

Optimisation discrète

PROJET : VRPTW



Julian DEGUT
Polytech Lyon - 4A Info





[Lien du Github](#)

@jDegut

Table des matières

Introduction	4
Modélisation du problème	5
Les ensembles	5
Les paramètres	5
Variables de décision	5
Contraintes	6
Equation	6
Structure du code	7
Diagramme de classe	7
Analyse du problème.....	9
Minimum théorique	9
Opérateurs de voisinage	10
Intra relocate.....	10
Intra exchange	10
Reverse	11
2-Opt.....	11
Inter relocate.....	11
Inter exchange	12
Cross exchange.....	12
Métaheuristiques implémentées.....	13
Méthode taboue	13
Paramètres de la méthode Taboue.....	14
Recuit simulé.....	19
Paramètres du Recuit simulé.....	19
Solutions optimales avec Recuit simulé	22
Comparaison des deux métáheuristiques	24
Résolution linéaire du VRPTW	25
Estimer la difficulté de résolution du VRPTW	25
Seuil d'indécidabilité	25
Conclusion	27
Sources	28

Introduction

Le VRPTW, ou Vehicle Routing Problem with Time Windows en anglais, est un problème d'optimisation combinatoire qui se pose dans le domaine de la logistique et de la planification de la distribution.

Ce problème consiste à trouver le meilleur itinéraire pour un ou plusieurs véhicules qui doivent desservir un ensemble de clients tout en respectant des contraintes de temps. Chaque client doit être visité dans une fenêtre de temps donnée, et chaque véhicule a une capacité maximale de transport à ne pas dépasser.

Le but du VRPTW est de minimiser le nombre de véhicules nécessaires pour desservir tous les clients dans les délais impartis, tout en minimisant la distance parcourue par les véhicules. Il s'agit donc d'un problème d'optimisation multi-objectif.

Le VRPTW est un problème NP-difficile, ce qui signifie qu'il n'existe pas d'algorithme efficace permettant de trouver une solution optimale en un temps raisonnable pour des instances de grande taille. Cependant, de nombreuses méthodes heuristiques et métaheuristiques ont été développées pour trouver des solutions de qualité en un temps raisonnable.

Dans le cadre de notre 4^{ème} année à Polytech Lyon en spécialité informatique, il nous est demandé de réaliser un projet modélisant le VRPTW. Ce projet vise à mettre en œuvre diverses méta-heuristiques pour résoudre ce problème.

Le fait d'inclure cette dernière contrainte indique que nous travaillons sur un CVRPTW ; toutefois, par abus de langage, nous dirons dans cette étude VRPTW.

Modélisation du problème

Le Vehicle Routing Problem with Time Windows (VRPTW) peut être modélisé par un programme linéaire mixte, composé d'ensembles, de différents paramètres, variables de décision, contraintes et d'une équation à minimiser.

Les ensembles

Dans ce problème, on distingue trois ensembles distincts :

- l'ensemble des clients et dépôts : $C = \{0, 1, \dots, n + 1\}$ | Clients : $C^* = C - \{0, n + 1\}$
- l'ensemble de véhicules $V : V = \{0, 1, \dots, m\}$
- l'ensemble des arcs E , avec E^* les arcs non incidents à 0 et $n+1$, formant le graphe $G(C, E)$

Les paramètres

Afin de correctement modéliser les contraintes et la formules, nous devons définir plusieurs paramètres :

- Q la capacité maximum
- q_i la capacité à livrer à $i \in C (q_0 = q_{n+1} = 0)$
- d_i la durée de livraison en $i \in C (d_0 = d_{n+1} = 0)$
- $[a_i ; b_i]$ la plage horaire dans laquelle la livraison doit débuter ($[a_0 ; b_0] = [a_{n+1} ; b_{n+1}]$)
- $t_{i,j}$ le temps de trajet associé à l'arc $(i, j) \in E$
- $c_{i,j}$ le coût du trajet associé à l'arc $(i, j) \in E$
- h_i^k l'heure du début de la livraison du client $i \in C$ par le véhicule k

Dans notre cas, nous décidons que le coût de déplacement d'un client i à un client $j \neq i$ est égale à la distance qui les sépare. De même le temps mis par un véhicule pour se déplacer est égal à la distance.

Variables de décision

Il nous faut maintenant déterminer nos variables permettant de créer une solution grâce à l'équation finale. Nous aurons deux variables de décision distinctes :

- $x_{i,j}^k = 1$ si l'arc $(i, j) \in E$ est présent dans la tournée du véhicule k , 0 sinon

Contraintes

Pour résoudre ce problème et trouver une solution valide, nous avons affaire à plusieurs contraintes :

- $\forall k \in V, \sum_{j \in C^*} x_{0j}^k = 1$: chaque véhicule doit commencer sa tournée par le dépôt
- $\forall k \in V, \sum_{i \in C^*} x_{i,n+1}^k = 1$: chaque véhicule doit finir sa tournée par le dépôt
- $\forall p \in C^*, \forall k \in V, \sum_{i \in C, i \neq p} x_{ip}^k = \sum_{j \in C, j \neq p} x_{jp}^k$: chaque client visité, et donc possédant un arc entrant, doit posséder un arc sortant sans lequel le véhicule ne pourrait se rendre au prochain client
- $\forall j \in C^*, \sum_{k \in V} \sum_{i \in C, i \neq j} x_{ij}^k = 1$: chaque client est visité exactement une fois
- $\forall k \in V, \sum_{i \in C} \sum_{j \in C, i \neq j} q_i x_{ij}^k \leq Q$: la somme des demandes de chaque client visité par un véhicule doit être inférieure à la capacité maximale du véhicule
- $\forall (i, j) \in E, \forall k \in V, h_i^k + d_i + t_{ij} - h_j^k \leq (1 - x_{ij}^k)M$: le véhicule doit arriver chez le client avant la fin de la plage horaire
- $\forall i \in C, \forall k \in V, a_i \leq h_i^k \leq b_i$: le véhicule peut commencer la livraison chez le client uniquement pendant la plage horaire

Equation

Le VRPTW consiste à minimiser la distance totale des routes empruntées par les véhicules :

$$\text{Minimiser } z = \sum_{k \in V} \sum_{i \in C} \sum_{j \in C} c_{ij} x_{ij}^k$$

Structure du code

Afin d'initialiser notre projet, nous commençons par établir une réflexion sur comment organiser nos packages, classes et méthodes afin qu'elles puissent être optimales.

Les packages seront répartis de façon à encapsuler les différents types de classes créés. Nous aurons besoin de plusieurs types de classes : les entités, les algorithmes, les opérateurs, les classes relatives au chargement des données, et à la génération d'un graphe.

Diagramme de classe

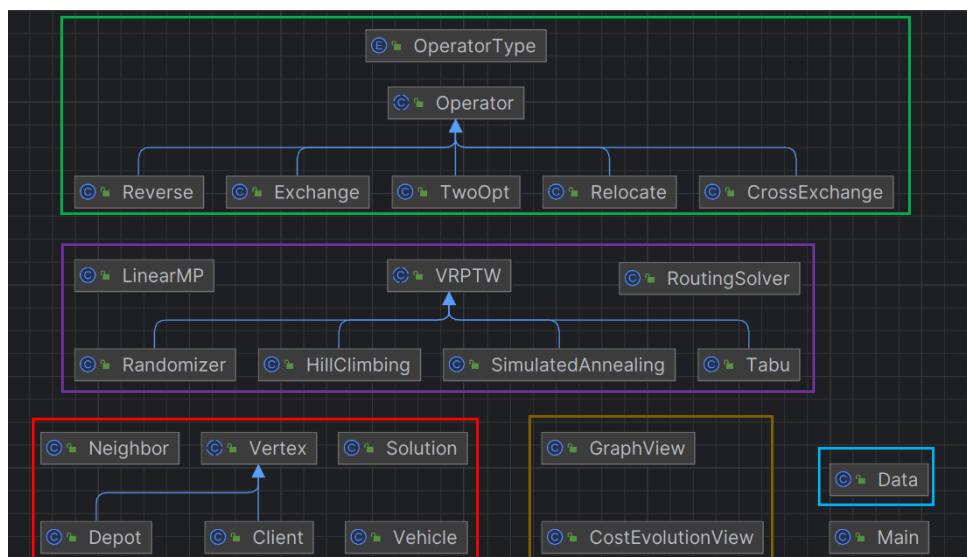


Diagramme de classe : ensemble des packages & classes

Sur ce diagramme, on peut observer toutes les relations entre les classes du projet. Chaque package a été encadré comme suivant : **model**, **view**, **data**, **algorithms**, **operators**.

