Rapport de stage

Clément Legrand-Lixon

3 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 [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 Présentation des notions et notations

1.1 Description du problème

1.1.1 Problèmes d'optimisations

1.1.2 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.

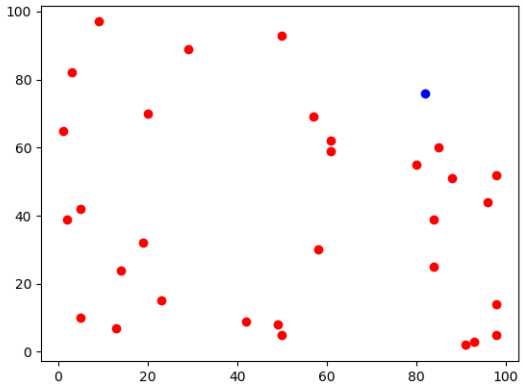

FIGURE 1 – Représentation de l'instance A-n32-k05 de la littérature

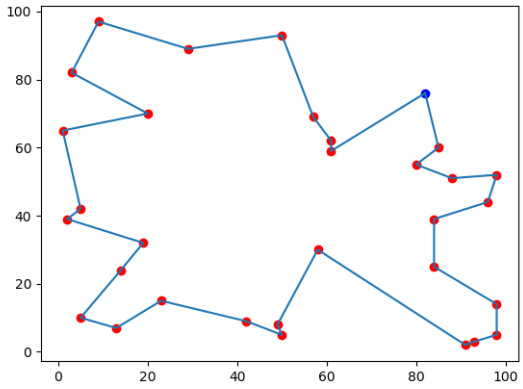

FIGURE 2 – Représentation d'une solution de l'instance A-n32-k05

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

L'une des extensions les plus étudiées du VRP, est celle où l'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 celle-ci. 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.

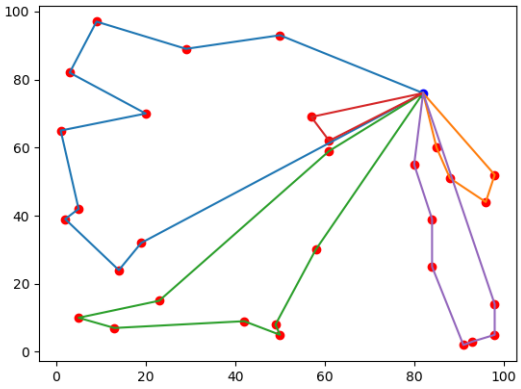

FIGURE 3 – Représentation d'une solution de l'instance A-n32-k05, où les demandes des clients sont prises en compte

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 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 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 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

Avant de pouvoir appliquer l'heuristique, il faut pouvoir déterminer une solution initiale sur laquelle les modifications vont être appliquées.

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) (algorithme 1), avec les paramètres (λ, μ, ν) [1]. Altinel a aussi montré [1] que l'on pouvait se contenter de choisir les paramètres dans l'intervalle $[0, 2]$ pour trouver de bons résultats. Les savings se calculent via la formule suivante :
$$s(i,j) = c_{i0} + c_{0j} - \lambda c_{ij} + \mu |c_{i0} - c_{0j}| + \nu \frac{d_i + d_j}{\bar{d}}$$
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.
L'intérêt de cet algorithme est qu'il donne une bonne solution initiale en peu de temps, ce qui permet de l'exécuter un grand nombre de fois rapidement.

— Elle peut être calculée grâce à l'intégration de connaissances (Learn). C'est ce dernier point que nous tenterons de mettre en œuvre par la suite.

