obtenu avec la méthode du recuit simulé.

Résultat sur le jeu de données **A5509** (constitué de 54 clients) avec les paramètres suivants :

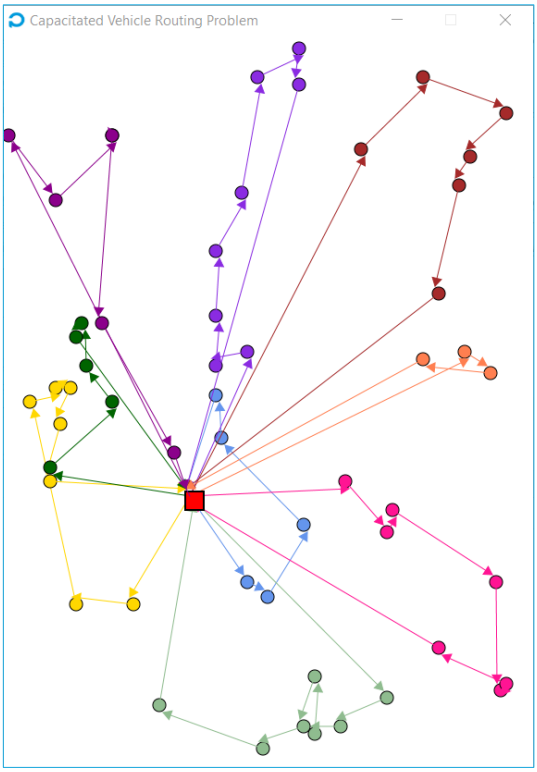


Figure 18 : Meilleure solution trouvée sur le fichier A5509 avec le recuit simulé

- Température initiale = 5
- Nombre d'itérations pour lequel la température est constante = 500
- Coefficient de refroidissement = 0,99
- Type de génération : aléatoire unique
- Type de transformation : 2-opt

Fitness = 1145,27

Résultat sur le jeu de données **R101** (le nouveau fichier ajouté sur claroline contenant 100 clients) :

On remarque que dans le cas de fichiers avec beaucoup de clients, nous obtenons des meilleurs résultats en diminuant la température initiale et le nombre d'itérations pour lequel la température est constante (par rapport aux tests précédent) :

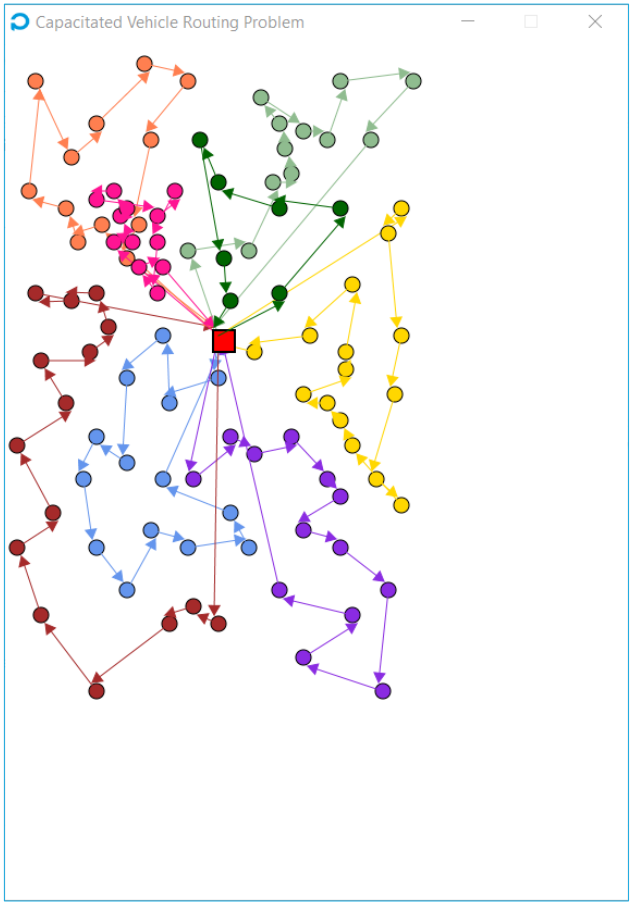


Figure 19 : Meilleure solution trouvée sur le fichier R101 avec le recuit simulé

- Température initiale = 3
- Nombre d'itérations pour lequel la température est constante = 300
- Coefficient de refroidissement = 0,99
- Type de génération : aléatoire unique
- Type de transformation : 2-opt

Fitness = 983,66

4.2. Méthode Tabou

4.2.1. Données brutes

L'ensemble des tests qui suivent ont été réalisés avec un ordinateur doté d'un processeur Intel Core i7-8550U 1.80GHz – 1.99 GHz et 8,00Go de mémoire RAM.

