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

Clément Legrand-Lixon

10 juillet 2018

Résumé

De nombreux algorithmes font régulièrement leur apparition pour tenter de résoudre des problèmes d'optimisation stochastique, ou d'améliorer les solutions existantes. Dans la plupart des problèmes, il est possible d'extraire de la connaissance de petites instances résolues, afin d'aider les algorithmes lors de la résolution de plus grandes instances. Intégrer de cette manière de la connaissance à un algorithme permet de créer une *Learning Heuristic*, plus performante que l'algorithme initial.

Introduction

Les problèmes d'optimisation stochastique sont généralement des problèmes difficiles à résoudre, puisque le nombre de solutions à tester varie de manière exponentielle en la taille de l'instance. Le Vehicle Routing Problem (VRP) est un problème dit d'optimisation stochastique. L'objectif est de relier un nombre n de clients par des véhicules, démarrant et finissant tous à un même point défini, le dépôt. Ce problème est NP-complet, et dispose de nombreuses variantes (ajout d'une contrainte de temps, plusieurs dépôts possibles...). Cela permet de modéliser un grand nombre de situations réelles. 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).

De nombreux types d'algorithme existent pour résoudre ces problèmes d'optimisation (algorithme génétique, colonie de fourmis...), mais ne parviennent pas toujours à trouver la solution optimale. De nombreuses heuristiques ont également vu le jour pour résoudre le CVRP, 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. Encore récemment [2], une nouvelle heuristique efficace a vu le jour. L'une des méthodes pour améliorer les algorithmes d'optimisation consiste en l'extraction de connaissances sur des petites instances résolues, puis leur intégration dans l'algorithme d'optimisation choisi.

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

1.1 Description du problème

De nombreux problèmes sont dits d'optimisation stochastique, dont font notamment partie le *Traveller Salesman Problem* (TSP), ainsi que le *Capacitated Vehicle Problem* (CVRP). Cette partie détaille de manière formelle ce qu'est un problème d'optimisation stochastique, puis présente le problème étudié : *CVRP*

1.1.1 Optimisation stochastique

Un problème d'optimisation combinatoire (également appelée optimisation discrète), consiste à trouver dans un ensemble discret, les meilleures solutions (au sens d'une certaine fonction, dite *fonction objectif*) réalisables. Formellement, on a :

— Un ensemble discret N ;
— Une fonction $f : 2^N \rightarrow \Re$, 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 :

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

Toutefois dans ce type de problème, le nombre de solutions réalisables varie généralement de manière exponentielle selon la taille du problème. Cela rend donc impossible une énumération complète des solutions réalisables, d'où la difficulté de trouver la solution optimale à ce type de problème.

1.1.2 Vehicle Routing Problem (VRP)

Le problème de tournées de véhicules, est un problème NP-complet d'optimisation stochastique, où sont donnés n points de coordonnées (x_i, y_i), représentant 1 dépôt et $n-1$ clients. On dispose également d'une flotte de k véhicules. L'objectif est de minimiser la longueur du réseau (ie l'ensemble des tournées effectuées par les véhicules). 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 :

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

La fonction *cost* correspond à la fonction objectif de ce problème.

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.

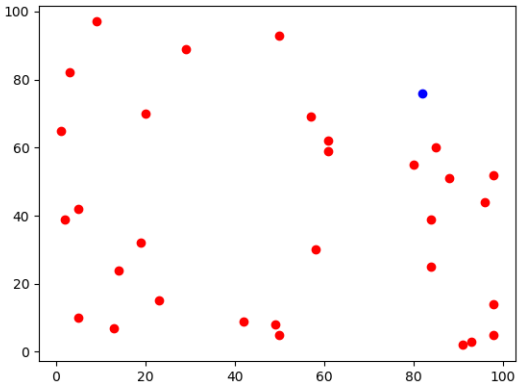

FIGURE 1 – Représentation de l'instance A-n32-k05 de la littérature

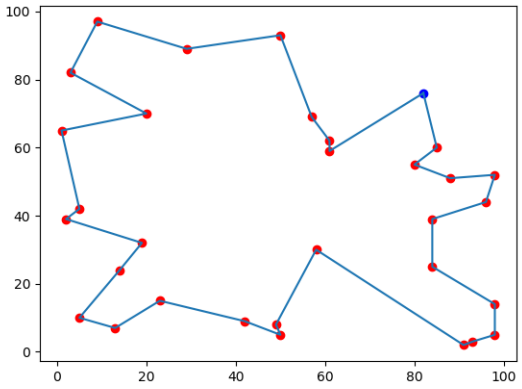

FIGURE 2 – Représentation d'une solution de l'instance A-n32-k05

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 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 d'optimisation, il est souvent intéressant d'explorer les voisinages d'une solution pour voir s'il n'y a pas mieux autour. 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) ;