Parmi ces classes, nous retrouvons notre classe principale *Main*, nous permettant d'exécuter notre application Java, notre classe abstraite *Operator* parente de chaque classe représentant un opérateur de voisinage.

Les algorithmes implémentés étendent la classe *VRPTW* contenant les méthodes nécessaires aux fonctionnements de ces derniers.

Les données utilisées pour cette étude sont récupérées grâce à la classe *Data* en utilisant les classes IO Stream. Cette classe permet de récupérer les dépôts et clients à partir d'un fichier.

Un graphe est représenté comme une solution du problème. Il possède des sommets et des arcs représentés sous formes de routes (véhicules). Chaque véhicule possède une liste de clients formant bout-à-bout la route associée.

Afin de pouvoir observer le comportement de la solution en temps réel, nous utilisons des *Viewer* se chargeant de nous afficher la solution et son coût en temps réel.

Voici ci-dessous le diagramme de classe avec les détails des méthodes :

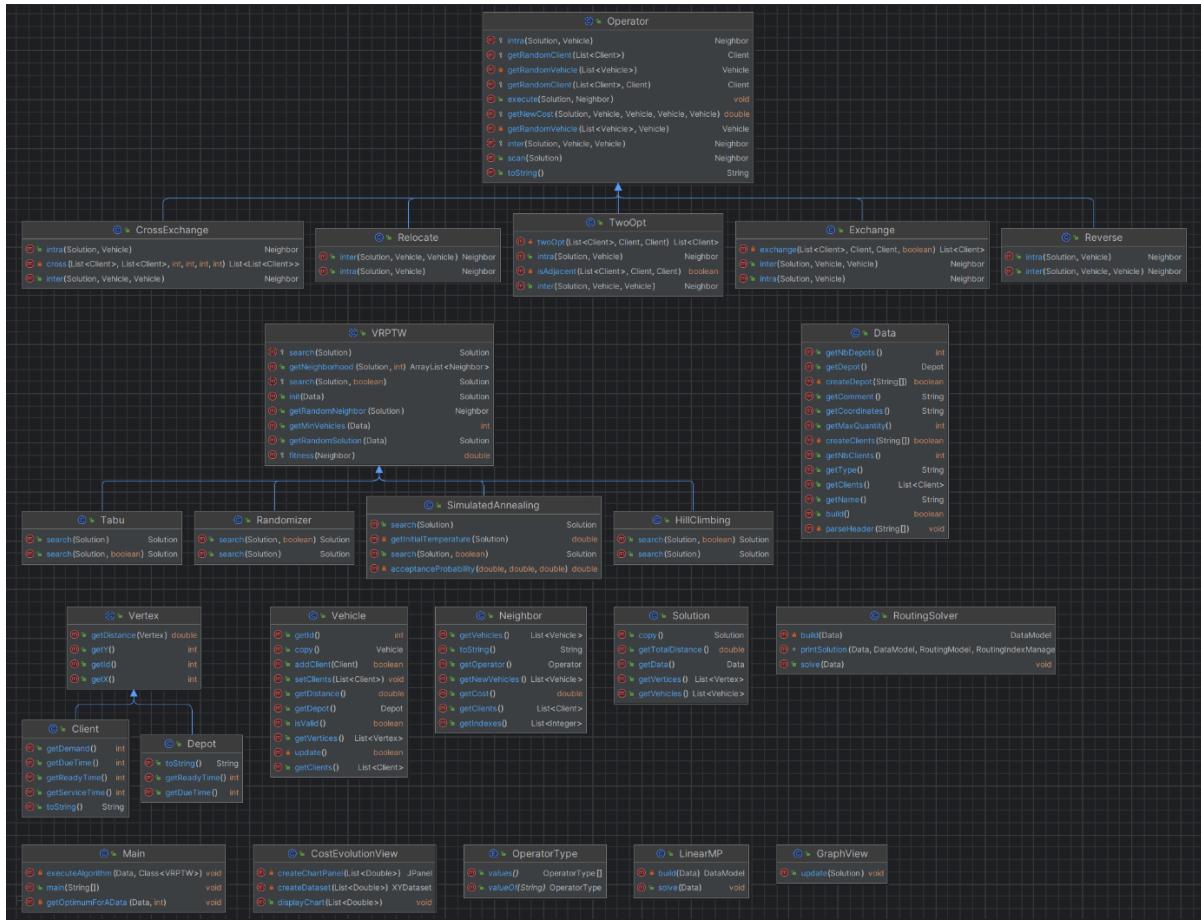


Diagramme de classe : ensemble des classes et leurs méthodes

Analyse du problème

Une solution du problème est caractérisée par son coût et le nombre de véhicules requis pour effectuer la tournée. L'optimum globale est donc la solution possédant une combinaison du coût total et du nombre de véhicules utilisés inférieur à celle de n'importe quelle autre solution.

Minimum théorique

Une première approche de ce problème consiste à calculer le nombre de véhicules nécessaires en théorie. Pour cela, nous devons nous aider de la capacité maximale de chaque véhicule avec celle demandée à la livraison de chaque client.

Ce nombre théorique est en général une représentation idyllique puisque nous ne prenons pas en compte les fenêtres de temps de chaque client (cela reviendrait à calculer l'optimum global ce qui n'est clairement pas l'objectif ici).

La somme des demandes de livraison de chaque client est égale à la demande totale de la solution. Chaque véhicule possède une capacité maximale de chargement : il ne peut donc logiquement pas livrer plus que ce qu'il ne peut. Il nous suffit donc de diviser la somme totale de la marchandise par la capacité totale par véhicule, et de prendre l'arrondi au supérieur :

$$V_{minTheo} = \frac{\sum_{i \in C} q_i}{Q_i}$$

Dans le cadre de notre étude, nous avons obtenons ce tableau :

Fichier	Quantité totale	Quantité maximale	V _{min}
Data101.vrp	1458	200	8
Data102.vrp	1458	200	8
Data111.vrp	1458	200	8
Data112.vrp	1458	200	8
Data201.vrp	1458	1000	2
Data202.vrp	1458	1000	2
Data1101.vrp	1724	200	9
Data1102.vrp	1724	200	9
Data1201.vrp	1724	1000	2
Data1202.vrp	1724	1000	2

Opérateurs de voisinage

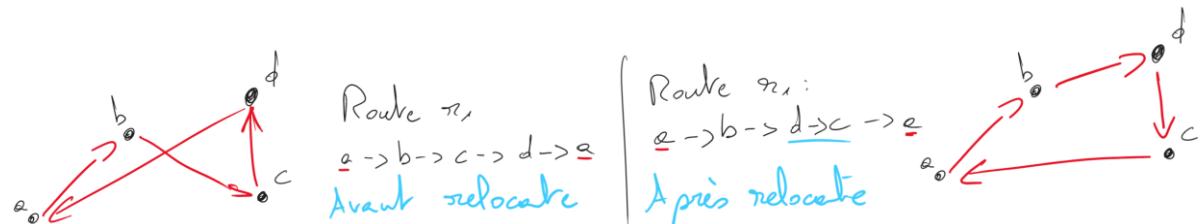
Les opérateurs de voisinage sont des outils clés pour la mise en œuvre des métahéuristiques pour résoudre le problème de VRPTW. Ces opérateurs sont utilisés pour explorer l'espace de recherche en générant de nouvelles solutions à partir des solutions existantes.

Il existe deux types d'opérateur de voisinage : les opérateurs dits intra-groupe et inter-groupes. Dans le cadre de notre étude, les opérateurs du premier type sont utilisés pour modifier l'ordre des clients visités dans une même tournée.

De l'autre côté, les opérateurs du second type sont utilisés pour modifier les relations entre les tournées en transférant un client d'une tournée à une autre. Ces opérateurs de voisinage sont utilisés pour explorer de nouvelles solutions qui améliorent les solutions actuelles en termes de coût total, de distance parcourue, de durée totale et de nombre de véhicules utilisés.

