Rapport de stage

Clément Legrand-Lixon

4 juillet 2018

Introduction

Le Vehicle Routing Problem (VRP), consiste à relier un nombre n de clients par des tournées, commençant et finissant toutes à un même point défini, le dépôt. Ce problème est NP-complet, et dispose de nombreuses applications dans le monde d'aujourd'hui (notamment gestion d'un réseau routier). D'autant plus que ce problème dispose de nombreuses variantes (ajout d'une contrainte de temps, plusieurs dépôts possibles...). L'une des variantes les plus connues consiste à prendre en compte pour chaque client sa demande, de sorte à ce que les tournées créées ne dépassent pas une certaine capacité définie à l'avance. On nomme ce problème Capacitated Vehicle Routing Problem (CVRP).

Si de nombreuses heuristiques ont vu le jour pour résoudre ce problème, aucune d'entre elles ne parvient à trouver des solutions optimales pour toutes les instances de la littérature, malgré de très bons résultats dans la plupart des cas. Récemment [**?**], une nouvelle heuristique efficace a vu le jour. L'objectif de mon stage est de créer une *Learning Heuristic* afin de trouver de meilleures solutions.

Ce rapport commence par présenter le problème étudié, et introduit les notations et opérateurs utilisés dans la suite. Il présente ensuite l'objectif du stage et la méthode mise en place pour y parvenir. Différents problèmes sont alors soulevés, et sont traités dans chacune des parties qui suit.

1 Présentation des notions et notations

Cette partie introduit le problème étudié ainsi que les différentes notations utilisées ensuite dans le reste du rapport.

1.1 Description du problème

Le problème étudié ici (CVRP) est une extension du problème (VRP). Ces problèmes font partie de la famille des problèmes d'optimisation stochastique.

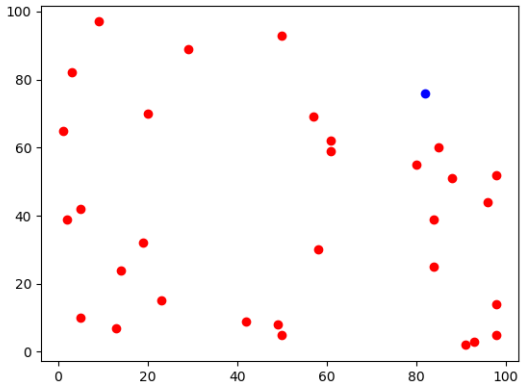

FIGURE 1 – Représentation de l'instance A-n32-k05 de la littérature

1.1.1 Problèmes d'optimisations

1.1.2 Vehicle Routing Problem (VRP)

Le problème de tournées de véhicules, est un problème NP-complet, où sont donnés n points de coordonnées (x_i, y_i), représentant 1 dépôt et $n-1$ clients. k véhicules sont disponibles. L'objectif est alors de minimiser la longueur du réseau (ensemble des tournées). On définit alors $x_{i,j}^v$ qui vaut 1 si j est desservi après i par le véhicule v, et 0 sinon. On définit également $c_{i,j}$ comme étant la distance entre i et j. Il faut donc déterminer la solution Sol vérifiant :

$$Sol = argmin_{Sol} \sum_{i=0}^{n} \sum_{j=0}^{n} \sum_{v=1}^{k} c_{i,j} x_{i,j}^v = argmin_{Sol} cost(Sol)$$

Les tournées créées doivent également respecter les contraintes suivantes :
— Chaque client doit être desservi par une et une seule tournée ;
— Chaque tournée doit partir et s'arrêter au dépôt.
Un exemple d'instance est présenté en figure 1, où les points rouges représentent les clients et le point bleu le dépôt. Une solution possible au problème est représenté en figure 2 mais n'est à priori pas optimale. De nombreux algorithmes ont vu le jour pour tenter de résoudre ce problème, ainsi que les nombreuses variantes qui existent (ajout de contraintes de capacité, temps ou longueur sur les tournées, ces contraintes sont cumulables). C'est l'ajout de capacité aux tournées qui nous intéressera plus particulièrement.