4.2.1.1. Transformation d'échange :

Taille maximale de la liste Tabou	Nombre d'itérations maximal de l'algorithme	Nombre de solutions voisines à chaque itération	Temps d'exécution (sur une solution)	Temps d'exécution (sur 50 solutions)	Meilleure distance totale parcourue (sur une solution)	Meilleure distance totale parcourue (sur 50 solutions)
50	500	25	0,761 s	24,2 s	1458,73	1334,62
150	500	25	0,796 s	17,6 s	1404,32	1347,75
500	500	25	0,724 s	17,7 s	1491,12	1295,63
200	100	25	0,374 s	3,99 s	1819,14	1503,36
200	700	25	0,868 s	22,8 s	1606,49	1217,50
200	1000	25	1,277 s	34,5 s	1553,43	1316,87
200	500	50	1,114 s	33,2 s	1431,85	1304,09
200	500	100	2,074 s	65,6 s	1358,48	1276,93
200	500	150	2,564 s	98,7 s	1466,98	1285,96

4.2.1.2. Transformation d'insertion décalage :

Taille maximale de la liste Tabou	Nombre d'itérations maximal de l'algorithme	Nombre de solutions voisines à chaque itération	Temps d'exécution (sur une solution)	Temps d'exécution (sur 50 solutions)	Meilleure distance totale parcourue (sur une solution)	Meilleure Distance totale parcourue (sur 50 solutions)
50	500	25	0,991 s	22,2 s	1294,09	1181,32
150	500	25	0,774 s	20,7 s	1270,84	1165,85
500	500	25	0,811 s	26,5 s	1239,52	1168,35
200	100	25	0,302 s	4,415 s	1523,77	1400,19
200	700	25	1,087 s	28,1 s	1388,26	1183,41
200	1000	25	1,431 s	43,1 s	1467,29	1205,82
200	500	50	1,329 s	38,3 s	1454,50	1234,02
200	500	100	2,292 s	82 s	1292,87	1196,88
200	500	150	2,678 s	115,5 s	1201,88	1188,50

4.2.1.3. Transformation d'inversion :

Taille maximale de la liste Tabou	Nombre d'itérations maximal de l'algorithme	Nombre de solutions voisines à chaque itération	Temps d'exécution (sur une solution)	Temps d'exécution (sur 50 solutions)	Meilleure distance totale parcourue (sur une solution)	Meilleure distance totale parcourue (sur 50 solutions)
50	500	25	0,845 s	21,6 s	1265,46	1166,20
150	500	25	0,782 s	20,7 s	1248,42	1161,17
500	500	25	0,9 s	31,2 s	1312,11	1201,84
200	100	25	0,426 s	4,851 s	1290,27	1215,71
200	700	25	0,979 s	28,8 s	1197,74	1161,53
200	1000	25	1,318 s	40,4 s	1256,54	1171,13
200	500	50	1,558 s	41,9 s	1227,52	1169,11
200	500	100	2,32 s	82,3 s	1293,80	1163,84
200	500	150	3,084 s	119,1 s	1181,81	1145,01

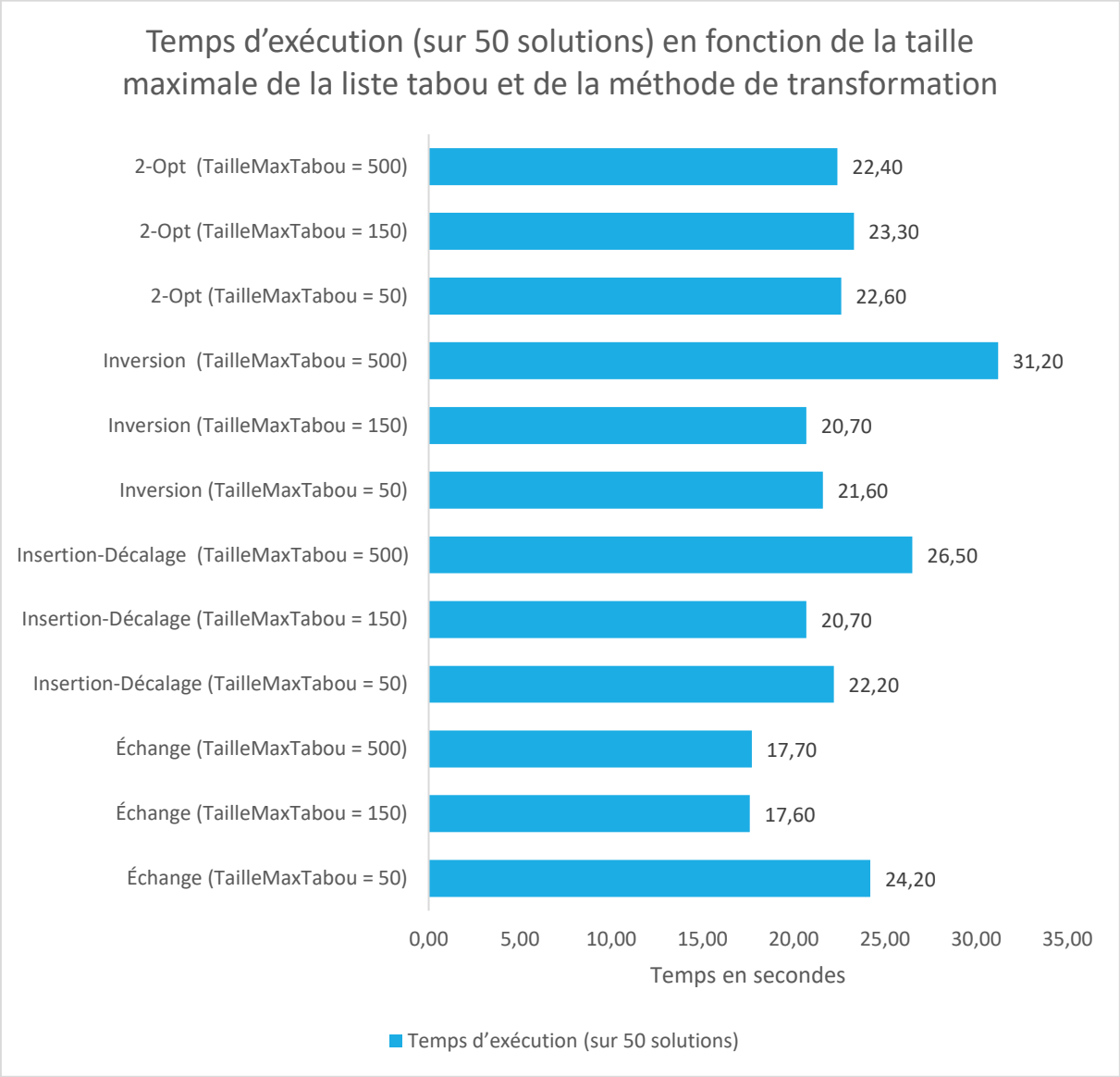
4.2.1.4. Transformation 2-opt :