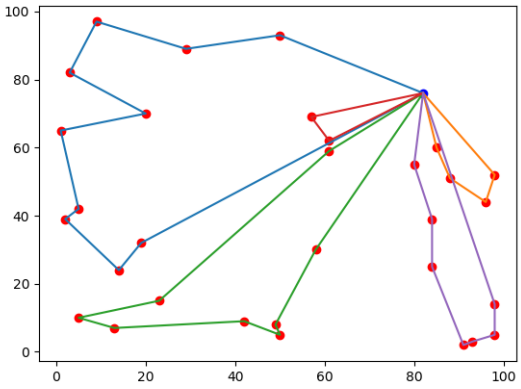

FIGURE 3 – Représentation d'une solution de l'instance A-n32-k05, où les demandes des clients sont prises en compte

— 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 ce papier est d'améliorer les performances d'un algorithme d'optimisation utilisé pour résoudre CVRP, en y intégrant de la connaissance. 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 ? Enfin, 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]. Ainsi, nous commençons par décrire l'algorithme utilisé, puis décrivons le problème du choix des paramètres pour son exécution.

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.

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 (λ, μ, ν)

Le triplet (λ, μ, ν) a déjà été étudié de nombreuses fois dans la littérature. L'article [1] précise qu'il suffit de considérer (λ, μ, ν) 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 (λ, μ, ν) différents. On remarque qu'il n'y a aucun lien entre les résultats et les valeurs de (λ, μ, ν). On ne peut donc pas prévoir à l'avance si le triplet (λ, μ, ν) 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.

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 $i \leftarrow 1$ ***to*** n **do**
    $Sol \leftarrow Sol \cup [0, i, 0]$
Calculer les savings de toutes les arêtes
while $max_{i,j} s(i,j) > 0$ **do**
    $(i,j) \leftarrow argmax_{(i,j)} s(i,j)$
    $r_i \leftarrow findRoute(Sol, i)$
    $r_j \leftarrow findRoute(Sol, j)$
    if r_i *et* r_j *peuvent fusionner* **then**
        Retirer r_i et r_j de Sol
        Si possible fusionner r_i et r_j
        Ajouter le résultat dans Sol et mettre $s(i,j) = 0$
return Sol