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

```
for i ← 1 to n do
    Sol ← Sol ∪ [0, i, 0]
Calculer les savings de toutes les arêtes
while le saving maximal est positif do
    (i, j) ← argmax_(i,j) s(i, j)
    r_i ← findRoute(Sol, i)
    r_j ← findRoute(Sol, j)
    if r_i et r_j peuvent fusionner then
        Retirer r_i et r_j de Sol
        Fusionner r_i et r_j
        Ajouter le résultat dans Sol et mettre s(i, j) = 0
return Sol
```

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 [2]) ;
— $d(i, j)$ 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 (on ne considère pas le cas où $\lambda_w = \lambda_c = 0$, puisqu'il fournit $b(i, j) = 0$.
Les notions de largeur, profondeur et coût sont illustrées par la figure 4. Plus précisément, la largeur est la différence de longueur entre les projetés de i et j sur la droite issue du dépôt passant par le centre de gravité de la tournée. Le centre de gravité d'une tournée étant obtenu en faisant la moyenne, pour chaque composante, des points de cette tournée.

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

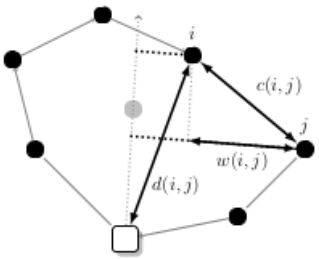

FIGURE 4 – Illustration des caractéristiques d'une arête

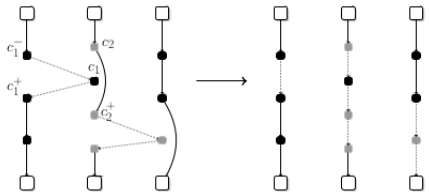

FIGURE 5 – Exemple de fonctionnement de l'opérateur ejection-chain

1.3.4 Ejection-Chain

Cet opérateur va essayer de déplacer au plus l clients sur des tournées plus adaptées. Le fonctionnement de cet opérateur est présenté sur la figure 5.

Dans l'article [2] $l = 3$. En effet l'algorithme 2, qui décrit le fonctionnement de cet opérateur, s'exécute en $O(n^{l-1})$. Il vaut donc mieux choisir une valeur de l assez petite, pour que la complexité n'explose pas.

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

Aux lignes 3, 4, 7 et 8 de l'algorithme 2, il est possible d'utiliser les méthodes de la section 1.2 pour explorer les voisinages.

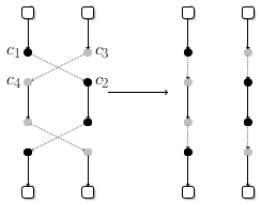

Figure 1: Illustration of the CROSS-exchange with sequences of two customers.

FIGURE 6 – Exemple de fonctionnement de l'opérateur cross-exchange

1.3.5 Cross-Exchange

Cet opérateur essaie d'échanger deux séquences de clients successifs entre deux tournées. Le fonctionnement de cet opérateur est présenté sur la figure 6.

Il est possible de limiter le nombre de clients par séquence échangée. L'algorithme 3 présente l'exécution de l'opérateur et s'exécute en $O(n^2)$.

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

A la ligne 6 de l'algorithme 3, 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). Un exemple d'utilisation de cet opérateur est présent sur la figure

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. On commence alors par appliquer 2-opt, si une amélioration est trouvée, on passe à 3-opt, et ainsi de suite jusqu'à atteindre k-opt. On repart alors de 2-opt, et ce jusqu'à ne plus trouver d'améliorations. D'après l'article [2], on peut prendre $k = 2$. L'algorithme 4 décrit l'exécution de l'opérateur.