1.1.3 Capacitated VRP (CVRP)

On étend le VRP au CVRP en ajoutant à chaque client i une demande d_i, ainsi qu'une capacité C aux véhicules. Une nouvelle contrainte vient donc s'ajouter aux contraintes classiques du VRP :
— La demande totale sur chaque tournée ne doit pas excéder la capacité du véhicule.
Si on reprend l'instance A-n32-k05, en considérant les demandes des clients ainsi que la capacité disponible pour chaque véhicule, on obtient une solution présente sur la figure 3, qui n'est pas optimale. Ce problème est beaucoup étudié car il a de nombreuses applications (comme

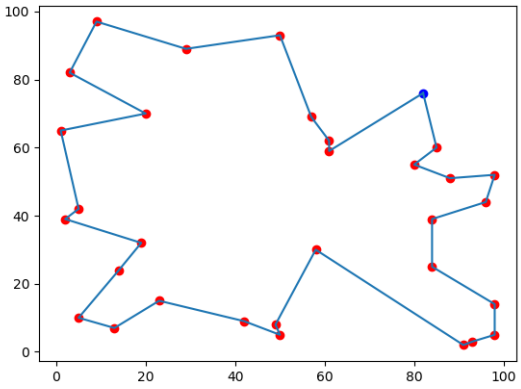

FIGURE 2 – Représentation d'une solution de l'instance A-n32-k05

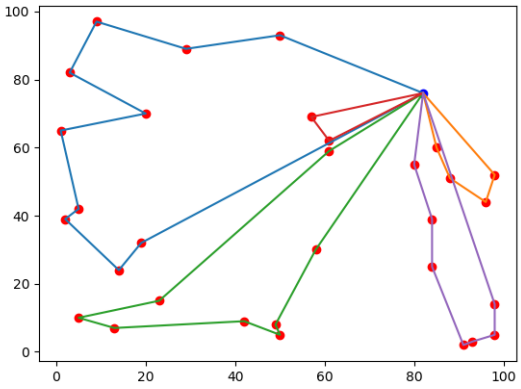

FIGURE 3 – Représentation d'une solution de l'instance A-n32-k05, où les demandes des clients sont prises en compte

par exemple la gestion du trafic routier, ou alors la gestion d'un réseau de bus), et peu de solutions optimales ont été trouvées pour des instances de plus de 500 clients.

1.2 Parcours et exploration des voisinages

Lorsqu'il s'agit de trouver une solution optimale à un problème, il est souvent intéressant d'explorer les voisinages d'une solution pour voir s'il n'y a pas mieux. Selon la méthode d'exploration employée, il peut être intéressant de parcourir le voisinage de différentes manières, pour ne pas toujours favoriser les mêmes voisins.

L'exploration d'un voisinage de solutions peut être plus ou moins exhaustif selon la condition d'arrêt utilisée. On distingue principalement, deux conditions d'arrêt lorsqu'il s'agit d'explorer des voisinages :

— First improvement (*FI*) : on parcourt le voisinage jusqu'à trouver un changement qui améliore la solution actuelle (on s'arrête donc à la première amélioration trouvée) ;
— Best improvement (*BI*) : on parcourt tout le voisinage, et on applique le changement

qui va le plus améliorer notre solution actuelle.

Pour explorer un voisinage, on peut le parcourir de différentes manières de sorte à ne pas toujours favoriser les mêmes voisins. On considérera ici 3 parcours différents :

— Dans l'ordre (*O*) : les voisins sont parcourus dans un ordre naturel (du premier au dernier) ;
— Dans un semi-ordre (*SO*) : on commence le parcours là où on s'était arrêté au dernier parcours, on parcourt ensuite les voisins dans l'ordre ;
— Aléatoirement (*RD*) : on tire aléatoirement l'ordre dans lequel on va parcourir les voisins.

On peut remarquer que peu importe le parcours effectué, pour faire une exploration *BI*, il faudra passer par tous les voisins. Pour qu'une exploration *FI* soit efficace, il faut éviter un parcours *O*, car dans ce cas on privilégie une certains voisinages qui seront choisis plus souvent. On retiendra le tableau récapitulatif suivant :

	BI	*FI*
O	Oui	Non
SO	Non	Oui
RD	Non	Oui

1.3 Motivation et objectif

L'objectif de mon stage d'intégrer de la connaissance à un algorithme d'optimisation utilisé pour résoudre CVRP, afin de le rendre plus performant. Une idée pour y parvenir serait de réussir à prédire des arêtes qui appartiendront à la solution optimale, en n'observant que des solutions initiales que l'on peut générer rapidement. On pourra ensuite exploiter ces arêtes pour construire une nouvelle solution. Nous adopterons la méthodologie suivante pour atteindre notre objectif :

— Comparer des solutions initiales à des solutions optimales pour des petites instances ;
— Établir de l'étude précédente des règles qui permettent de caractériser ces arêtes ;
— Exploiter les arêtes obtenues dans un algorithme d'optimisation.