Taille maximale de la liste Tabou	Nombre d'itérations maximal de l'algorithme	Nombre de solutions voisines à chaque itération	Temps d'exécution (sur une solution)	Temps d'exécution (sur 50 solutions)	Meilleure distance totale parcourue (sur une solution)	Distance totale parcourue (sur 50 solutions)
50	500	25	0,915 s	22,6 s	1237,87	1156, 49
150	500	25	0,876 s	23,3 s	1267,46	1180,55
500	500	25	0,964 s	22,4 s	1244,64	1178,73
200	100	25	0,335 s	5,199 s	1313,28	1243,10
200	700	25	1,139 s	31,1 s	1244,05	1164,92
200	1000	25	1,541 s	44 s	1142,78	1180,85
200	500	50	1,454 s	52,2 s	1190,99	1162,21
200	500	100	2,513 s	100,1 s	1264,90	1152,53
200	500	150	3,551	121,4 s	1259,74	1170,60

4.2.2. Influence des paramètres

4.2.2.1. Influence de la taille de liste tabou

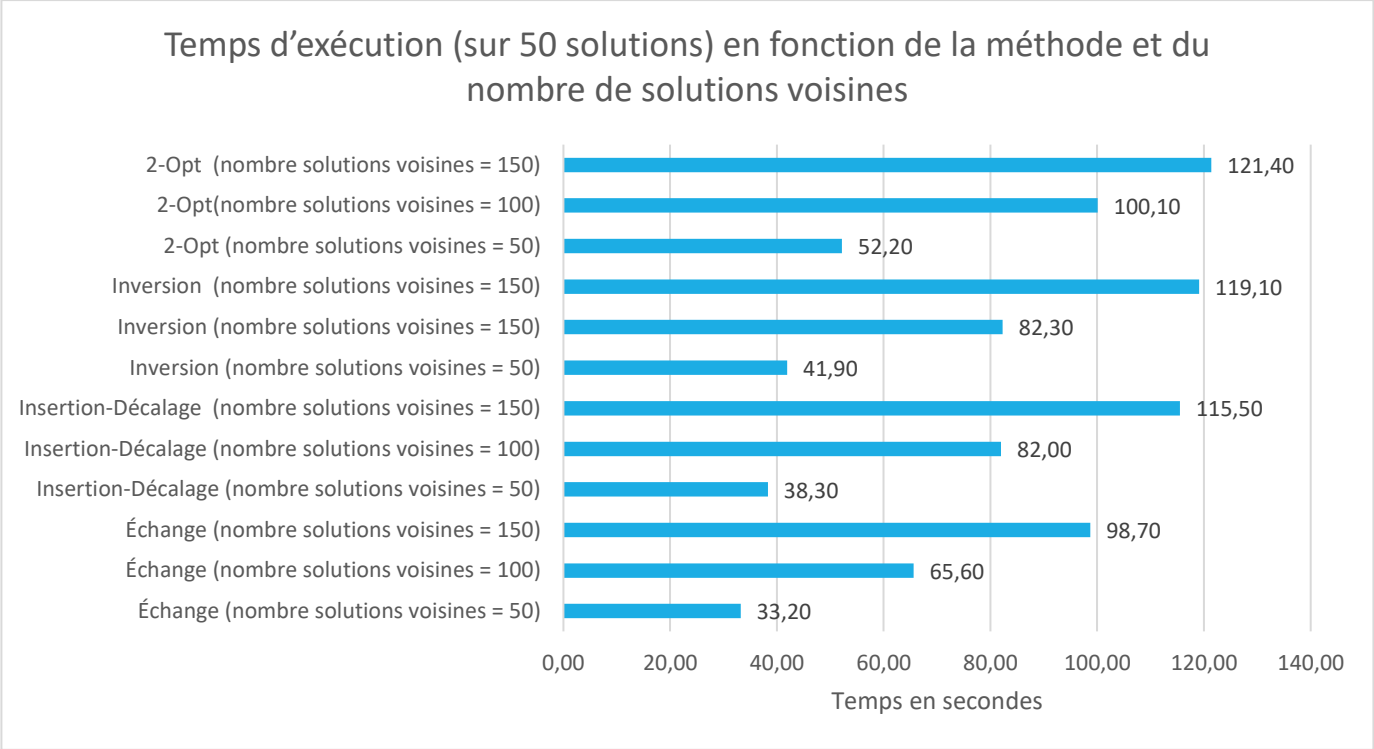
Grâce aux données précédentes, il est possible de créer des graphiques et d'interpréter l'influence des différents paramètres de l'algorithme Tabou.