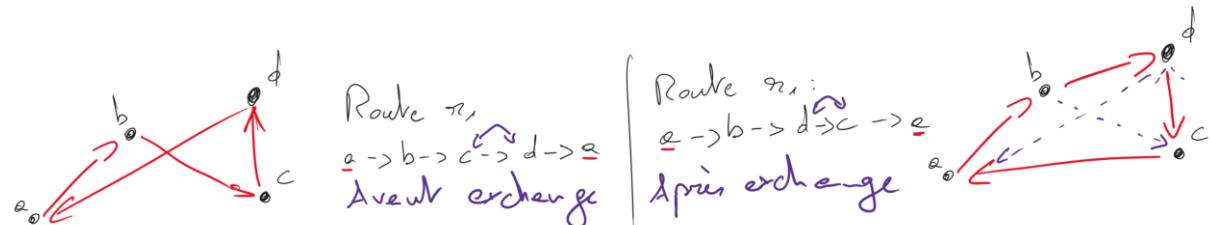
Intra relocate

L'opérateur « Intra relocate » permet de repositionner un client appartenant à une tournée au sein de celle-ci.



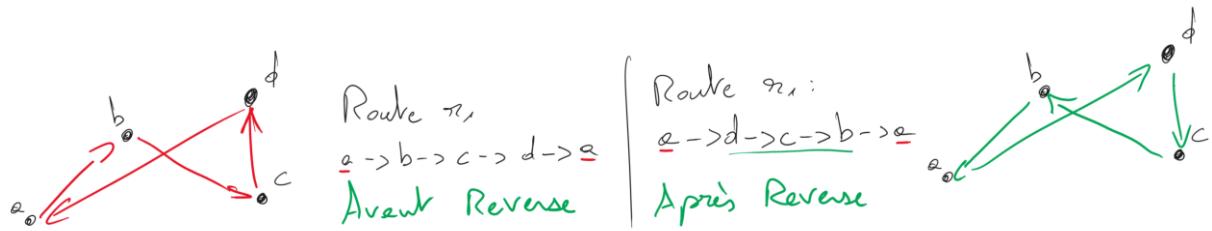
Intra exchange

L'opérateur « Intra exchange » permet d'échanger deux clients appartenant à une même tournée.



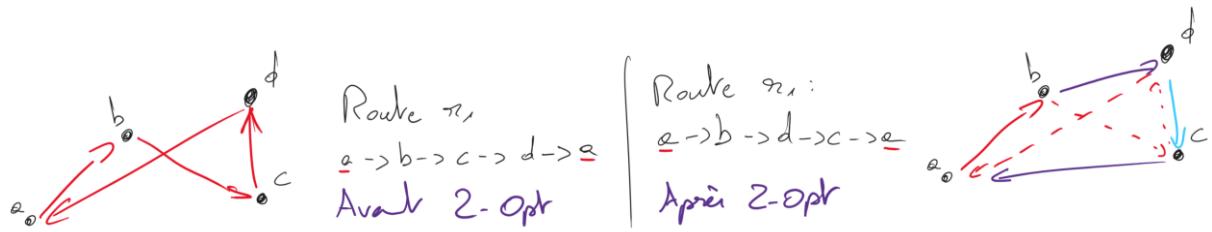
Reverse

L'opérateur « Reverse » permet d'inverser le sens de visite de l'intégralité d'une tournée.



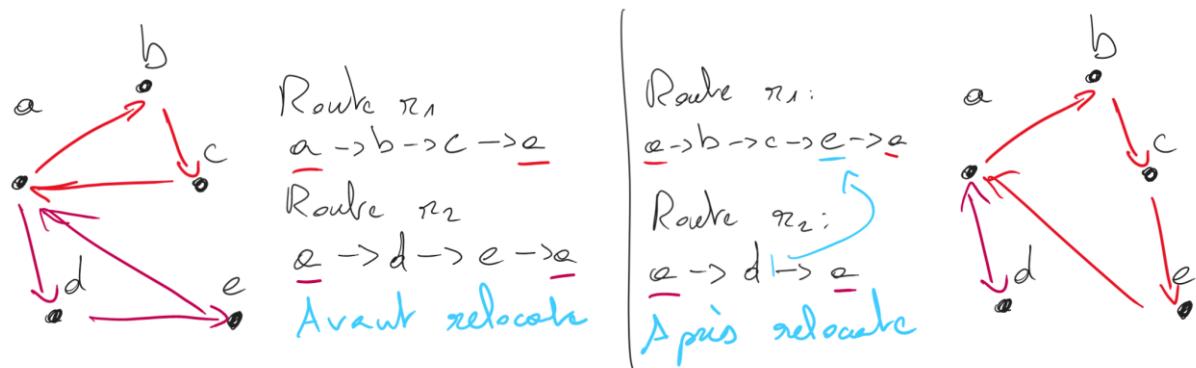
2-Opt

L'opérateur « 2-Opt » ou « TwoOpt » permet de modifier la tournée en enlevant deux arêtes non adjacentes et de relier les morceaux de tournée restants.



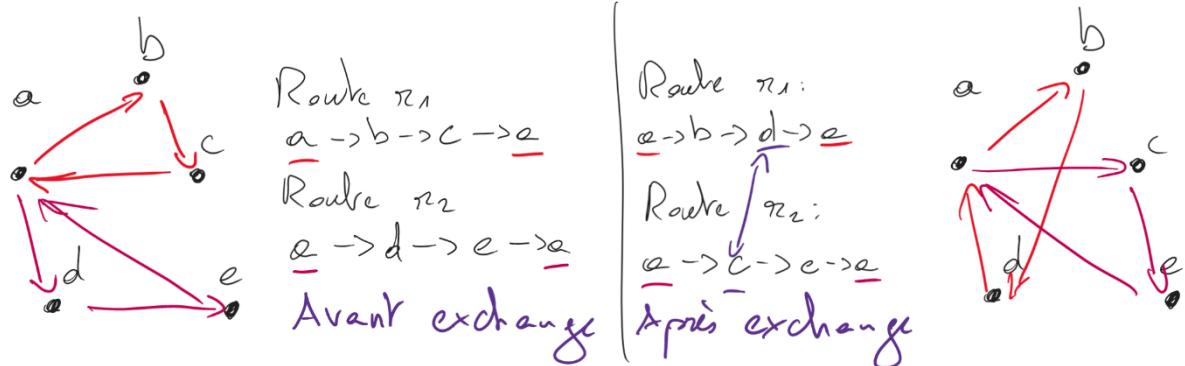
Inter relocate

L'opérateur « Inter relocate » consiste à repositionner aléatoirement un client d'une tournée dans une autre tournée.



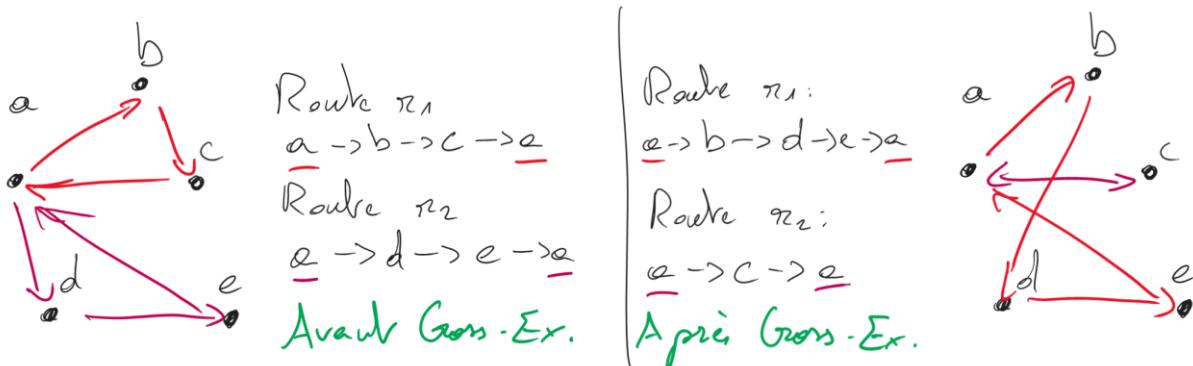
Inter exchange

L'opérateur « Inter exchange » consiste à échanger deux clients aléatoirement appartenant à deux tournées différentes.



Cross exchange

L'opérateur « Cross exchange » consiste à échanger aléatoirement deux morceaux de deux tournées différentes.



En utilisant ces opérateurs de voisinage dans le cadre de métaheuristiques telles que le recuit simulé, la recherche taboue, la méthode de colonie de fourmis et la recherche locale itérative, il est possible de résoudre le VRPTW de manière efficace et efficiente.

