Rapport de stage

Clément Legrand-Lixon

25 juin 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 [2], une nouvelle heuristique efficace a vu le jour. L'objectif de mon stage est de s'inspirer de cette heuristique, et d'y intégrer de la connaissance pour rendre l'algorithme plus performant.

Ce rapport commence par présenter le problème étudié, et introduit les notations et opérateurs utilisés dans la suite. Il décrit ensuite comment a été mise en place l'intégration de connaissances au sein de l'algorithme, puis présente les résultats obtenus.

1 Présentations

1.1 Description du problème

1.1.1 Vehicle Routing Problem (VRP)

Le problème de tournées de véhicules, est un problème NP-complet, qui consiste à déterminer k tournées pour desservir l'ensemble des n clients présents. Ces tournées doivent toutes passer par un dépôt fixé par l'instance. Ainsi les tournées créées doivent respecter les règles suivantes :

— Chaque client doit être desservi par une et une seule tournée ;

— Chaque tournée doit partir et s'arrêter au dépôt.

L'objectif est alors de minimiser la longueur du réseau (ensemble des tournées). La distance euclidienne est privilégiée pour la majorité des instances. 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.

1.1.2 Capacitated VRP (CVRP)

L'une des extensions les plus étudiées du VRP, est celle où on rajoute une contrainte de capacités sur les tournées, cette contrainte étant la même pour toutes les tournées. Dorénavant, chaque client a une certaine demande, mais la demande présente sur une tournée ne doit pas excéder la capacité disponible sur la tournée. Les tournées doivent donc respecter la règle suivante, en plus de celles décrites à la section précédente :

— La demande totale sur chaque tournée ne doit pas excéder la capacité disponible.

Ce problème est beaucoup étudié car il a de nombreuses applications (comme le trafic routier), 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 Les constituants de l'algorithme

Cette partie décrit l'ensemble des briques utilisées pour construire l'algorithme. Ces briques dépendent du problème étudiée (ici CVRP), mais sont indépendantes entre elles. De fait, il est possible de construire de nombreux algorithmes en les empilant de différents manières.

1.3.1 Condition d'arrêt

Lors de la recherche d'une solution optimale d'un problème, il est indispensable de s'intéresser à la condition d'arrêt de l'heuristique utilisée. En effet, on doit trouver un compromis entre temps de calcul et qualité de la solution recherchée. On ne peut jamais être totalement sûr que la solution obtenue est bien optimale, mais on ne peut pas non plus explorer l'intégralité des solutions. Les principales conditions d'arrêt rencontrées sont les suivantes :

— Un nombre d'itérations à ne pas dépasser ;
— Un certain temps d'itérations sans solutions améliorantes à ne pas dépasser (3 minutes dans l'article [2]) ;
— Un nombre d'itérations sans solutions améliorantes à ne pas dépasser (de l'ordre de n^2 dans mon algorithme).

1.3.2 Initialisation

Cette brique consiste en la création d'une solution initiale, sur laquelle seront appliquées des modifications.

Cette solution peut être créée de différentes manières :

— Elle peut être générée aléatoirement (Alea) ;
— Elle peut être obtenue grâce à l'algorithme de Clarke & Wright (CW), avec les paramètres (λ, μ, ν) [1] :
  — Construire autant de tournées que de clients qui passent par le dépôt ;
  — Pour toutes les paires de clients (i, j) calculer le saving $s(i, j)$ via la formule :
  $$s(i,j) = c_{i0} + c_{0j} - \lambda c_{ij} + \mu |c_{i0} - c_{0j}| + \nu \frac{d_i + d_j}{\bar{d}}$$
  où c_{ij} dénote la distance entre les clients i et j (le client 0 étant le dépôt), et d_i dénote la demande du client i.
  — Considérer le couple (i, j) possédant le saving le plus élevée ;
  — Fusionner les tournées auxquelles appartiennent i et j, si possible (tournées différentes, et i et j doivent être premier et dernier clients de leur tournée) ;
  — Mettre $s(i, j) = 0$
  — Recommencer tant que le saving maximal n'est pas négatif.
— Elle peut être calculée grâce à l'intégration de connaissances (Learn) (objectif du stage).