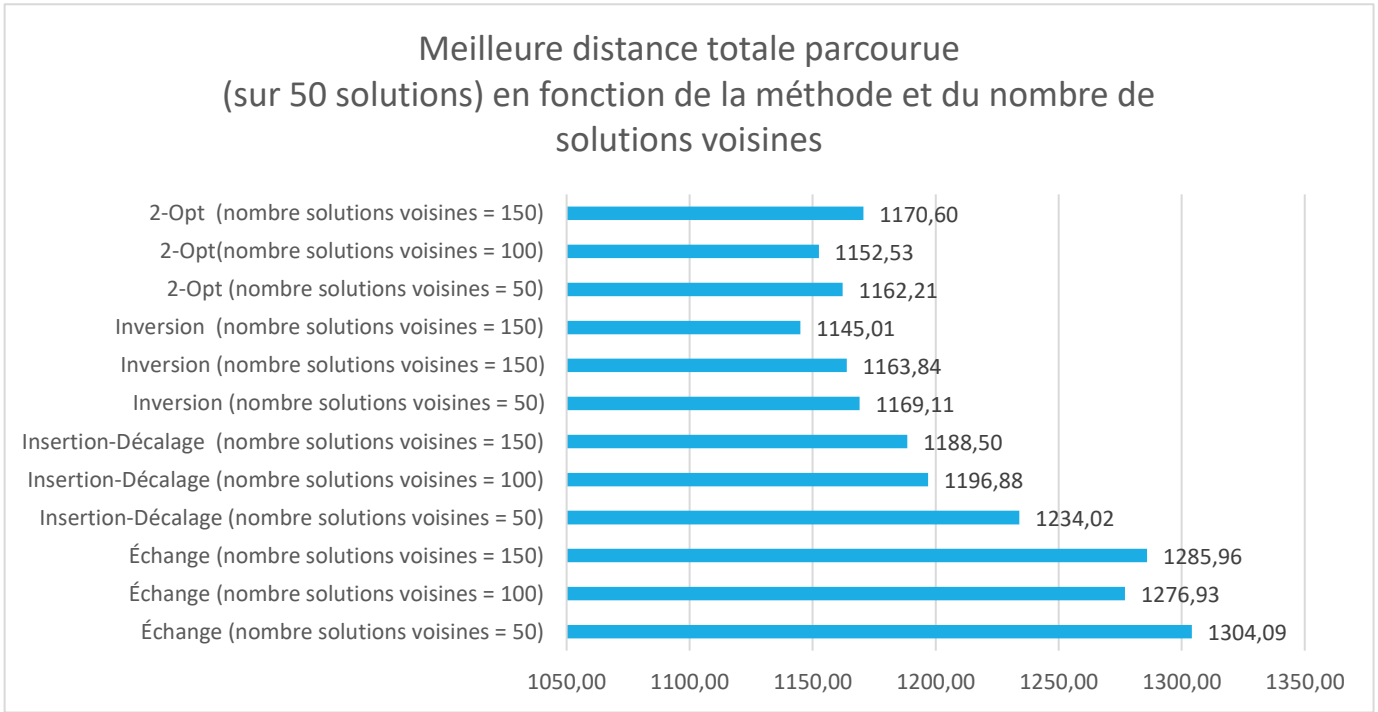


Sur l'insertion décalage et l'inversion, on remarque qu'augmenter la taille maximale de la liste de Tabou semble augmenter le temps d'exécution, mais dans l'ensemble, l'influence de ce paramètre est négligeable sur les temps d'exécution, peu importe la méthode employée. On a remarqué le même phénomène pour la fitness finale.

4.2.2.2. Influence du nombre de solutions voisines

En revanche, pour le nombre de solutions voisines, on voit tout de suite plus clairement un lien de cause à effet avec le temps d'exécution et l'optimisation de la fitness :





En augmentant le nombre de solutions voisines, on remarque une très nette augmentation du temps d'exécution, ce qui est normal puisque l'algorithme de Tabou va aller explorer un voisinage plus large, et donc plus de traitements sont réalisés. **2-Opt devient alors la transformation la plus longue** avec un voisinage de 150, contre **l'échange qui est la plus rapide** avec le même voisinage.