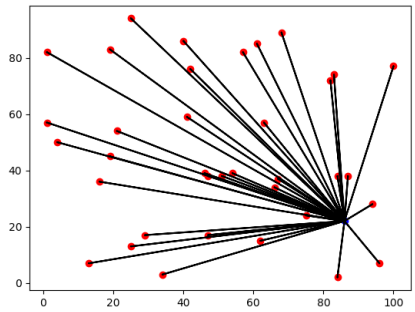

FIGURE 4 – Initialisation

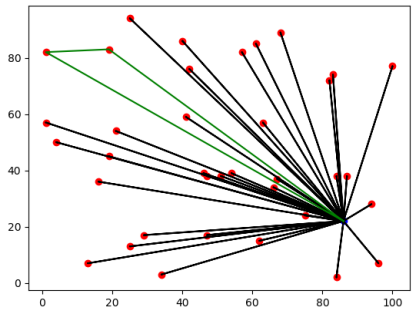

FIGURE 5 – 1^{ere} fusion

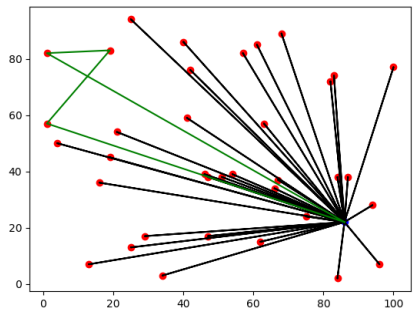

FIGURE 6 – 2^{eme} fusion

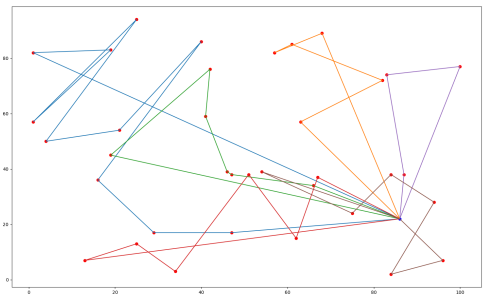

FIGURE 7 – Solution obtenue, $cost = 1297$

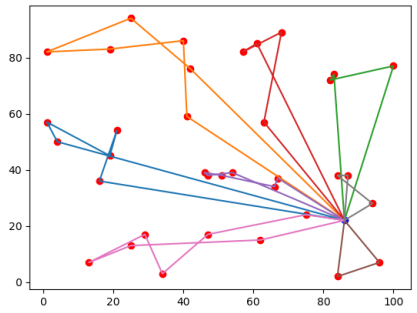

FIGURE 8 – $(1.9, 0.1, 1.5), cost = 1106$

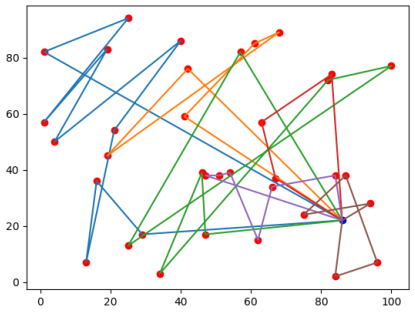

FIGURE 9 – $(0.1, 0.1, 0.1), cost = 1569$

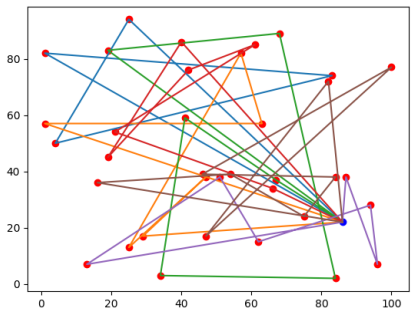

FIGURE 10 – $(0.0, 1.0, 1.5), cost = 2191$

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 d'abord le fonctionnement, avant de décrire davantage les mécanismes mis en jeu lors de son exécution.

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'heuristique 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. Ces opérateurs sont tous en mode $BI - O$, c'est-à-dire que tous les voisins sont parcourus et seul le meilleur est retenu.

L'algorithme 2 donne le fonctionnement de l'heuristique (A&S).

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

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 (λ, μ, ν)

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*

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, 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 :

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

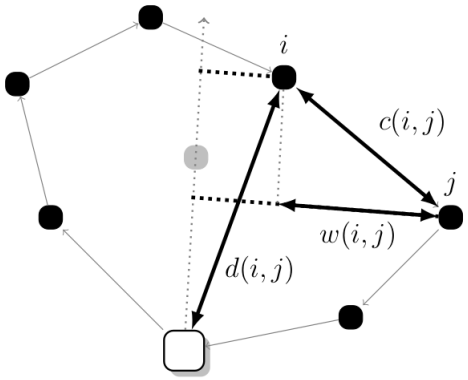

FIGURE 11 – Illustration des caractéristiques d'une arête

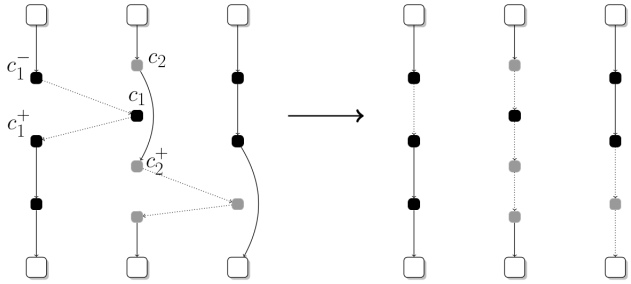

FIGURE 12 – Exemple de fonctionnement de l'opérateur ejection-chain

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 1.2 pour explorer les voisinages.

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*

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

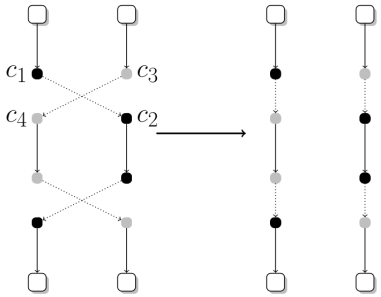

FIGURE 13 – Exemple de fonctionnement de l'opérateur cross-exchange

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 $O(n^2)$.