Métaheuristiques implémentées

Le VRPTW peut être résolu grâce à de nombreuses méthodes aujourd’hui. Parmi ces méthodes, il y a celles qui donnent une solution exacte (recherche arborescente Branch&Bound, PLM en nombre entiers Branch&Cut, etc...), celles fournissant une solution dites approchée : les métaheuristiques (ou même les heuristiques dans certains cas).

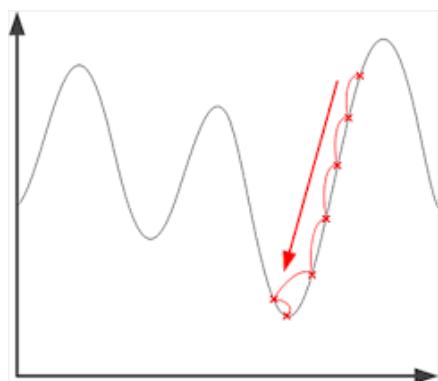
Une métaheuristique se définit comme une méthode de résolution de problèmes en utilisant une approche itérative visant à explorer un large panel de solutions et donc de donner les meilleures solutions, voire la plus optimale dans certaines conditions.

De nombreuses métaheuristiques existantes permettent de résoudre le VRPTW, toutefois nous utiliserons uniquement deux d’entre-elles se basant sur des opérateurs de voisinage pour la suite de l’étude.

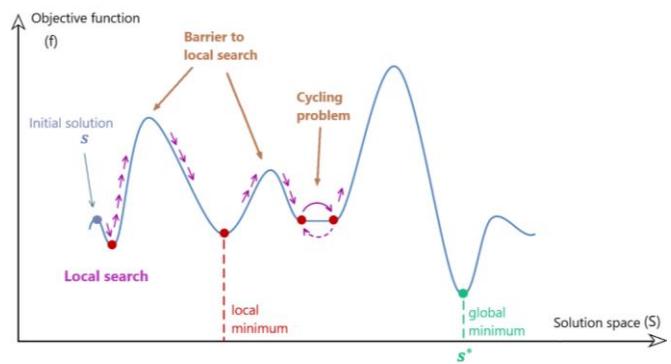
Méthode taboue

En optimisation combinatoire, l’algorithme de descente de gradient est très fréquent par sa simplicité d’implémentation. Elle consiste à chercher la meilleure solution voisine à partir d’une solution initiale, et ce jusqu’à l’entièreté du voisinage soit moins bon que la solution traitée. Cet algorithme permet de trouver efficacement un optimum local d’une fonction.

La méthode taboue (Tabu search) reprend le fonctionnement de l’algorithme de méthode de gradient en insérant la possibilité que la solution se dégrade afin de permettre de ne pas limiter la recherche à un seul optimum local.



Hill-Climbing



Tabu Search

Une fois l'optimum local atteint, la solution va venir se dégrader en interdisant le fait de revenir sur l'optimum local précédemment visité. Cette interdiction passe par une liste « taboue » contenant n éléments à définir.

Une fois la liste pleine, on la nettoie en enlevant l'élément le plus ancien afin de pouvoir insérer un nouvel élément au début de la liste.

Paramètres de la méthode Taboue

Afin d'obtenir la meilleure solution de la manière la plus efficace et moins coûteuse possible, il est important de bien configurer les paramètres utilisés pour la recherche. Dans la méthode taboue, nous avons 3 paramètres à prendre en compte :

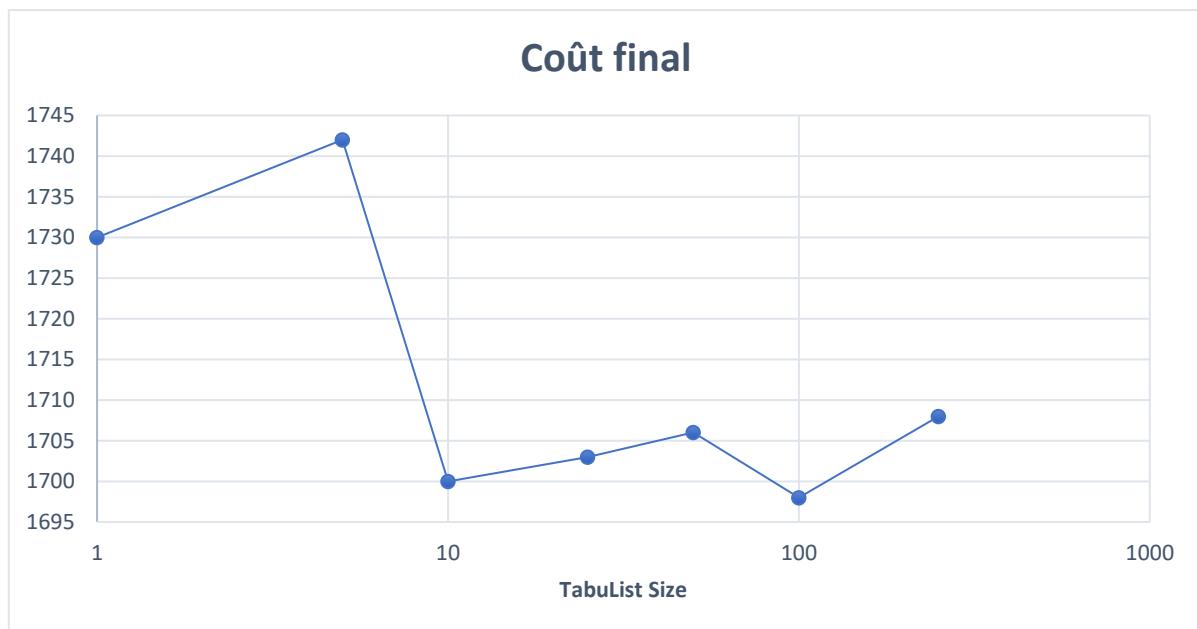
- La taille de la liste taboue : le paramètre spécifique à cette métahéuristique ; il déterminera la flexibilité des opérations de voisinages
- Le nombre d'itérations maximal : pour une raison évidente de coût de calcul, le nombre d'itération de la recherche est limitée
- Le nombre maximal de voisins (par opérateur) par itération : l'objectif est de rechercher le meilleur voisin parmi un voisinage généré aléatoirement

Afin de résoudre nos problèmes, nous devons préalablement trouver la configuration correspondant le plus à nos besoins. Ainsi, en faisant varier ces paramètres, nous obtenons avec le fichier *Data101.vrp* :

TabuList Size	Bloqués tabous	Temps de recherche (ms)	Coût final
1	8	771	1730
5	36	707	1742
10	38	775	1700
25	143	743	1703
50	144	641	1706
100	87	757	1698
250	323	740	1708

Réalisé avec $\text{maxIterations} = 1000$ et $\text{maxNeighbors} = 100$

On remarque l'influence faible de la taille de la liste taboue sur le temps. Quant au coût final, il est réduit à partir de $n = 10$ pour ensuite stagner. Enfin le nombre de voisins bloqués varie fortement avec $n \geq 25$.



Max Iterations	Bloqués tabous	Temps de recherche (ms)	Coût final
100	0	245	2136
500	4	468	1763
1000	36	609	1747
5000	657	1699	1704
10000	1083	3204	1649
15000	1585	4481	1720
30000	2622	9395	1657