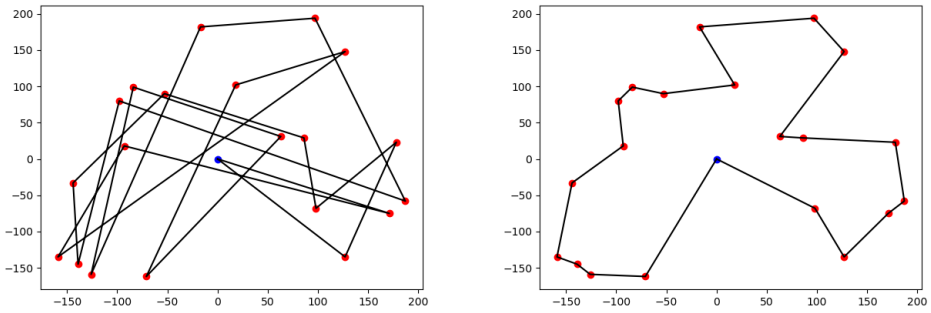

FIGURE 7 – Exemple de fonctionnement de l'opérateur LK

Algorithm 4: LIN-KERNIGHAN applique l'opérateur Lin-Kernighan

Input: Une tournée r à améliorer
Output: Une permutation de r ayant un meilleur coût que r

```
1 r_next ← 2-opt(r)
2 while r_next ≠ r do
3 |   r ← r_next
4 |   r_next ← 2-opt(r)
5 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.

1.4 Algorithmes mis en œuvre

Les algorithmes présentés précédemment peuvent être enchaînés dans l'ordre que l'on veut. En effet, seule le résultat obtenu importe, les algorithmes sont donc indépendants. Nous présentons ici l'algorithme utilisé par Arnold et Sörensen dans [2], ainsi que l'algorithme que nous utiliserons pour la suite.

1.4.1 Heuristique d'Arnold et Sörensen

L'heuristique décrite par Arnold et Sörensen est présentée dans l'algorithme 5. Elle est à la fois déterministe et efficace, c'est-à-dire qu'elle trouve de bonnes solutions pour de nombreuses instances en un temps raisonnable. Les auteurs pré-calculent également les 30 plus proches voisins de chacun des clients.

1.4.2 Algorithme utilisé

Il est possible de construire de nombreuses variantes de l'algorithme précédent, selon l'ordre dans lequel on exécute les opérateurs. Après avoir testé quelques possibilités, c'est finalement l'algorithme 6 que nous retiendrons. Il permet en effet le parcours de nombreux voisinages, puisqu'ils sont choisis aléatoirement. Mais le temps d'exécution est plus long, car

Algorithm 5: AS applique l'heuristique A& S au problème considéré

Input: Un ensemble de points I, les demandes des clients D, un triplet de flottants (λ, μ, ν)

Output: Une solution au problème I

$Sol \leftarrow CW(I, D, \lambda, \mu, \nu)$
$N \leftarrow length(D)$
$nextSol \leftarrow Sol$
while *La dernière amélioration date de moins de 3 min* **do**
  $worstEdge \leftarrow argmax_{(i,j)} b(i, j)$
  $nextSol \leftarrow EC_{BI-O}(worstEdge, I, D)$
  $nextSol \leftarrow LK_{BI-O}(nextSol)$
  $nextSol \leftarrow CE_{BI-O}(worstEdge, I, D)$
  $nextSol \leftarrow LK_{BI-O}(nextSol)$
  if $cost(Sol) > cost(nextSol)$ **then**
    $Sol \leftarrow nextSol$
  if *Pas d'améliorations depuis* $N/10$ *itérations* **then**
    Appliquer les opérateurs sur toutes les arêtes de la solution
  if *Pas d'améliorations depuis* $20N$ *itérations* **then**
    Changer de fonction de pénalisation en prenant un autre triplet $(\lambda_w, \lambda_c, \lambda_d)$
  if *Pas d'améliorations depuis* $100N$ *itérations* **then**
    Réinitialiser les pénalités des arêtes
return *Sol*

les solutions obtenues étant aléatoires, nous exécutons 20 fois l'algorithme. Cela permet d'obtenir le coût moyen d'une solution renvoyée, et nous retenons la meilleure solution trouvée lors des 20 exécutions.

2 Utilisation de connaissances

Maintenant que nous disposons d'un algorithme performant pour calculer des solutions aux instances considérées, nous allons essayer d'extraire de la connaissance à partir des solutions initiales fournies par Clarke-Wright, pour l' intégrer ensuite à l'algorithme A.

2.1 Motivation