A la ligne 6 de l'algorithme 4, il est possible d'utiliser les méthodes de la section 1.2 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.

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*

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

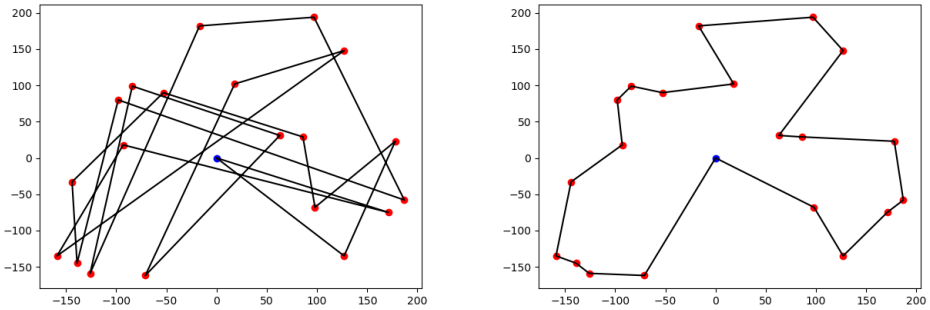

FIGURE 14 – Exemple de fonctionnement de l'opérateur LK

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 $k = 2$.

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 1.2 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, par rapport à l'algorithme 2.

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 *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/2 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*

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. Nous commençons par expliquer quelle va être la connaissance extraite des solutions initiales, puis nous présentons notre protocole d'apprentissage ainsi que les problématiques qu'il soulève. Enfin, nous présentons quelques résultats obtenus, servant à déterminer les paramètres choisis lors de l'apprentissage.

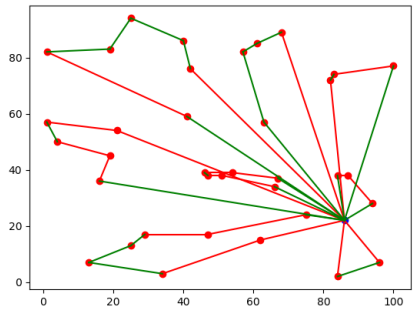

FIGURE 15 – $CW(1.9, 0.1, 1.5) + LK$, $cost = 1041$, 19 arêtes optimales

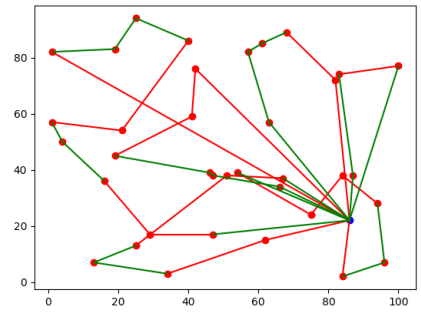

FIGURE 16 – $CW(0.1, 0.1, 0.1) + LK$, $cost = 1170$, 19 arêtes optimales

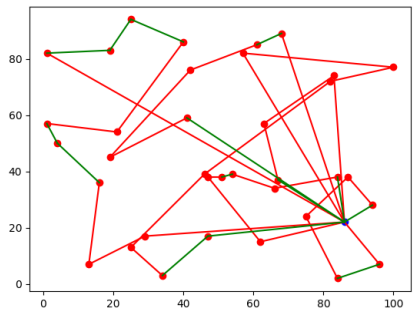

FIGURE 17 – $CW(0.0, 1.0, 1.5) + LK$, $cost = 1600$, 11 arêtes optimales

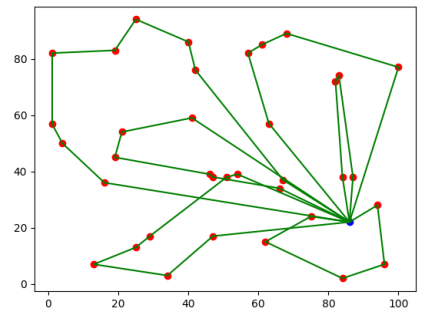

FIGURE 18 – Solution optimale, 42 arêtes

4.1 Quelle est la connaissance ?

En observant quelques solutions obtenues avec *CW*, auxquelles est appliquée 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 18, où les arêtes optimales sont vertes.

Il faudrait donc pouvoir déterminer à l'avance les arêtes optimales à partir des solutions initiales fournies par l'utilisation de *CW* suivi de *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. Afin de vérifier nos résultats, nous comparerons les arêtes obtenues aux arêtes de la solution optimale.