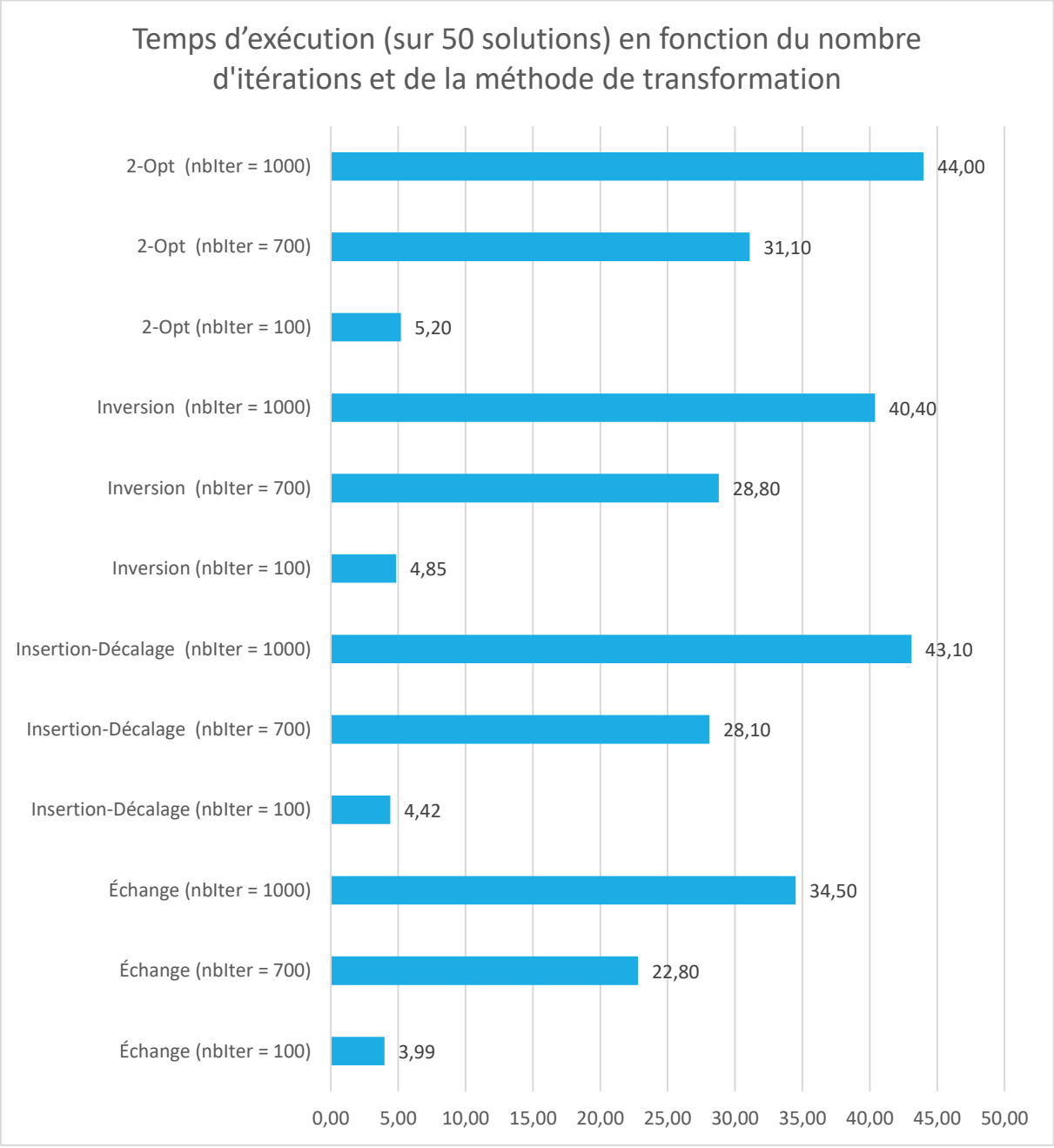
En revanche, 2-opt est de loin la transformation la plus efficace, et en augmentant la taille du voisinage de notre Tabou, la fitness semble être minimisée, mais les données ne permettent pas de le confirmer clairement. En pratique, nous avons néanmoins remarqué une amélioration à mesure que nous améliorons le voisinage. L'échange est la moins efficace d'après la figure précédente, et nous l'avons ressenti lors de nos tests.