Réalisé avec $n = 10$ et $\text{maxNeighbors} = 100$

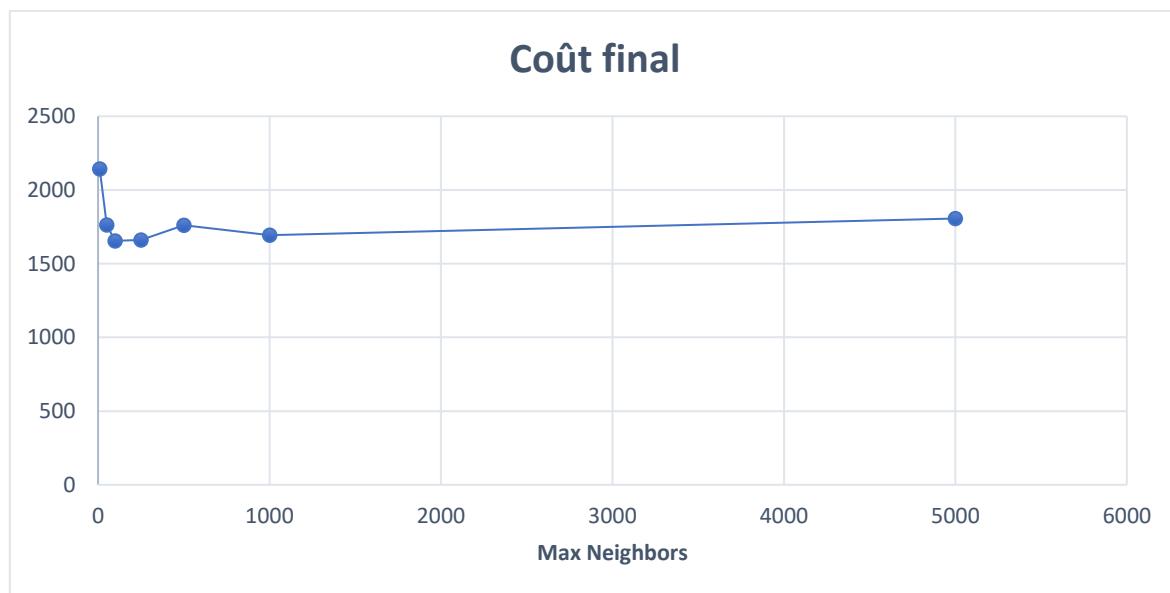
L'influence de ce paramètre permet de faire varier le coût final.



Max Neighbors	Bloqués tabous	Temps de recherche (ms)	Coût final
10	110	1113	2142
50	479	1925	1762
100	994	3178	1655
250	1797	7471	1661
500	0++	15735	1761
1000	0++	24295	1694
5000	0++	125813	1806

Réalisé avec $n = 10$ et $\text{maxIterations} = 10000$

Le nombre maximal de voisins influe grandement le coût final.



Dans le cadre de nos études, nous prendrons pour paramètres $n = 10$, $\text{maxIterations} = 10000$ et $\text{maxNeighbors} = 250$.

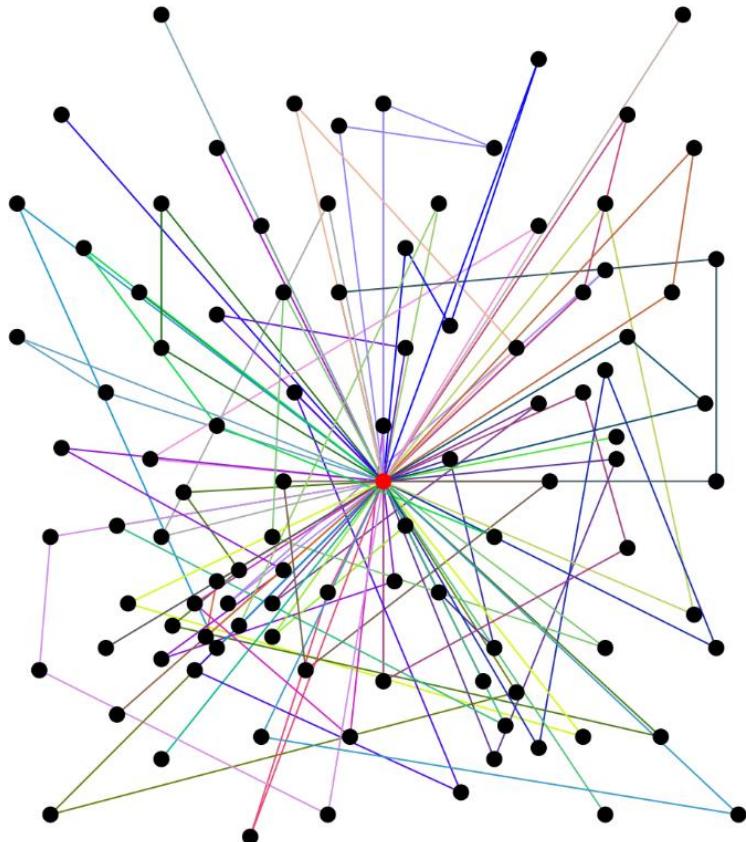
Solutions optimales avec Taboue

En partant d'une solution initiale aléatoire, avec une liste taboue de taille $n = 10$, un nombre maximal d'itérations $\text{maxIterations} = 10000$ et un nombre maximal de voisins par voisinage $\text{maxNeighbors} = 250$, nous avons réalisé 10 exécutions de l'algorithme en parallèle afin d'obtenir la meilleure solution. Voici les résultats :

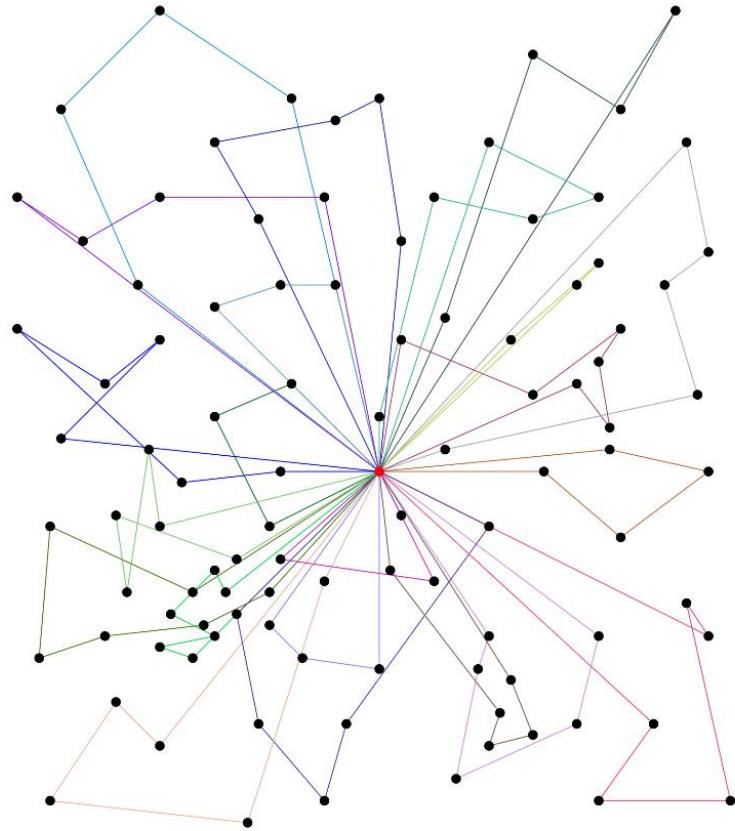
Fichier	Coût final ; V_{totFinal}	Bloqués tabous (max)	Temps d'exécution (x10 en parallèle)
Data101.vrp	1655 ; 21	2735	69891ms
Data102.vrp	1489 ; 19	137580	63701ms
Data111.vrp	1109 ; 13	449172	48467ms
Data112.vrp	1019 ; 11	686098	48453ms
Data201.vrp	1173 ; 10	319189	66225ms

Fichier	Coût final ; $V_{totFinal}$	Bloqués tabous (max)	Temps d'exécution (x10 en parallèle)
Data202.vrp	1112 ; 12	588290	43984ms
Data1101.vrp	1741 ; 19	39596	68131ms
Data1102.vrp	1580 ; 17	195571	70586ms
Data1201.vrp	1389 ; 12	345089	50366ms
Data1202.vrp	1153 ; 10	716772	45686ms