4.2.1 Génération de l'échantillon

Puisque l'on ne peut pas prédire les paramètres (λ, μ, ν) 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 (λ, μ, ν) (ie génération de 8820 solutions), pour obtenir l'échantillon. On l'appelle échantillon *complet* ;

— Soit on tire N triplets (λ, μ, ν) aléatoirement, et les solutions obtenus constitueront notre échantillon.

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

On appelle c_{min} et c_{max} les coûts respectifs de la meilleure et de la pire solution, obtenues dans l'échantillon.

Nous nous intéressons à présent, à la construction d'une base d'apprentissage.

4.2.2 Construction de la base d'apprentissage

La base d'apprentissage, contiendra toutes les solutions utilisées ensuite pour apprendre. Cette base correspond à un sous-ensemble de l'échantillon obtenu. Nous avons retenu trois manières différentes pour obtenir cette base :

— On conserve l'intégralité de l'échantillon. On appelle cette base *Tout* ;

— On conserve x% des meilleures solutions. Dans ce cas, on dit qu'on privilégie la quantité. On appelle cette base $Quan_x$;

— On conserve les solutions qui ont un coût inférieur à $c_{min} + (c_{max} - c_{min})\frac{x}{100}$. Dans ce cas, on dit qu'on privilégie la qualité. On appelle cette base $Qual_x$.

Avec cette base d'apprentissage, nous allons extraire des arêtes qui ont de grandes chances d'être optimales (ie qui appartiennent à la solution optimale).

4.2.3 Extraction des arêtes

Avant tout, nous considérons que les arêtes (i, j) et (j, i) sont identiques (ce qui est le cas en pratique). Chaque arêtes (i, j) a donc pour représentant (i', j'), avec $(i', j') = (i, j)$ si $i < j$, et $(i', j') = (j, i)$ sinon.

Pour savoir si une arête (i, j) est souvent prise dans les solutions de notre base d'apprentissage, nous incrémentons la valeur $Learn[i][j]$ d'une matrice *Learn* de taille n^2 (initialement nulle), lorsque cette arête appartient à la solution considérée. Une fois toutes les solutions parcourues, nous avons retenu deux critères pour extraire les arêtes potentiellement intéressantes :

— Soit nous conservons les *rang* premières arêtes (pour les valeurs de *Learn*). On appelle ce critère *Rang* ;

— Soit nous conservons $(i, j) \Leftrightarrow Learn[i][j] > seuil$. On appelle ce critère *Seuil*.

Afin de déterminer quels doivent être la taille de l'échantillon, la base d'apprentissage, ainsi que le critère utilisé, la partie suivante présente les résultats obtenus.

4.3 Résultats

On choisit de prendre les valeurs suivantes :
— Taille de l'échantillon : $N \in [50, 100, 500]$ et *complet*
— Base d'apprentissage : $Base \in [Tout, Quan_{10}, Qual_{10}]$
— Critères : $rang \in [10, 20, n/2]$ et $seuil \in [|Base|/2, 3|Base|/4]$.
L'apprentissage est réalisé sur trois instances de tailles différentes : *A-n37-k06*, *A-n65-k09* et *P-n101-k04*.

Les résultats obtenus pour chaque instance est présenté dans un tableau (cf tableaux 1 à 3). Ils correspondent aux moyennes obtenus sur 10 échantillons de la taille considérée. La colonne *Arêtes* donne le nombre d'arêtes obtenues. La colonne *Optimales* donne le nombre d'arêtes qui appartiennent effectivement à la solution optimale. Enfin, la colonne *Prop*, donne la proportion d'arêtes optimales parmi toutes les arêtes de la solution optimale.