1.3.3 Pire arête et pénalisation

A chaque tour de boucle, on calcule l'arête (i, j) qui maximise la fonction suivante (donnée dans l'article) :

$$b(i,j) = \frac{[\lambda_w w(i,j) + \lambda_c c(i,j)][\frac{d(i,j)}{max_{k,l} d(k,l)}]^{\frac{\lambda_d}{2}}}{1+p(i,j)}$$

où :

— $p(i, j)$ est la pénalisation de l'arête (i, j) (nombre de fois où l'arête a maximisé b) ;
— $w(i, j)$ est la largeur de l'arête (i, j) ;
— $c(i, j)$ est le coût de l'arête (i, j) ($c(i, j) = c_{ij}(1 + \lambda p(i, j)$, avec $\lambda = 0.1$ dans l'article) ;
— $d(i, j)$ est la profondeur de l'arête (i, j) (max de c_{0i} et c_{0j}) ;
— les paramètres $\lambda_w, \lambda_c, \lambda_d$, prennent comme valeurs 0 ou 1, selon les caractéristiques que l'on veut considérer. Il y a ainsi 6 fonctions de pénalisation différentes, que l'on peut choisir au cours de l'exécution.

C'est autour de l'arête calculée ici que vont s'orienter les recherches des opérateurs locaux qui suivent.

1.3.4 Ejection-Chain

Cet opérateur va essayer de déplacer au plus l clients sur des tournées plus adaptées. Dans l'article [2] $l = 3$. L'algorithme 1 décrit le fonctionnement de cet opérateur.

Algorithm 1: EJECTION-CHAIN applique l'opérateur ejection-chain

---

Input: Une arête (a, b), la liste des plus proches voisins des clients *voisins*, un entier l, la solution actuelle *sol*
Output: Une nouvelle solution au moins aussi bonne que *sol*

1 $initialCost \leftarrow cost(sol)$
2 $possibleSol \leftarrow sol$
3 $cand \leftarrow choose(a, b)$
4 $nextRoute \leftarrow findNextRoute(cand, voisins, possibleSol)$
5 $possibleSol \leftarrow$ déplacer *cand* après son voisin sur *nextRoute*
6 **for** $i \leftarrow 1$ ***to*** $l - 1$ **do**
7   *cand* $\leftarrow$un client de nextRoute différent de celui ajouté
8     $nextRoute \leftarrow findNextRoute(cand, voisins, possibleSol)$
9     $possibleSol \leftarrow$ déplacer *cand* après son voisin sur *nextRoute*
10 **if** $cost < cost(sol)$ **then**
11   **return** *sol*
12 **return** *possibleSol*

Aux lignes 3, 4, 7 et 8 de l'algorithme 1, il est possible d'utiliser les méthodes de la section 1.2 pour explorer les voisinages.

1.3.5 Cross-Exchange

Cet opérateur essaie d'échanger deux séquences de clients successifs entre deux tournées. Il est possible de limiter le nombre de clients par séquence échangée. L'algorithme 2 présente l'exécution de l'opérateur.

Algorithm 2: CROSS-EXCHANGE applique l'opérateur cross-exchange

---

Input: Une arête (c_1, c_2), la liste des plus proches voisins des clients *voisins*, la solution actuelle *sol*

Output: Une nouvelle solution au moins aussi bonne que *sol*

1 $initialCost \leftarrow cost(sol)$
2 $possibleSol \leftarrow sol$
3 $nextRoute \leftarrow findNextRoute(c_1, voisins, possibleSol)$
4 Considérer l'arête (c_3, c_4) de $nextRoute$, où c_4 est le proche voisin de c_1 utilisé
5 $possibleSol \leftarrow exchange(c_1, c_3, possibleSol)$
6 Choisir 2 clients c_5 et c_6 qui n'appartiennent pas à la même tournée
7 $possibleSol \leftarrow exchange(c_5, c_6, possibleSol)$
8 **if** $cost < cost(sol)$ **then**
9   **return** *sol*
10 **return** *possibleSol*

A la ligne 6 de l'algorithme 2, il est possible d'utiliser les méthodes de la section 1.2 pour explorer les voisinages, et choisir les clients à échanger.