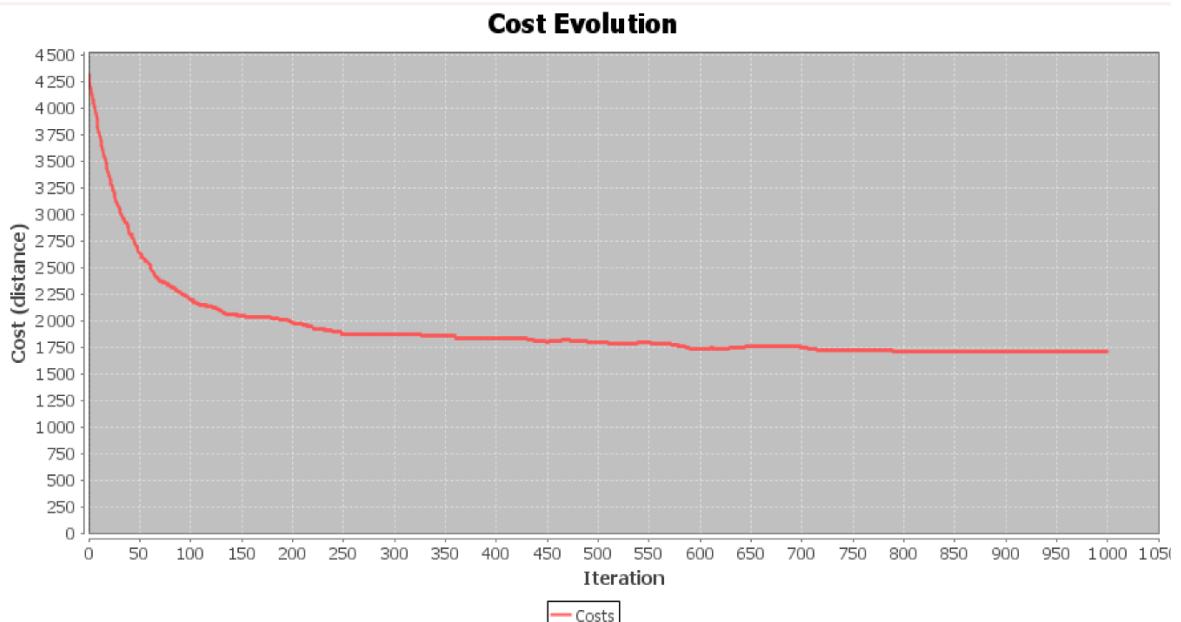
On remarque après application de l'algorithme Tabu Search que la solution optimale finale obtenue varie beaucoup pour les mêmes paramètres. Les résultats obtenus sont de bonne qualité mais le plus gros avantage de cette méthode est son temps d'exécution qui est très faible. En effet, lors de l'application de l'algorithme, la recherche de voisins par opérateurs peut se faire en parallèle.



Fichier data101.vrp : étape initiale (solution aléatoire)



Fichier data101.vrp : étape finale, solution optimale après algorithme Tabu Search



Fichier data101.vrp : évolution du coût de la solution en fonction de l'itération de l'algorithme Tabu Search

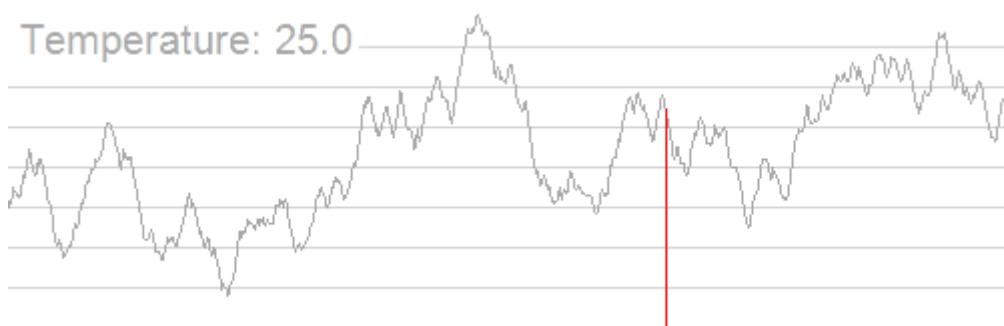
Sur le graphique ci-dessus, on remarque des dégradations (vers l'itération n°650 par exemple) du coût de la solution afin de laisser l'algorithme explorer d'autre solutions. Les paramètres à définir pour appliquer la méthode taboue sont crucial pour le comportement celle-ci (la solution finale est une solution généralement proche de l'optimum global).

Recuit simulé

L'algorithme de recuit simulé est une méthode de recherche métaheuristique inspirée du processus de recuit en physique. Cette méthode permet d'explorer l'espace de recherche pour trouver une solution de qualité pour un problème d'optimisation donné, comme le VRPTW.

L'algorithme commence par générer une solution initiale aléatoire et la considère comme la meilleure solution trouvée jusqu'à présent. Ensuite, il effectue des itérations en modifiant légèrement la solution courante à l'aide d'opérateurs de voisinage. La nouvelle solution est alors évaluée et, si elle est meilleure que la solution courante, elle est acceptée comme nouvelle solution courante. Si la nouvelle solution est moins bonne, elle peut être acceptée avec une probabilité calculée en fonction de la température actuelle.

La température diminue progressivement au cours des itérations, ce qui réduit la probabilité d'accepter des solutions moins bonnes et permet à l'algorithme de converger vers une solution de qualité.



Exemple du fonctionnement du recuit simulé (gif)

Paramètres du Recuit simulé

Comme la méthode taboue, la météahuristique du recuit simulé a besoin d'être configuré afin d'être le plus efficace possible. Ces paramètres sont :

- La température initiale : un paramètre important pour le déroulement de l'algorithme
- La température finale : représentant le critère d'arrêt de l'algorithme
- Le nombre maximal d'itérations par pallier de température : il représente le nombre de test effectué afin d'améliorer ou non la solution

- Le taux de refroidissement : la température initiale doit atteindre la température finale en la multipliant par son taux de refroidissement

La température initiale est calculée suivant l'écart de coût entre la solution initiale et le pire voisinage avec la formule suivante :

$$t = \frac{-\Delta_{solution}}{\ln(0.8)}$$

Cette formule considère que la pire solution est acceptée avec une probabilité de 80%. Cette probabilité diminuera ensuite au fil des palliers.

Nous devons donc maintenant trouver les bonnes valeurs pour les 3 autres paramètres. Pour cela, nous effectuons des tests sur le fichier *Data101.vrp* :

Cooling Rate	Température initial	Temps de recherche (ms)	Coût final
0.99	266	112703	1651
0.95	260	16280	1654
0.9	221	6523	1655
0.85	185	4211	1653
0.8	216	3125	1664

Réalisé avec *maxIterationsPerTemp* = 1000 et *finalTemperature* = 0.1

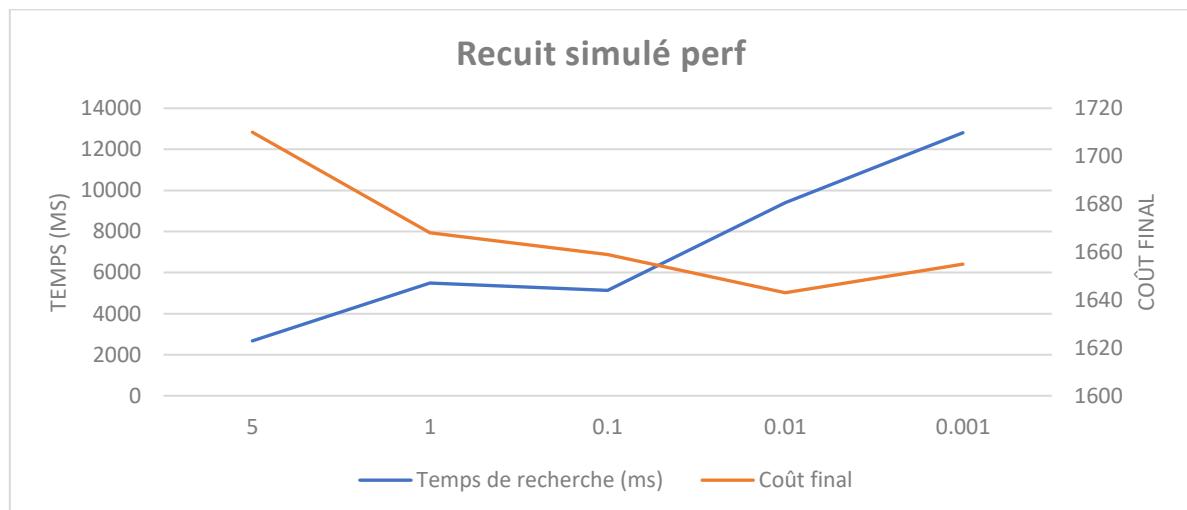
On remarque clairement une diminution du temps de recherche puisque le taux de refroidissement représente le pourcentage de solutions parcourue au sein de l'algorithme (vitesse de convergence vers la solution optimale). Toutefois, le coût final lui n'en est pas beaucoup impacté.



