# Création d'une *Learning Heuristic* pour résoudre le *Capacitated Vehicle Routing Problem*

Clément Legrand-Lixon

9 juillet 2018

**Résumé**

Les problèmes d'optimisation sont des problèmes difficiles à résoudre de manière exacte. Le *Capacitated Routing Vehicle Problem* est l'un de ces problèmes, et est très étudié dans la littérature. Pour approcher les solutions optimales, différents types d'algorithme existent. Des méthodes d'extraction de connaissance voient le jour afin d'améliorer les performances d'algorithme existant.

## 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 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 Optimisation combinatoire

Un problème d'optimisation combinatoire (également appelée optimisation discrète), consiste à trouver dans un ensemble discret les meilleures solutions réalisables. La notion de meilleure solution étant définie par une fonction objectif. Formellement, on a :

— Un ensemble discret $N$ ;
— Une fonction $f : 2^N \to \Re$, dite fonction objectif ;
— Un ensemble $R$ de sous-ensembles de $N$, dont les éléments sont appelés solutions réalisables.

Ainsi, un problème d'optimisation combinatoire consiste à déterminer :

$$max_{S \subseteq N}\{f(S), S \in R\}$$

Beaucoup trop de solutions

## 1.2 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.

### 1.2.1 Problèmes d'optimisations

### 1.2.2 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.

### 1.2.3 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 :

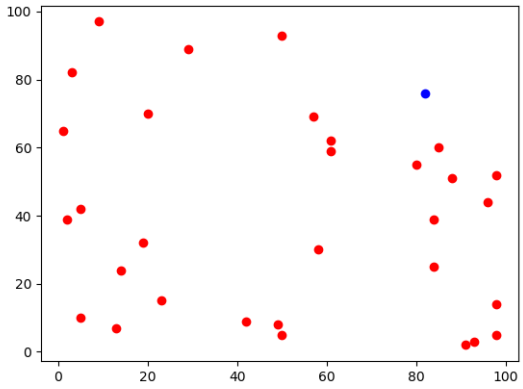


FIGURE 1 – Représentation de l'instance A-n32-k05 de la littérature

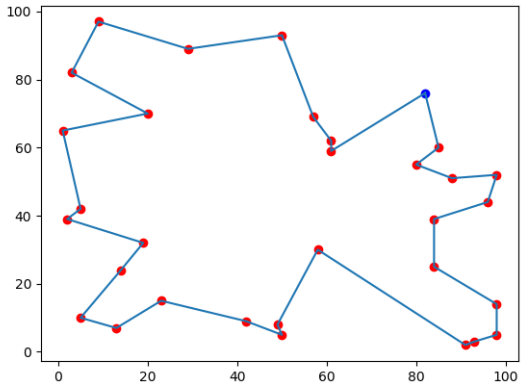


FIGURE 2 – Représentation d'une solution de l'instance A-n32-k05

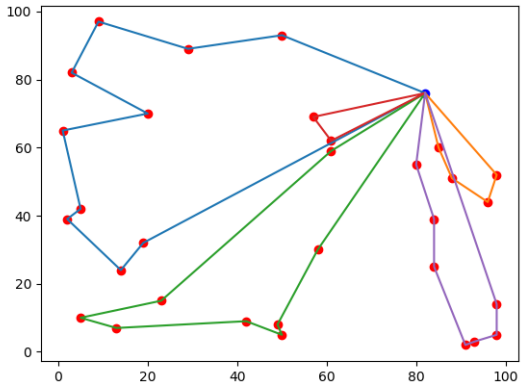


FIGURE 3 – Représentation d'une solution de l'instance A-n32-k05, où les demandes des clients sont prises en compte

— 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 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.

### 1.3 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 |

### 1.4 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 [1].

### 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 $(\lambda, \mu, \nu)$ 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 $(\lambda, \mu, \nu)$ 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, représenté sur les figures 4 à 7. On remarque sur la figure 7 que l'on pourrait améliorer la solution rien qu'en réorganisant les différentes tournées, pour minimiser leur coût.