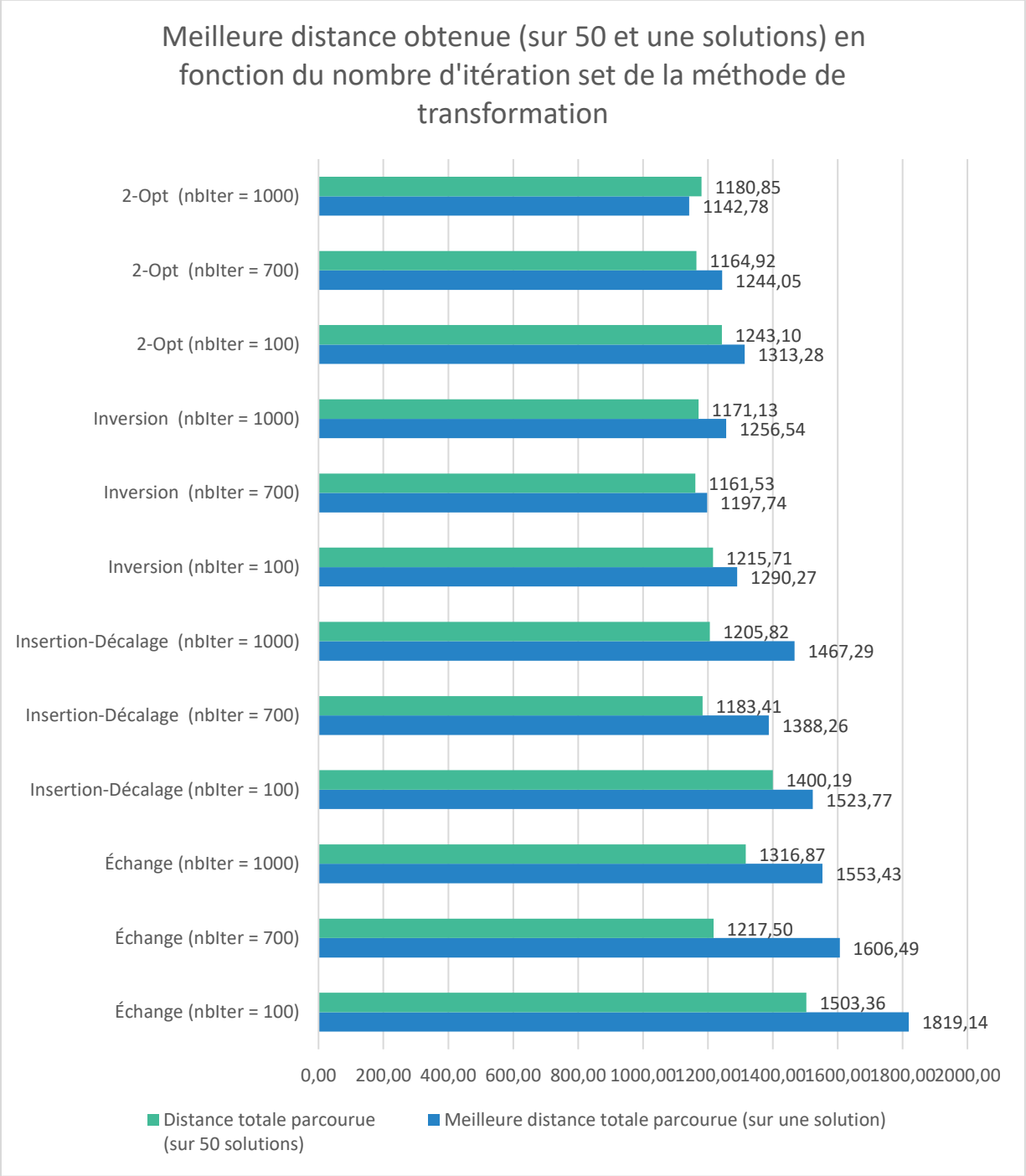
Nous avions tendance à établir une taille maximale de la liste tabou proportionnelle au nombre de voisins par itération. Par exemple, pour une taille maximale de Tabou de 400, nous paramétrions un nombre de voisins de 100 à 200. Ainsi, au bout de 4 ou 2 itérations, on s'autorise alors à vider proportionnellement la liste de tabou du nombre de voisins défini.

Dans l'ensemble, les **deux précédents paramètres doivent être choisis l'un par rapport à l'autre**. Si l'on souhaite optimiser au maximum la solution et que l'on a du temps devant nous, alors augmenter la taille du voisinage par itération est intéressant. Il faudra en revanche augmenter la taille maximale de la liste tabou en conséquence, afin de ne pas revenir en arrière.

Si l'on souhaite une exécution rapide, ces paramètres devront alors être réduit, mais cela se fera au détriment de la qualité de la fitness finale, qui risque d'être plus élevée et donc moins optimisée.

4.2.2.3. Influence du nombre d'itérations





Augmenter le nombre d'itérations augmente également de manière drastique le temps d'exécution du programme. Ce paramètre est donc à choisir avec la plus grande attention. 2-opt explose le score avec plus de 40 secondes de temps d'exécution pour 1000 itérations. Juste en dessous, on a l'inversion et l'insertion décalage.

Mais ce désavantage ne vient pas sans bénéfice ; en effet, en contrepartie d'un temps d'exécution plus long, on va avoir tendance à **obtenir des fitness plus faibles et donc des solutions optimisées**. Plus le nombre d'itérations augmente, plus on explore

le voisinage et donc plus on a de chance de trouver de meilleures solutions. Avec le 2-opt, pour 700 et 1000 itérations, on remarque en effet que **la fitness a été efficacement réduite**, bien que le temps d'exécution soit un peu long. Inversion et Insertion-décalage permettent également d'obtenir des résultats très satisfaisants, mais 2-opt reste devant.

En pratique, nous avons également remarqué qu'avec tabou, 2**-opt était très efficace rapidement**, contrairement aux autres transformations pour lesquelles il fallait itérer un peu plus pour avoir de bons résultats.

En général, pour nos besoins, nous fixions le nombre d'itérations entre 500 et 1000. Cela permettait d'obtenir un temps d'exécution raisonnable tout en obtenant des résultats relativement raisonnables. Pour des exécutions très rapides, nous fixions en revanche un nombre d'itérations entre 200 et 400, assurant ainsi des tests très rapides mais la fitness de sortie n'était pas toujours des plus élevées.