Cette méthode, nous impose de résoudre les problèmes suivants : Comment construire une solution initiale de bonne qualité ? Quel algorithme d'optimisation utiliser ? Comment extraire la connaissance ? Comment intégrer la connaissance dans l'algorithme d'optimisation retenu ?

2 Construction d'une solution initiale de bonne qualité

Pour construire une solution initiale nous allons utiliser la dernière version d'un algorithme très répandu dans la littérature : l'algorithme Clarke & Wright [**?**].

2.1 Description de l'algorithme

L'algorithme Clarke & Wright (CW) est un algorithme glouton. Initialement chaque client est desservi par un véhicule (de cette manière la contrainte sur le nombre de véhicules disponibles n'est pas respectée). Ensuite les tournées sont fusionnées en fonction des *savings* calculées. On définit le *saving* des clients i et j de la manière suivante :

$$s(i,j) = c_{i0} + c_{0j} - \lambda c_{ij} + \mu |c_{i0} - c_{0j}| + \nu \frac{d_i + d_j}{\bar{d}}$$

Les paramètres (λ, μ, ν) jouent un rôle important dans la formule précédente, ce que nous verrons plus tard. (rôle des paramètres à détailler).

L'algorithme 1 présente le fonctionnement de l'algorithme CW.

Algorithm 1: CLARKE-WRIGHT calcule une solution initiale

Input: Un ensemble de points I, un ensemble d'entiers $D = d_1, ..., d_n$ et un triplet (λ, μ, ν) de flottants

Output: Une solution au problème I

```
1  for i ← 1 to n do
2      Sol ← Sol ∪ [0, i, 0]
3  Calculer les savings de toutes les arêtes
4  while max_{i,j} s(i,j) > 0 do
5      (i,j) ← argmax_{(i,j)} s(i,j)
6      r_i ← findRoute(Sol, i)
7      r_j ← findRoute(Sol, j)
8      if r_i et r_j peuvent fusionner then
9          Retirer r_i et r_j de Sol
10         Si possible fusionner r_i et r_j
11         Ajouter le résultat dans Sol et mettre s(i,j) = 0
12 return Sol
```

Exemple d'exécution de l'algorithme avec $(\lambda, \mu, \nu) = (1, 1, 1)$, sur l'instance A-n37-k06.

2.2 Choix des paramètres (λ, μ, ν)

3 Proposition d'un algorithme d'optimisation

4 Extraction de la connaissance

5 Intégration de la connaissance

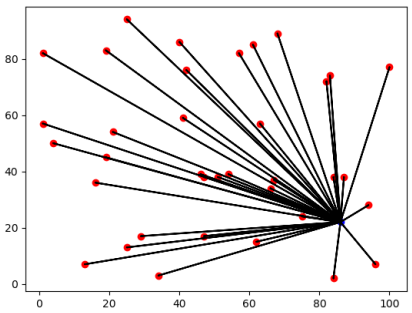

FIGURE 4 – Initialisation

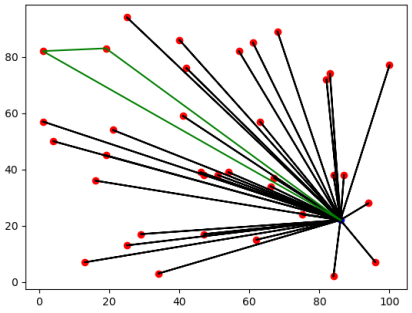

FIGURE 5 – 1^{ere} fusion

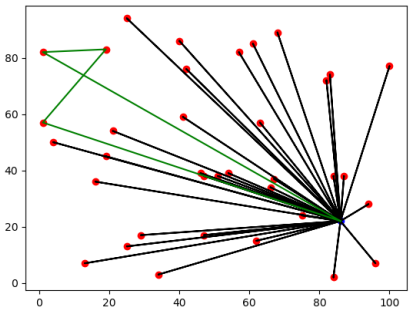

FIGURE 6 – 2^{eme} fusion

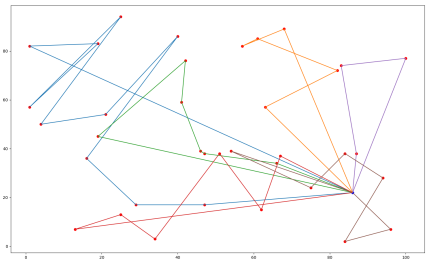

FIGURE 7 – Solution obtenue, $cost = 1297$