### 2.2 Choix des paramètres $(\lambda, \mu, \nu)$

Le triplet $(\lambda, \mu, \nu)$ a déjà été étudié de nombreuses fois dans la littérature. L'article [1] précise qu'il suffit de considérer $(\lambda, \mu, \nu)$ dans $]0, 2] \times [0, 2]^2$ pour avoir de bonnes solutions. Par ailleurs, il est inutile de prendre précision inférieure au dixième lorsqu'on choisit les valeurs des paramètres. Les figures 8 à 10 représentent différents résultats obtenus pour des triplets $(\lambda, \mu, \nu)$ différents. On remarque qu'il n'y a aucun lien entre les résultats et les valeurs de $(\lambda, \mu, \nu)$. On ne peut donc pas prévoir à l'avance si le triplet $(\lambda, \mu, \nu)$ va donner un bon résultat ou non. L'influence de ces paramètres dépend aussi des caractéristiques de l'instance considérée, ainsi on ne peut pas se restreindre au choix d'un triplet qui conviendrait pour toutes les instances.

## 3 Proposition d'un algorithme d'optimisation

Nous proposons dans cette partie un algorithme d'optimisation qui sera utilisé pour y intégrer de la connaissance. Il s'inspire d'un algorithme proposé récemment [2], dont nous détaillons le fonctionnement.

### 3.1 Heuristique Arnold & Sörensen

L'heuristique proposée par Arnold et Sörensen, est à la fois simple et efficace. Il semble donc pertinent de vouloir améliorer cet algorithme en y intégrant de la connaissance. L'heu-

