Rapport de stage (mise en place de l'algorithme)

Clément Legrand-Lixon

Ce papier présente l'algorithme mis en œuvre dans le cadre du problème CVRP (Capacitated Vehicle Routing Problem), qui s'inspire en grande partie de l'article d'Arnold et Sörensen.

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 connus 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 CVRP (pour Capacitated Vehicle Routing Problem).

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 [1], 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 introduire les notations et opérateurs utilisés dans la suite. L'algorithme commence par initialiser une solution. Il applique ensuite des opérateurs locaux, tant que la condition d'arrêt n'est pas respectée. Lors de certaines itérations, il peut effectuer des opérations spéciales. A la sortie de la boucle, il effectue un raffinement de la solution.

1 Présentations

1.1 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.

1. A simple, deterministic, and efficient knowledge-driven heuristic for the vehicle routing problem (Florian Arnold, Kenneth Sorensen)

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 peut importe le parcours effectué, pour faire une exploration BI, il faudra passer par tous les voisins. On retiendra le tableau récapitulatif suivant :

	BI	FI
O	Oui	Oui
SO	Non	Oui
RD	Non	Oui

1.2 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.

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

1.2.2 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) :
  — 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).

1.2.3 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.

1.2.4 Ejection-Chain

Cet opérateur va essayer de déplacer au plus l clients sur des tournées plus adaptées. Dans l'article $l = 2$.

Cet opérateur s'exécute de la manière suivante :
— Déterminer une arête à éliminer ;
— Considérer un des deux clients ;

— Regarder parmi les pp-voisins pour trouver une deuxième tournée ;
— Déplacer le client sur cette tournée (après le voisin trouvée) ;
— Essayer de déplacer un client de cette tournée sur une autre tournée (en répétant les étapes précédentes) ;
— Recommencer l'étape précédente l fois ;

Pour les étapes 2, 3 et 5, il est possible d'utiliser les méthodes de la section 1.1 pour explorer les voisinages.

1.2.5 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 (non implémenté).

Cet opérateur s'exécute de la manière suivante :
— Choisir une arête (c_1, c_2) à éliminer ;
— Trouver une autre tournée grâce aux pp-voisins de c_1 ;
— On note c_4 le voisin considéré, et c_3 le client précédent c_4 sur la nouvelle tournée ;
— On échange c_1 et c_3 ;
— On essaie d'échanger deux autres clients entre les deux tournées.

Pour les étapes 2 et 5, il est possible d'utiliser les méthodes de la section 1.1 pour explorer les voisinages.

1.2.6 Lin-Kernighan

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. D'après l'article on peut prendre $k = 2$.

LK va donc s'exécuter de la manière suivante :
— On considère la tournée à améliorer ;
— On applique 2-opt sur la tournée (on échange deux clients sur la tournée) ;
— Tant qu'on a des améliorations on applique 2-opt sur la tournée ;
— On renvoie la tournée améliorée.

Pour l'étape 2, il est possible d'utiliser les méthodes de la section 1.1 pour explorer les voisinages.

1.2.7 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 pp-voisins du client).

1.3 Algorithmes mis en œuvre

Nous présentons ici les algorithmes utilisés.

1.3.1 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.

1.3.2 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.

Temps d'exécution moyen des algos ? En théorie, avec un nombre suffisant d'itérations, le meilleur résultat donné par A_a est au moins aussi bon que celui donné par A_d, mais cela implique un plus grand temps de calcul.

Au vu de l'efficacité de l'algo A& S montrée dans l'article, il serait peut-être plus judicieux de partir de l'algo A_d qui s'en rapproche le plus. Par ailleurs, en regardant les premiers résultats obtenus, il semblerait que l'algo A_{dd} soit plus adapté pour l'intégration des connaissances, puisqu'il donne de meilleurs résultats jusqu'à maintenant.