Température finale (<)	Température initial	Temps de recherche (ms)	Coût final
5	254	2672	1710
1	211	5482	1668
0.1	252	5131	1659
0.01	201	9394	1643
0.001	459	12808	1655

Réalisé avec $maxIterationsPerTemp = 1000$ et $coolingRate = 0.9$

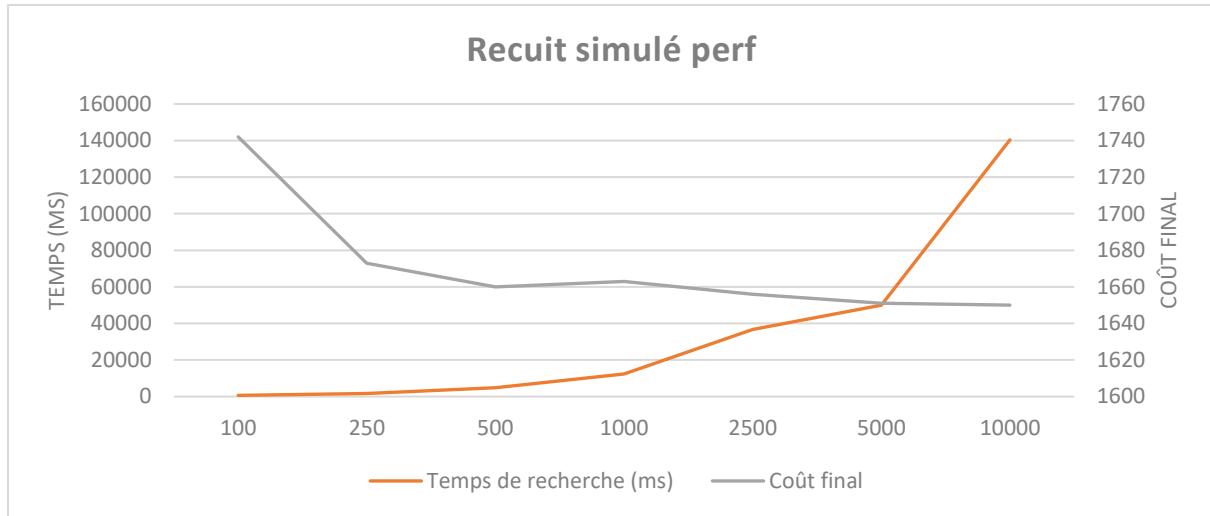
La température finale influe sur le coût final, mais surtout sur le temps de recherche. Malgré le fait que pour chacune des valeurs testées, le temps de recherche reste raisonnable, il est inutile d'avoir une température finale trop basse (coût final parfois plus haut).



Max Iterations per Temp	Température initial	Temps de recherche (ms)	Coût final
100	184	646	1742
250	277	1735	1673
500	182	4744	1660
1000	263	12310	1663
2500	303	36661	1656
5000	357	50004	1651
10000	283	140389	1650

Réalisé avec $finalTemperature = 0.01$ et $coolingRate = 0.9$

Le nombre maximal d'itérations par palier de température est également un facteur premier de la valeur du coût final. Toutefois, tout comme la température finale, il est important de ne pas choisir ni trop petit, ni trop grand afin de garder un temps de calcul raisonnable.



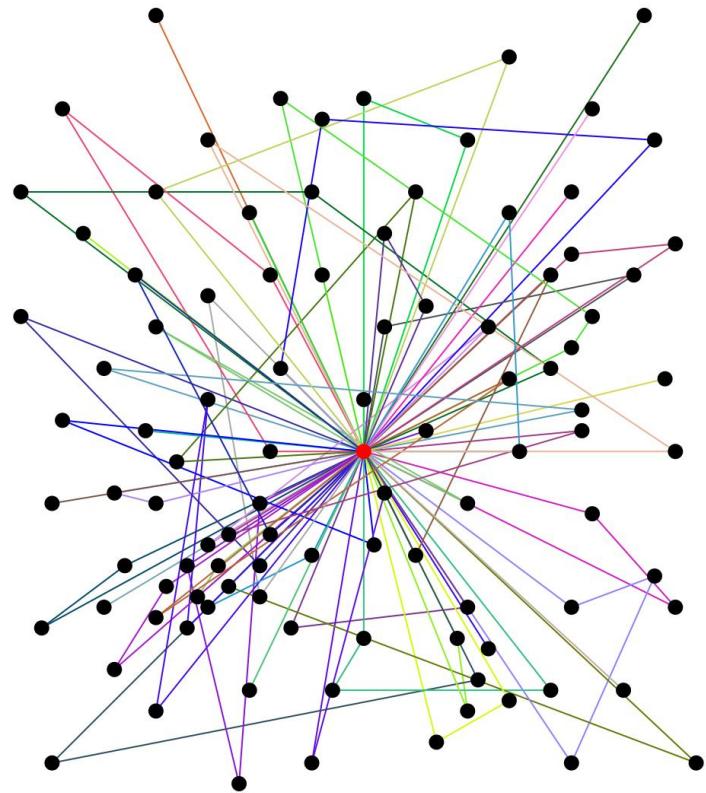
Ainsi, pour notre utilisation, nous configurerons le recuit simulé avec pour paramètres $finalTemperature = 0.01$, $coolingRate = 0.9$ et $maxIterationsPerTemp = 1000$.

Solutions optimales avec Recuit simulé

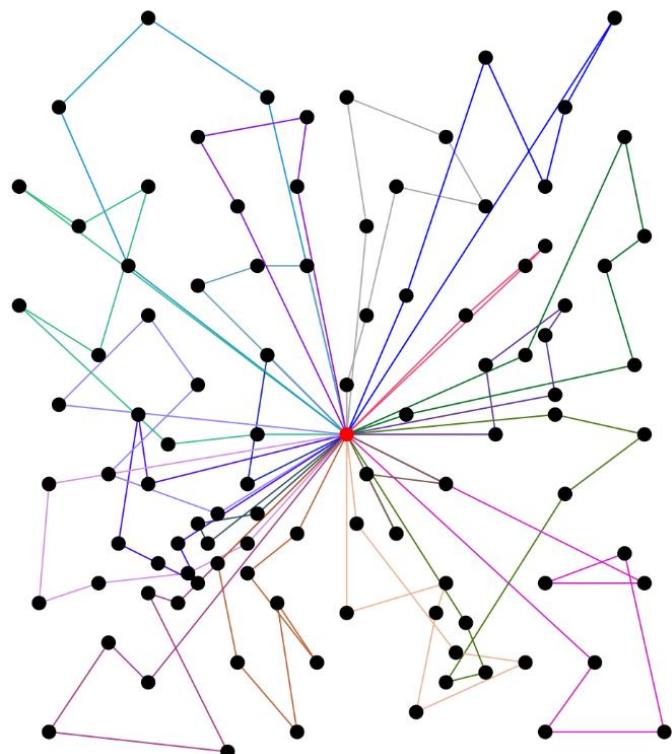
Nous appliquons, 10 fois en parallèle, cet algorithme suivant la configuration de plusieurs paramètres importants, à savoir le nombre d’itérations par changement de température $maxIterationsPerTemp = 1000$, le taux de refroidissement $coolingRate = 0.9$ et la température final (notre critère d’arrêt) $finalTemperature = 0.01$. Voici les solutions finales obtenues :

Fichier	Coût final ; $V_{totFinal}$	Initial temperature (max)	Temps d’exécution (x10 en parallèle)
Data101.vrp	1643 ; 20	340	28157ms
Data102.vrp	1482 ; 18	567	5324ms
Data111.vrp	1095 ; 12	293	2490ms
Data112.vrp	1046 ; 12	287	1700ms
Data201.vrp	1186 ; 8	471	3118ms
Data202.vrp	1097 ; 9	442	2086ms
Data1101.vrp	1668 ; 18	443	7471ms
Data1102.vrp	1493 ; 16	513	4951ms
Data1201.vrp	1309 ; 9	752	3336ms
Data1202.vrp	1138 ; 8	716	2340ms