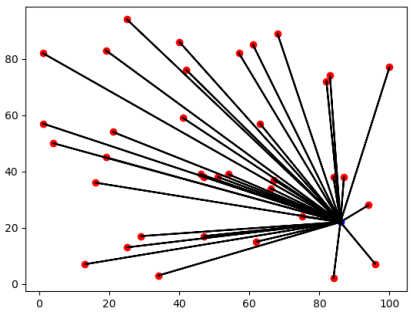


FIGURE 4 – Initialisation

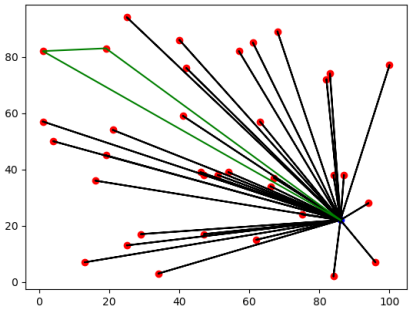


FIGURE 5 – $1^{ere}$ fusion

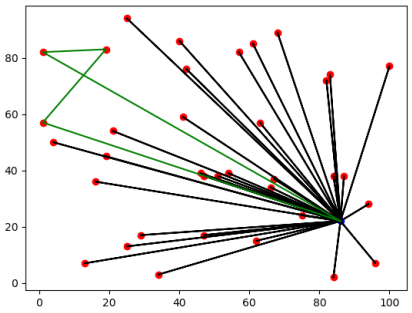


FIGURE 6 – $2^{eme}$ fusion

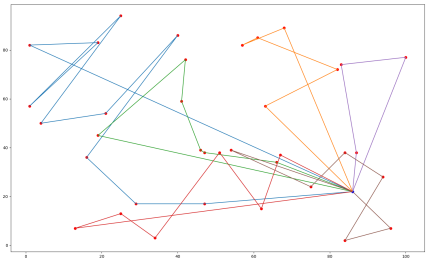

FIGURE 7 – Solution obtenue, $cost = 1297$

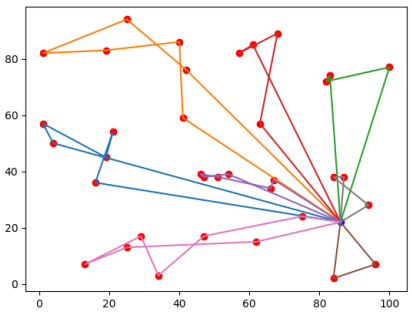


FIGURE 8 – $(1.9, 0.1, 1.5), cost = 1106$

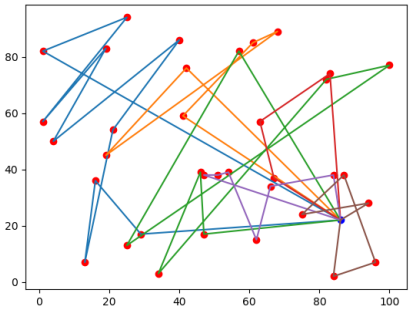


FIGURE 9 – $(0.1, 0.1, 0.1), cost = 1569$

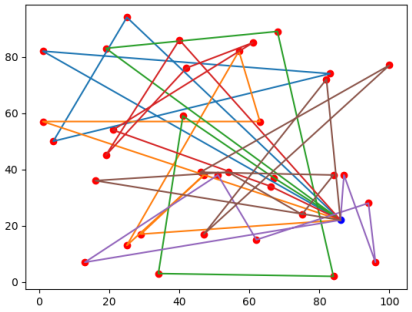


FIGURE 10 – $(0.0, 1.0, 1.5), cost = 2191$

ristique commence par déterminer une solution initiale via l'algorithme CW, présenté en section 2. Différents opérateurs de voisinage sont ensuite appliqués autour d'une arête, considérée comme étant la pire du graphe. L'algorithme 2 donne le fonctionnement de l'heuristique (A&S).

**Algorithm 2:** AS applique l'heuristique A& S au problème considéré

**Input:** Un ensemble de points I, les demandes des clients D, un triplet de flottants $(\lambda, \mu, \nu)$

**Output:** Une solution au problème I

$Sol \leftarrow CW(I, D, \lambda, \mu, \nu)$
$N \leftarrow Size(D)$
$nextSol \leftarrow Sol$
**while** *La dernière amélioration date de moins de 3 min* **do**
  $worstEdge \leftarrow$ Calcul de la pire arête
  $nextSol \leftarrow EC_{BI-O}(worstEdge, I, D)$
  $nextSol \leftarrow LK_{BI-O}(nextSol)$
  $nextSol \leftarrow CE_{BI-O}(worstEdge, I, D)$
  $nextSol \leftarrow LK_{BI-O}(nextSol)$
  **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 $(\gamma_w, \gamma_c, \gamma_d)$
  **if** *Pas d'améliorations depuis* $100N$ *itérations* **then**
    Réinitialiser les pénalités des arêtes
**return** *Sol*

Les prochaines sections détaillent le calcul de la pire arête, ainsi que le fonctionnement des opérateurs utilisés.

### 3.1.1 Pire arête et pénalisation

Afin de pouvoir comparer les différentes arêtes entre elles et déterminer laquelle est la pire, il faut disposer de certaines métriques sur les arêtes pour pouvoir les caractériser.

Trois métriques sont détaillées dans l'article [2] :

— Le coût d'une arête $(i, j)$, que l'on note $c(i, j)$ se calcule de la manière suivante :
$$c(i, j) = c_{ij}(1 + \beta p(i, j)$$
Dans l'article [2]) $\lambda = 0.1$. $p(i, j)$ correspond au nombre de fois où l'arête $(i, j)$ a été pénalisé ;

— La profondeur d'une arête $(i, j)$, noté $d(i, j)$ a pour formule :
$$\max(c_{0i}, c_{0j})$$
Autrement dit c'est la distance entre le point le plus éloigné du dépôt et le dépôt.

— La largeur de l'arête $(i, j)$, noté $w(i, j)$ 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,

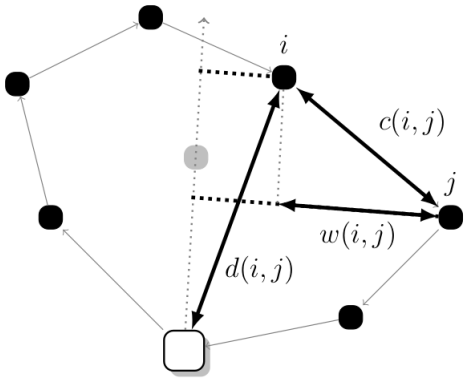


FIGURE 11 – Illustration des caractéristiques d'une arête

des points de cette tournée.
Les notions de coût, profondeur et largeur sont illustrées par la figure 11.

On définit alors la fonction de pénalisation $b$ de la manière suivante :

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

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 $b(i,j) = 0$.

On peut alors définir ce qu'est la pire arête $(i^*, j^*)$ du graphe :

$$(i^*, j^*) = argmax_{i,j} b(i,j)$$

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

### 3.1.2 Ejection-Chain

Le premier opérateur utilisé est appelé Ejection-Chain. Son objectif est de déplacer au plus $l$ clients sur des tournées.

Le fonctionnement de cet opérateur est présenté sur la figure 12.

Dans l'article [2] $l = 3$. En effet l'algorithme 3, 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.