L'intérêt d'intégrer des connaissances dans un algorithme est de le rendre plus performant, autant en efficacité qu'en rapidité. L'algorithme A & S, est décrit comme l'un des plus performants algorithme à ce jour pour résoudre les problèmes liés à VRP. En effet, il est initialement destiné aux problèmes CVRP, mais peut très bien s'adapter à d'autres problèmes [2], comme le *multi-depot VRP* (MDVRP), qui est un problème VRP où plusieurs dépôts sont disponibles. Il serait donc intéressant de pouvoir intégrer de la connaissance à cet algorithme pour le rendre encore plus performant.

Algorithm 6: H_c calcule une solution du problème considéré

Input: Un ensemble de points I, les demandes des clients D, un triplet de flottants (λ, μ, ν)

Output: Une solution au problème I

$Sol \leftarrow CW(I, D, \lambda, \mu, \nu)$
$N \leftarrow length(D)$
$nextSol \leftarrow Sol$
while *Pas 1 minute depuis la dernière amélioration* **do**
  $worstEdge \leftarrow argmax_{(i,j)} b(i, j)$
  $nextSol \leftarrow EC_{FI-RD}(worstEdge, I, D)$
  $nextSol \leftarrow LK_{BI-O}(nextSol)$
  $nextSol \leftarrow CE_{FI-RD}(worstEdge, I, D)$
  $nextSol \leftarrow LK_{BI-O}(nextSol)$
  if $cost(Sol) > cost(nextSol)$ **then**
    $Sol \leftarrow nextSol$
  if *Pas d'améliorations depuis* $N/2$ *itérations* **then**
    $nextSol \leftarrow Sol$
  if *Pas d'améliorations depuis* $2N$ *itérations* **then**
    Changer de fonction de pénalisation en prenant un autre triplet $(\lambda_w, \lambda_c, \lambda_d)$
    Réinitialiser les pénalités des arêtes
return *Sol*

2.2 Extraction des connaissances

L'objectif lors de l'extraction des connaissances est de pouvoir retrouver un maximum d'arêtes qui appartiennent à la solution optimale que l'on recherche. Cette étape ne doit pas être trop longue par rapport à l'exécution de l'heuristique, pour que cela reste exploitable. Un choix évident a semblé être l'utilisation de l'algorithme de Clarke and Wright pour générer des solutions rapidement, et ainsi construire des bases d'apprentissage.

2.2.1 Construction de la base de départ

Cette base a pour objectif de représenter au mieux les solutions trouvées lors de l'exécution de CW. Générer les solutions pour tous les couples (λ, μ, ν) prendrait trop de temps, c'est pourquoi nous nous contenterons de tirer NB couples (λ, μ, ν) aléatoirement (de manière uniforme), pour créer notre base.

2.2.2 Construction de la base d'apprentissage

A priori, toutes les solutions stockées dans la base de départ ne sont pas bonnes à prendre. Il faut donc extraire un ensemble de solutions que l'on juge utile pour représenter la connaissance et ainsi pouvoir l'utiliser par la suite. Deux méthodes ont été retenues pour extraire ces solution :

— On conserve x% des meilleures solutions de la base de départ. C'est la quantité qui est privilégiée ici. On appellera cette base Quantity$_x$;

— On conserve les solutions qui ont un coût inférieur à $cot_{min} + (cot_{max} - cot_{min})\frac{x}{100}$. On privilégie ici la qualité. On appellera cette base Quality_x.

A présent que nous disposons d'une base d'apprentissage, nous allons pouvoir en extraire des connaissances.

2.2.3 Récupération des arêtes jugées importantes

2.3 Intégration des connaissances

décrire nouvel algo + résultats

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. *Transportation Science*, December 2017.