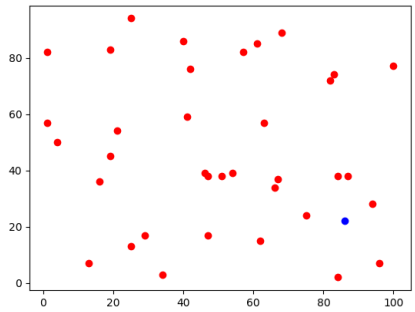

FIGURE 19 – Instance *A-n37-k06*

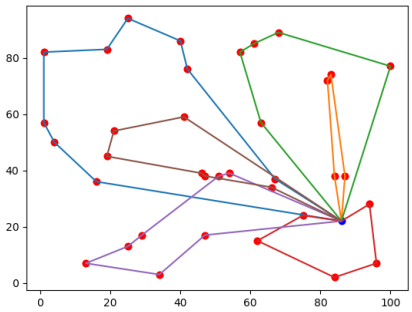

FIGURE 20 – Solution optimale *A-n37-k06*

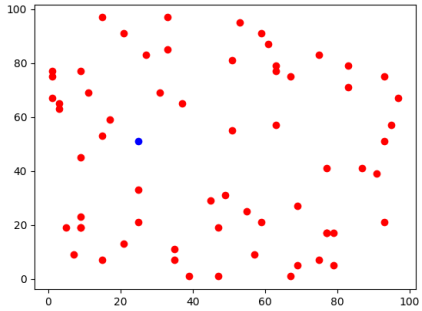

FIGURE 21 – Instance *A-n65-k09*

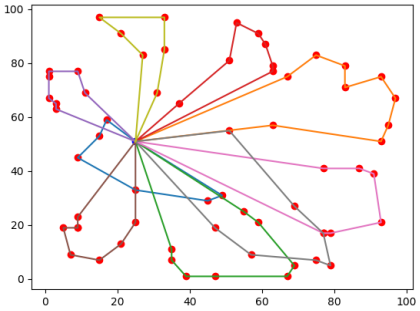

FIGURE 22 – Solution optimale *A-n65-k09*

Voici ce qu'on peut tirer des résultats présents sur les tables 1 à 3 :

— La taille de l'échantillon ne semble pas avoir d'influence sur les résultats (en effet *prop* reste semblable quelle que soit la taille de l'échantillon) ;
— Pour toutes les instances la base *Tout* renvoie de moins bons résultats que les autres bases ;
— La base $Quan_{10}$ est petite pour des petites valeurs d'échantillon ;
— Pour le critère *Rang* les proportions restent similaires quelle que soit la base utilisée. Il faut aussi choisir un *Rang* dépendant de la taille de l'instance pour pouvoir l'adapter à des instances de différentes tailles ;

TABLE 1 – Résultats pour l'instance *A-n37-k06*

Échantillon	$Quan_{10}$				$Qual_{10}$				Tout			
	Seuil	Arêtes	Optimales	Prop	Seuil	Arêtes	Optimales	Prop	Seuil	Arêtes	Optimales	Prop
50	3	34	21	0.5	11	33	21	0.50	25	23	15	0.35
	4	23	14	0.33	17	17	12	0.28	38	10	7	0.16
100	5	30	21	0.5	15	31	23	0.55	50	24	17	0.40
	8	16	15	0.36	23	17	14	0.33	75	6	6	0.14
500	25	32	24	0.57	58	31	22	0.52	250	22	15	0.36
	38	15	14	0.33	88	20	16	0.38	375	7	7	0.18
Complet	400	33	24	0.57	732	30	23	0.55	4000	25	16	0.38
	600	15	14	0.33	1097	18	16	0.38	6000	9	6	0.14
	Rang	Arêtes	Optimales	Prop	Rang	Arêtes	Optimales	Prop	Rang	Arêtes	Optimales	Prop
50	10		6	0.14	10		6	0.14	10		7	0.16
	20		13	0.31	20		13	0.32	20		13	0.31
	18		12	0.28	18		13	0.3	18		12	0.28
100	10		9	0.21	10		9	0.21	10		10	0.24
	20		16	0.38	20		16	0.38	20		15	0.36
	18		13	0.3	18		13	0.3	18		12	0.29
500	10		9	0.21	10		10	0.24	10		9	0.21
	20		16	0.38	20		16	0.38	20		15	0.36
	18		13	0.3	18		13	0.3	18		12	0.28
Complet	10		8	0.19	10		9	0.21	10		7	0.17
	20		14	0.33	20		14	0.33	20		14	0.33
	18		12	0.29	18		12	0.29	18		12	0.29

Ces remarques nous incitent à privilégier un échantillon de taille 50 pour améliorer la vitesse d'apprentissage. La base $Qual_{10}$ semble également être plus intéressante que les autres bases avec une taille d'échantillon 50.