Aux lignes 3, 4, 7 et 8 de l'algorithme 3, il est possible d'utiliser les méthodes de la section 1.3 pour explorer les voisinages.

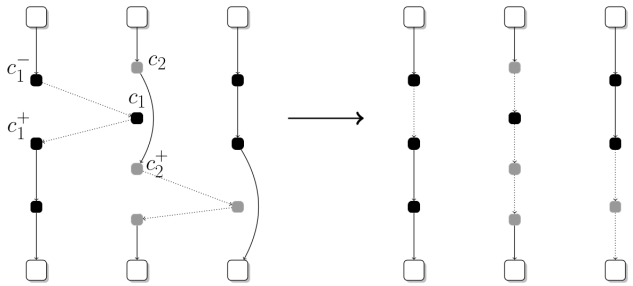


FIGURE 12 – Exemple de fonctionnement de l'opérateur ejection-chain

**Algorithm 3:** 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*

```
possibleSol ← sol
cand ← choose(a, b)
nextRoute ← findNextRoute(cand, voisins, possibleSol)
possibleSol ← déplacer cand après son voisin sur nextRoute
for i ← 1 to l − 1 do
    cand ←un client de nextRoute différent de celui ajouté
    nextRoute ← findNextRoute(cand, voisins, possibleSol)
    possibleSol ← déplacer cand après son voisin sur nextRoute
if cost(possibleSol) < cost(sol) then
    sol ← possibleSol
return sol
```

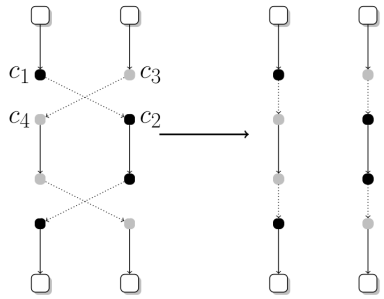


FIGURE 13 – Exemple de fonctionnement de l'opérateur cross-exchange

### 3.1.3 Cross-Exchange

Un deuxième opérateur utilisé est le Cross-Exchange. Son objectif est d'échanger deux séquences de clients entre deux tournées. Le fonctionnement de cet opérateur est présenté sur la figure 13.

Il est possible de limiter le nombre de clients par séquence échangée. L'algorithme 4 présente l'exécution de l'opérateur et s'exécute en $O(n^2)$.

**Algorithm 4:** 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 4, il est possible d'utiliser les méthodes de la section 1.3 pour explorer les voisinages, et choisir les clients à échanger.

### 3.1.4 Lin-Kernighan

Le dernier opérateur utilisé est l'heuristique Lin-Kernighan. Elle a été créée 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. 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 $k = 2$.

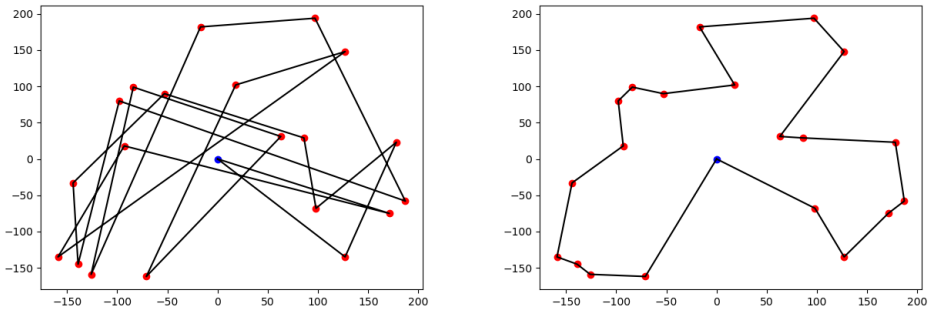

FIGURE 14 – Exemple de fonctionnement de l'opérateur LK

Un exemple d'utilisation de cet opérateur est présent sur la figure 14

L'algorithme 5 décrit l'exécution de l'opérateur.

**Algorithm 5:** 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 1.3 pour explorer les voisinages.

### 3.2 Algorithme d'optimisation utilisé

L'algorithme d'optimisation que nous utilisons est un peu différent de l'heuristique A&S précédente. Pour diminuer le temps d'exécution, nous choisissons de diminuer le temps limite entre deux nouvelles solutions à $\frac{n}{3}$ secondes. Pour que l'exploration du voisinage des solutions soit plus efficace, nous décidons de passer les opérateurs *EC* et *CE* en mode $FI - RD$ (ce qui est généralement le cas dans les algorithmes d'optimisation). Enfin, nous ajoutons une condition de *Restart*, s'il n'y a pas eu d'améliorations depuis $n$ itérations.