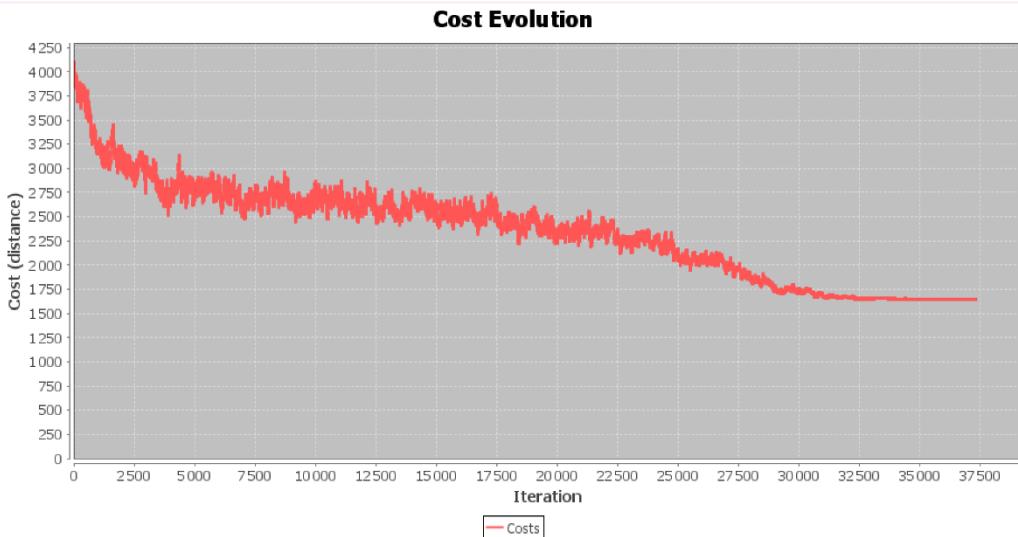
L’algorithme de recuit simulé nous donne des solutions finales majoritairement meilleures que celles obtenues avec la méthode Taboue. En effet cet algorithme explore un plus large panel de solution que la méthode Taboue (pas de remontée de col, mais un choix de voisin aléatoire). Cependant le désavantage de cet algorithme est son temps d’exécution : la génération de voisinage ne peut pas se faire en parallèle puisqu’il doit itérer et traiter voisin par solution



Fichier data101.vrp : étape initiale (solution aléatoire)



Fichier data101.vrp : étape finale, solution optimale après algorithme Simulated Annealing



Fichier data101.vrp : évolution du coût de la solution en fonction de l’itération de l’algorithme Simulated Annealing

Le graphique ci-dessus nous indique l’évolution du coût de la solution au fil des itérations. Nous remarquons beaucoup de dégradations dans les premiers paliers dû à une température élevée, contrairement aux derniers paliers où la température s’approche de *finalTemperature* = 0.1.

Comparaison des deux métaheuristiques

Après avoir réalisé l’ensemble de nos expériences sur les données et ainsi obtenu les résultats des deux différentes métaheuristiques, nous pouvons maintenant les comparer.

Fichier	TabuSearch	SimulatedAnnealing	Différence
Data101.vrp	1655	1643	-12
Data102.vrp	1489	1482	-7
Data111.vrp	1109	1095	-14
Data112.vrp	1019	1046	27
Data201.vrp	1173	1186	13
Data202.vrp	1112	1097	-15
Data1101.vrp	1741	1668	-73
Data1102.vrp	1580	1493	-87
Data1201.vrp	1389	1309	-80
Data1202.vrp	1153	1138	-15

On remarque un écart (Δ) majoritairement négatif entre les coûts finaux de la méthode taboue et le recuit simulé : la méthode du recuit simulé est plus longue que la méthode taboue, mais les solutions finales sont majoritairement meilleures.

Résolution linéaire du VRPTW

Le problème du voyageur de commerce, qui est un problème de la famille des problèmes NP-Difficile, ne peut pas être résolu efficacement pour de grands paramètres. Notre VRPTW est une version modifiée du VRP, qui lui-même est une modification du TSP, et donc fait également partie de cette même famille de problèmes.

Bien que certaines méthodes de programmation linéaire, telles que l'algorithme du simplexe, puissent résoudre certains problèmes combinatoires en trouvant la solution optimale respectant toutes les contraintes, elles ne peuvent pas être utilisées efficacement pour résoudre le TSP ou le VRPTW.

Estimer la difficulté de résolution du VRPTW

Dans le cas du VRPTW, nous pouvons utiliser la modélisation du problème afin d'en implémenter un solveur linéaire. Pour cela nous utiliserons un package libre d'utilisation nous facilitons l'implémentation du problème, ainsi que sa résolution. Notre objectif consiste à observer quels paramètres influent sur la résolution du problème, et à partir de quelles valeurs ce dernier n'est pas solvable linéairement.

Il est difficile d'estimer la complexité temporelle du VRPTW puisqu'elle dépend de beaucoup de facteurs parfois inconnus : la puissance de la machine (nombre d'instructions par secondes : FLOPS) et le nombre d'instruction pour résoudre le problème lui-même dépendant d'autres facteurs, comme le nombre de clients, de véhicules disponibles et la capacité maximale de chaque véhicule.

Seuil d'indécidabilité

Dans un cas général de résolution de problème NP-Difficile, la complexité est de l'ordre de $O(2^n)$ avec n le nombre de variables décisionnelles (binaires). Le nombre n est obtenu pour le VRPTW grâce à cette formule :

$$n = (nb_{Clients} + nb_{Dépôts} + 1) * nb_{Véhicules}$$

Cependant, l'objectif de la programmation linéaire est de trouver la solution optimale sans avoir à tester l'intégralité des solutions réalisables. Pour ce faire, différentes méthodes sont disponibles, mais celle qui nous intéresse est la méthode du simplexe. Le solveur implémenté utilise le package Google OR-Tools reproduisant l'algorithme simplexe dont la complexité est de l'ordre de $O((n + m)^3)$ avec n le nombre de variables total, et m le nombre de contraintes d'inégalité.

Afin de déterminer le seuil d'indécidabilité du VRPTW, des expériences ont été réalisées sur un fichier de test en augmentant petit à petit le nombre de clients.

Nb de clients	Temps (secondes)	Nb de véhicules
12	15,273	1
13	41,982	1
14	23,098	1
15	26,382	1
16	202,836	1
17	/	1
18	/	1

Tableau résumant les expériences réalisées

On distingue qu'à partir de 17 clients et un seul véhicule, aucune solution n'est obtenue en un temps raisonnable. Le temps augmente drastiquement (en général) au fur et à mesure que le nombre de clients augmente.

Nous pouvons en déduire que $nbClients = 18$ clients est notre seuil d'indécidabilité pour notre problème (la solution optimale est souvent celle avec le moins de véhicules). Ce nombre peut varier en fonction du problème et de la configuration du solveur : le nombre de véhicules influe ce seuil.

Nb de clients	Temps (secondes)	Nb de véhicules
12	15,273	1
13	41,982	1
14	23,098	1
15	26,382	1
16	202,836	1
17	/	1
17	146,97	2
18	314,961	2
19	/	2
19	336,888	3
20	184,66	3
21	231,884	3
22	/	3
22	409,223	4
23	268,384	4
24	463,62	4
25	504,123	4
26	/	4
26	/	5

Tableau complet des expériences

Conclusion

En conclusion, ce rapport a abordé la résolution du VRPTW, qui est un problème classique d'optimisation combinatoire rencontré dans de nombreuses applications logistiques. Nous avons présenté une modélisation détaillée du problème, ainsi que les différentes contraintes et variables qui le caractérisent. Nous avons également examiné les méthodes couramment utilisées pour résoudre ce problème, en mettant l'accent sur l'implémentation de méta-heuristiques telles que la méthode taboue, le recuit simulé et d'autres techniques d'optimisation pour trouver des solutions efficaces.

La performance de ces méthodes d'optimisation a pu nous fournir des résultats prometteurs pour résoudre les problèmes complexes de VRPTW dans la vie réelle. Enfin, nous avons présenté une dernière partie bonus sur la Programmation Linéaire en Nombres Entiers (PLNE), qui est une autre technique couramment utilisée pour résoudre les problèmes d'optimisation combinatoire. En somme, ce rapport fournit une analyse détaillée et une mise en œuvre pratique des différentes techniques pour résoudre efficacement le problème difficile de VRPTW.

Sources

Optimisation Discrète, Cours

<https://moodle.univ-lyon1.fr/course/view.php?id=254>

Une approche génétique pour la résolution du problème VRPTW dynamique,
Thèse

https://www.lgi2a.univ-artois.fr/spip/IMG/pdf/these_haiyan_housroum.pdf

Présentation du problème, Wikipédia

https://fr.wikipedia.org/wiki/Probl%C3%A8me_de_tourn%C3%A9es_de_v%C3%A9hicules

Documentation Google OR-Tools, Google Developers

https://developers.google.com/optimization/mip/mip_example?hl=fr