5 Intégration de la connaissance

Cette section s'intéresse à l'intégration de connaissance dans l'algorithme d'optimisation retenu, ie l'algorithme 6. Nous commençons par expliquer où la connaissance va être intégrée, puis nous présenterons la *Learning Heuristic* créée. Enfin nous présenterons les résultats obtenus avec cette heuristique sur des instances de la littérature.

5.1 Où intégrer la connaissance ?

Grâce à l'extraction de connaissances nous disposons désormais d'un ensemble d'arêtes dont la plupart appartiennent à la solution optimale. Nous pouvons donc recréer à partir de ces arêtes des morceaux de tournées et rattacher leurs extrémités aux dépôts, comme le montre la figure 26. Cette solution temporaire, que l'on appellera *Init*, peut alors être utilisée comme initialisation pour l'algorithme CW. En effet, l'algorithme va conserver les morceaux de tour-

TABLE 2 – Résultats pour l'instance *A-n65-k09*

Échantillon	$Quan_{10}$				$Qual_{10}$				Tout			
	Seuil	Arêtes	Optimales	Prop	Seuil	Arêtes	Optimales	Prop	Seuil	Arêtes	Optimales	Prop
50	3	73	43	0.59	10	64	44	0.60	25	40	31	0.43
	4	61	40	0.55	15	39	29	0.40	38	14	9	0.13
100	5	70	44	0.6	22	58	42	0.58	50	43	33	0.45
	8	63	41	0.56	33	36	28	0.39	75	15	10	0.14
500	25	71	43	0.59	111	56	41	0.56	250	45	35	0.48
	38	60	40	0.55	167	35	28	0.39	375	14	9	0.13
Complet	400	62	41	0.56	1005	56	40	0.55	4000	45	35	0.48
	600	15	14	0.33	1508	35	28	0.39	6000	13	9	0.12
	Rang	Arêtes	Optimales	Prop	Rang	Arêtes	Optimales	Prop	Rang	Arêtes	Optimales	Prop
50	10		6	0.08	10		7	0.1	10		7	0.1
	20		14	0.2	20		15	0.21	20		14	0.19
	33		23	0.32	33		26	0.36	33		24	0.33
100	10		6	0.08	10		7	0.1	10		7	0.1
	20		16	0.22	20		16	0.22	20		14	0.19
	33		26	0.36	33		26	0.36	33		25	0.34
500	10		7	0.1	10		7	0.1	10		6	0.08
	20		17	0.23	20		15	0.21	20		13	0.18
	33		27	0.37	33		26	0.36	33		25	0.34
Complet	10		7	0.1	10		7	0.1	10		6	0.08
	20		17	0.23	20		17	0.23	20		13	0.18
	33		27	0.37	33		27	0.37	33		25	0.34

nées, qui ont été fixées lors de l'apprentissage, et va fusionner ces morceaux entre eux, pour obtenir au final une nouvelle solution initiale.

Cette solution initiale pourra ensuite être utilisée dans l'algorithme d'optimisation H_c.

Néanmoins il faut trouver un nouveau triplet de paramètre $(\lambda^*, \mu^*, \nu^*)$ pour pouvoir appliquer CW. Nous décidons pour cela de choisir le triplet associée à la solution de coût le plus faible présente dans l'échantillon, ie vérifiant :

$$(\lambda^*, \mu^*, \nu^*) = argmin_{(\lambda,\mu,\nu)\in Echantillon} CW((\lambda, \mu, \nu))$$

Nous disposons enfin de tous les outils nécessaires à la création de la *Learning Heuristic*.

5.2 Learning Heuristic

La *Learning Heuristic* est présenté sur l'algorithme 7.

L'intérêt de cette heuristique est d'apprendre au départ, mais aussi d'apprendre des solutions trouvées en exécutant H_c. Ces solutions sont stockées dans *newBase*, et servent à déterminer un nouvel $Init$. On réapprend ensuite en se basant sur le modèle d'$Init$ (ie les solutions de l'échantillon sont obtenues avec des CW ayant $Init$ pour initialisation).