4.2.3. Quelques résultats

Concernant le fichier sur lequel les tests ont été réalisés **(A5509, constitué de 54 clients)**, la meilleure fitness trouvée avec Tabou est de **1142.8397**, avec notamment pour paramètres :

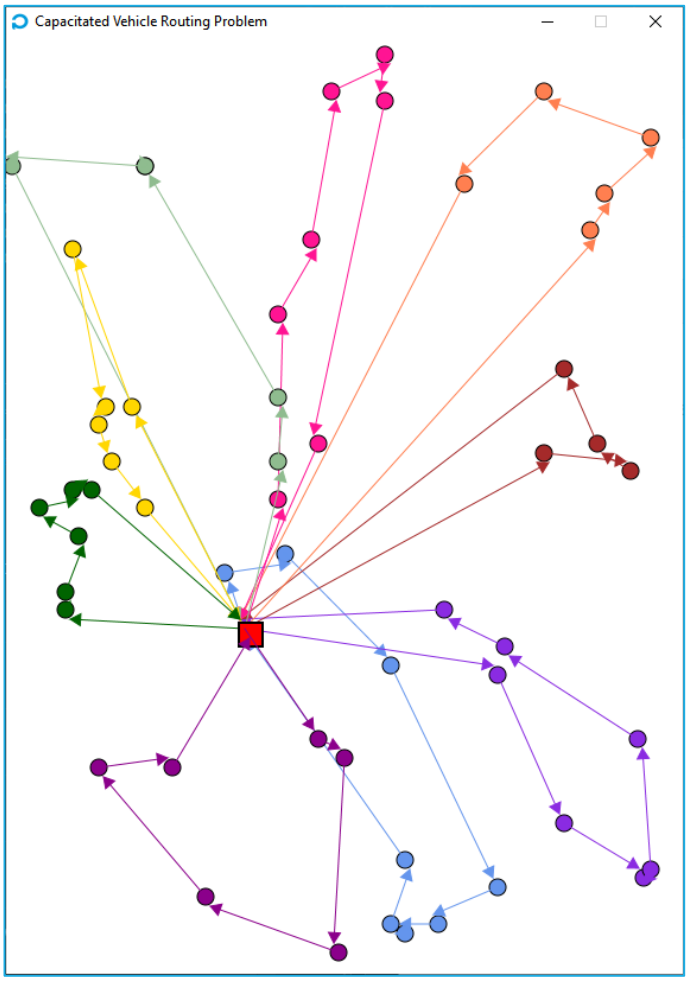


Figure 20 : Meilleure solution trouvée sur le fichier A5509 avec Tabou.

- Taille maximale liste tabou : 400
- Nombre de solutions voisines à chaque itération : 100
- Nombre d'itérations : 1000
- Type de génération : Aléatoire unique
- Type de transformation : 2-opt ou 2-opt couplé à de la méta-transformation « échange » (les deux donnent des résultats très similaires).

Avec les mêmes paramètres que précédemment, on trouve également des résultats très concluants sur le nouveau fichier ajouté sur Claroline, contenant 100 clients, le fichier **R101**, avec une fitness à **980.6442830.**