1.3.6 Lin-Kernighan

L'heuristique Lin-Kernighan est utilisé en général pour résoudre le problème du voyageur de commerce (TSP). Il effectue une optimisation intra-tournée (c'est-à-dire que la tournée considérée est améliorée indépendamment des autres). Cela consiste en une réorganisation des clients sur la tournée. On choisit k tel que *LK* ne dépasse pas *k-opt* au cours de son exécution. On appelle *k-opt*, l'opération qui consiste à échanger k clients différents sur la tournée. D'après l'article [2], on peut prendre $k = 2$. L'algorithme 3 décrit l'exécution de l'opérateur.

Algorithm 3: LIN-KERNIGHAN applique l'opérateur Lin-Kernighan

---

Input: Une tournée r à améliorer

Output: Une permutation de r ayant un meilleur coût que r

$r_{next} \leftarrow 2\text{-}opt(r)$
while $r_{next} \neq r$ **do**
    $r \leftarrow r_{next}$
    $r_{next} \leftarrow 2\text{-}opt(r)$
return r

Lorsqu'il s'agit d'appliquer *2-opt*, il est possible d'utiliser les méthodes de la section 1.2 pour explorer les voisinages.

1.3.7 Raffinement

Il est possible que la solution actuelle puisse être améliorée simplement. En effet, la solution considérée peut contenir une tournée qui ne contient qu'un client, et les solutions optimales ne contiennent en général pas ce genre de tournées.

Une fonction reject vient donc compléter l'opérateur EC, et sert à supprimer les routes qui n'ont qu'un client. Pour cela elle essaie d'intégrer le client sur une tournée proche (en parcourant les plus proches voisins du client).

1.4 Algorithmes mis en œuvre

Nous présentons ici les algorithmes utilisés.

1.4.1 Heuristique d'Arnold et Sörensen

Cette heuristique se compose de la manière suivante :

Initialisation$_{CW}$
Condition d'arrêt : 3 minutes depuis la dernière amélioration
Compute worst edge
EC$_{BI-O}$
LK$_{BI-O}$
CE$_{BI-O}$
LK$_{BI-O}$
Itérations spéciales

Remarques : La brique itérations spéciales, contient des opérations qui ne sont effectuées qu'après un certain nombre d'itérations sans améliorations. En effet, il peut souvent arriver que la solution actuelle soit bloquée sur un optimum local. Les auteurs ont choisi les opérations suivantes (N désigne le nombre de clients) :

- $N/10$ itérations sans améliorations $\rightarrow$ Optimisation global ;
- $20N$ itérations sans améliorations $\rightarrow$ Changement fonction de pénalisation ;
- $100N$ itérations sans améliorations $\rightarrow$ Reset des pénalités.

Les auteurs calculent aussi à l'avance 30 plus proches voisins pour chacun des clients.

1.4.2 Algorithme utilisé

L'algorithme (A$_d$) que j'utilise se compose de la manière suivante :

Initialisation$_{CW}$
LK$_{BI-O}$
Condition d'arrêt : 1500 itérations depuis la dernière amélioration
Compute worst edge
EC$_{BI-O}$
LK$_{BI-O}$
CE$_{FI-O}$
LK$_{BI-O}$
Itérations spéciales

Remarques : Ici la brique itérations spéciales contient (N désigne le nombre de clients) :
- $N/2$ itérations sans améliorations $\rightarrow$ Retour à la dernière meilleure solution ;
- $5N/2$ itérations sans améliorations $\rightarrow$ Changement fonction de pénalisation et reset des pénalités ;

Un autre algorithme (A$_a$, où seul le bloc CE est modifié) :

Initialisation$_{CW}$
LK$_{BI-O}$
Condition d'arrêt : 1500 itérations depuis la dernière amélioration
Compute worst edge
EC$_{BI-O}$
LK$_{BI-O}$
CE$_{FI-RD}$
LK$_{BI-O}$
Itérations spéciales

Pour cet algorithme, j'effectue 25 itérations en gardant la moyenne des coûts obtenus, ainsi que la solution qui a donné le meilleur coût.

2 Intégration de connaissances

Références

[1] IK. Altinel and T. Öncan. A new enhancement of the clarke and wright savings heuristic for the capacitated vehicle routing problem. *Journal of the Operational Research Society*, 2005.

[2] Florian Arnold and Kenneth Sörensen. A simple, deterministic and efficient knowledge-driven heuristic for the vehicle routing problem. *Operations Research*, December 2017.