L'algorithme 6 présente en rouge les changements effectués.

Nous pouvons à présent nous intéresser à l'extraction des connaissances des solutions initiales.

## 4 Extraction de la connaissance

Cette section s'intéresse à l'extraction de connaissance à partir de solutions initiales générées.

**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 $(\lambda, \mu, \nu)$

**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 n/3 sec* **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 itérations* **then**
    $nextSol \leftarrow Sol$
  **if** *Pas d'améliorations depuis n itérations* **then**
    Changer de fonction de pénalisation en prenant un autre triplet $(\gamma_w, \gamma_c, \gamma_d)$
    Réinitialiser les pénalités des arêtes
**return** *Sol*

## 4.1 Quelle est la connaissance ?

En observant quelques solutions obtenues avec CW, et application de l'opérateur LK, on remarque que plus la solution initiale est bonne et plus elle possède d'arêtes en commun avec la solution optimale. Ce résultat est illustré sur les figures 15 à 17, où les arêtes optimales sont vertes.

Il faudrait donc pouvoir déterminer à l'avance les arêtes optimales à partir des solutions initiales fournies par $CW + LK$.

## 4.2 Protocole d'apprentissage

Pour extraire cette connaissance, nous allons devoir créer un échantillon de solutions initiales, à partir duquel nous allons extraire une base d'apprentissage. Cette base va nous permettre d'extraire des arêtes.

### 4.2.1 Génération de l'échantillon

Puisque l'on ne peut pas prédire les paramètres $(\lambda, \mu, \nu)$ pour obtenir de bonnes solutions, nous allons devoir en générer un certain nombre :

— Soit on génère l'intégralité des solutions initiales possibles, en parcourant tous les triplets $(\lambda, \mu, \nu)$ (ie génération de 8820 solutions), pour obtenir l'échantillon ;
— Soit on tire $N$ triplets $(\lambda, \mu, \nu)$ aléatoirement, et les solutions obtenus constitueront notre

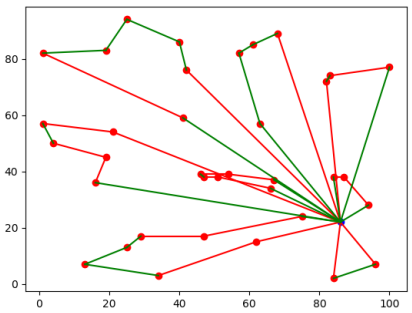


FIGURE 15 – $CW(1.9, 0.1, 1.5) + LK$, $cost = 1041$, 19 arêtes optimales

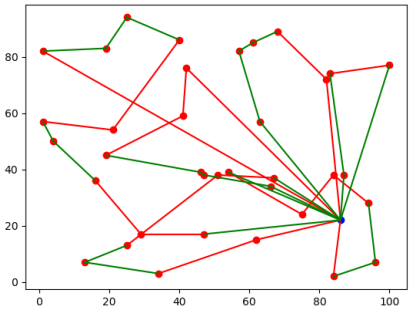


FIGURE 16 – $CW(0.1, 0.1, 0.1) + LK$, $cost = 1170$, 19 arêtes optimales

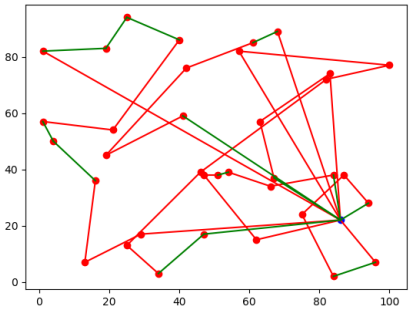


FIGURE 17 – $CW(0.0, 1.0, 1.5) + LK$, $cost = 1600$, 11 arêtes optimales

échantillon.

Les solutions qui constituent notre échantillon ne sont pas nécessairement toutes différentes.

#### 4.2.2 Construction de la base d'apprentissage

#### 4.2.3 Extraction des arêtes

### 4.3 Résultats

## 5 Intégration de la connaissance

### 5.1 Où intégrer la connaissance ?

### 5.2 Learning Heuristic

### 5.3 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.