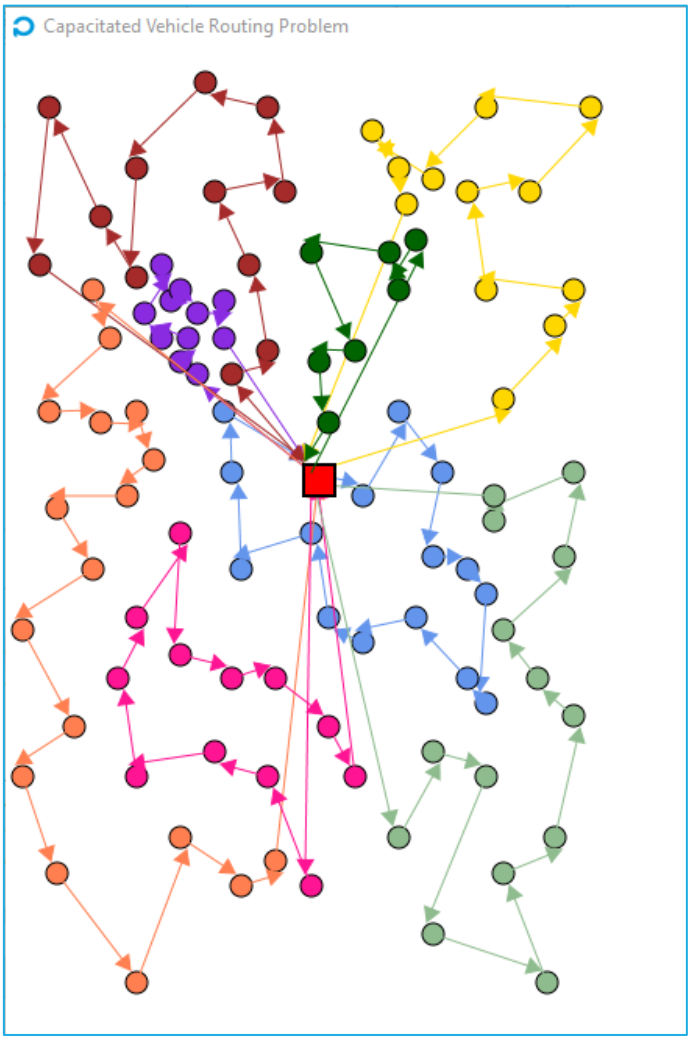
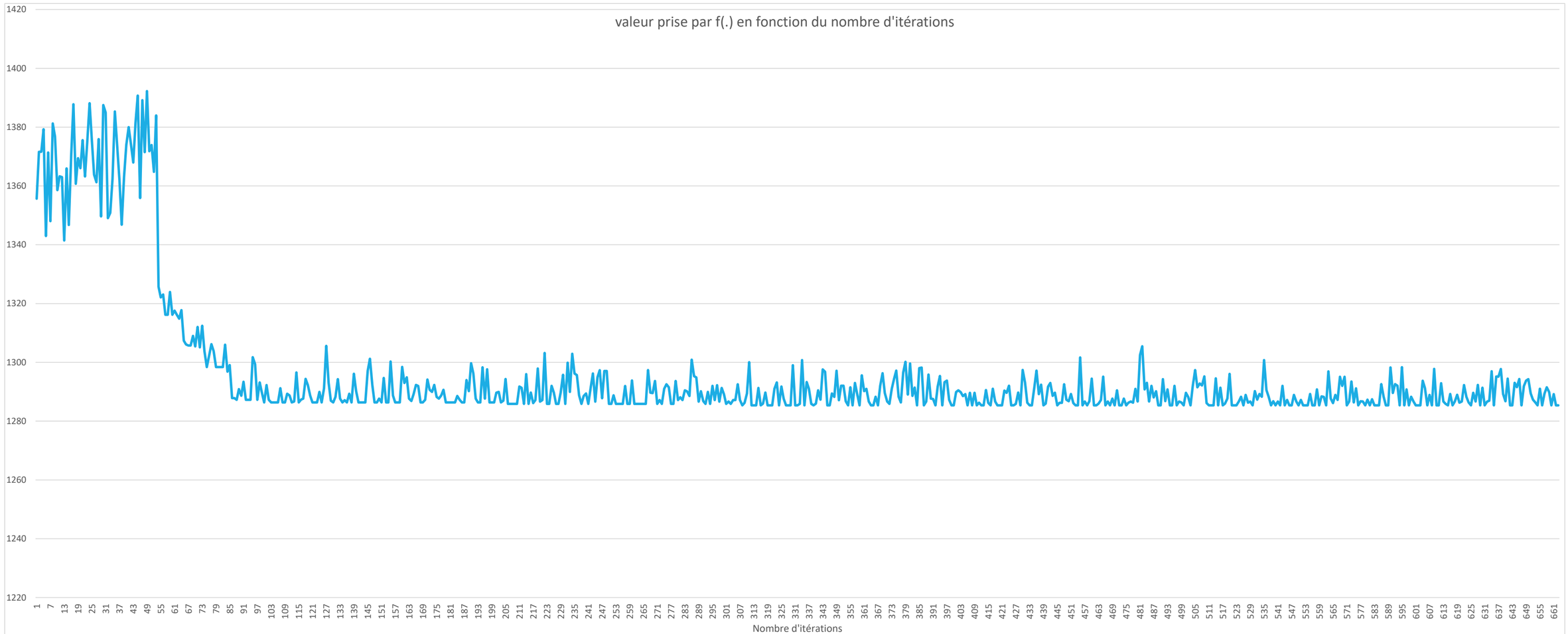


Figure 21 : Meilleure solution trouvée sur le fichier R101 avec Tabou.

Concernant le fichier sur lequel ont été réalisés nos tests, **il nous a semblé intéressant d'observer les valeurs prises par la fitness courante**, à chaque itération de l'algorithme de Tabou. Ainsi, il est possible de visualiser le « parcours » de l'algorithme et de le voir évoluer dans l'espace des solutions, jusqu'à converger vers des solutions optimales. **On voit clairement l'algorithme trouver des minimas locaux**, puis **sortir de ces minimas pour aller explorer le voisinage**. **Cela confirme totalement ce qui a été vu en cours**.



4.3. Comparaison des deux méthodes.

Dans l'ensemble, les deux méthodes sont assez efficaces pour résoudre le problème du *Capacitated Vehicle Routing Problem*. Les deux possèdent des fonctionnements (et des paramètres) bien différents. Nous obtenons des résultats assez similaires entre les deux métaheuristiques, cependant, la méthode de tabou possède un temps d'exécution plus long.

5. CONCLUSION

Ce TP portant sur des méthodes recherche à base de voisinage fut très intéressant et nous a largement permis de comprendre les différents concepts du cours. Nous avons pu entre autre mettre en place l'algorithme du recuit simulé et de la recherche tabou, dans le cadre du *Capacitated Vehicle Routing Problem*.

Malgré les mesures de confinement actuelles, nous avons pu travailler efficacement et collaborer grâce à des outils de communications à distance et surtout grâce à un outil de gestion de version (GitHub).

Ce projet aura été une expérience riche, nous permettant d'approfondir nos compétences dans le langage JAVA entre autres mais surtout d'appréhender et de comprendre le fonctionnement de méthodes de recherche à base de voisinage, dans le cadre de la résolution d'un problème complexe.

En ouverture à ce projet, nous effectuerons bientôt la résolution de ce même problème mais avec des méthodes à base de population, grâce à des algorithmes génétiques. Nous aurons alors une base de comparaison solide, afin de voir quelles méthodes sont les plus efficaces, les plus rapides, mais surtout nous en sortirons plus instruits sur ces deux types de méthodes.