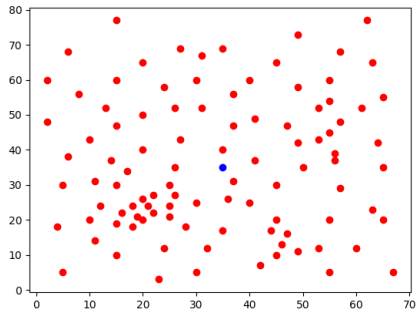

FIGURE 23 – Instance *P-n101-k04*

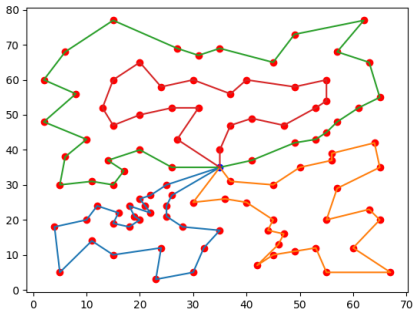

FIGURE 24 – Solution optimale *P-n101-k04*

TABLE 3 – Résultats pour l'instance *P-n101-k04*

Échantillon	$Quan_{10}$				$Qual_{10}$				Tout			
	Seuil	Arêtes	Optimales	Prop	Seuil	Arêtes	Optimales	Prop	Seuil	Arêtes	Optimales	Prop
50	3	93	65	0.62	5	83	66	0.64	25	71	61	0.59
	4	54	44	0.42	8	42	37	0.36	38	24	21	0.20
100	5	80	66	0.64	9	79	66	0.63	50	72	62	0.60
	8	45	41	0.40	14	42	39	0.38	75	24	22	0.21
500	25	83	69	0.67	44	81	68	0.66	250	72	63	0.60
	38	43	39	0.38	67	39	36	0.35	375	22	20	0.19
Complet	400	87	73	0.7	411	85	71	0.68	4000	70	60	0.58
	600	42	39	0.38	616	41	38	0.37	6000	23	21	0.2
	Rang	Arêtes	Optimales	Prop	Rang	Arêtes	Optimales	Prop	Rang	Arêtes	Optimales	Prop
50	10		8	0.08	10		8	0.08	10		8	0.08
	20		18	0.17	20		17	0.16	20		18	0.17
	50		43	0.41	50		44	0.43	50		44	0.43
100	10		8	0.08	10		8	0.08	10		8	0.08
	20		18	0.17	20		18	0.17	20		18	0.17
	50		46	0.44	50		46	0.44	50		46	0.44
500	10		8	0.08	10		8	0.08	10		8	0.08
	20		18	0.17	20		18	0.17	20		18	0.17
	50		46	0.44	50		46	0.44	50		46	0.44
Complet	10		8	0.08	10		8	0.08	10		8	0.08
	20		18	0.17	20		18	0.17	20		18	0.17
	50		46	0.44	50		46	0.44	50		46	0.44

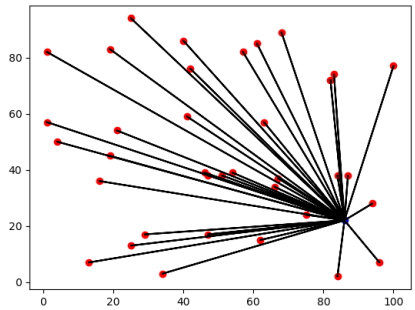

FIGURE 25 – Initialisation habituelle de CW

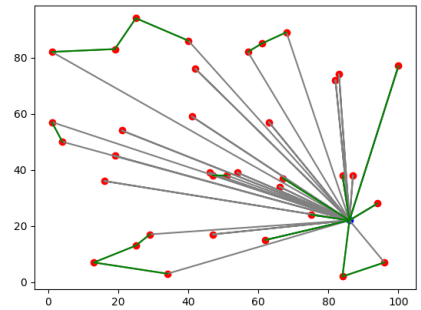

FIGURE 26 – Initialisation de CW après apprentissage : *Init*

5.3 Résultats

Conclusion

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.

Algorithm 7: LEARNHEURISTIC renvoie une solution d'une instance du CVRP

Input: Un ensemble de points I, les demandes des clients D
Output: Une solution au problème I

```
(λ*, μ*, ν*), Init ← Apprentissage()
newBase ← []
for i ← 1 to 10 do
    if i = 1 then
        for j ← 1 to 10 do
            Sol ← H_c(Init, I, D, λ*, μ*, ν*)
            newBase ← newBase ∪ Sol
    else
        Déterminer Init avec les connaissances de newBase
        newBase ← []
        (λ*, μ*, ν*), Init ← Apprentissage(Init)
        for j ← 1 to 10 do
            Sol ← H_c(Init, I, D, λ*, μ*, ν*)
            newBase ← newBase ∪ Sol
return La meilleure solution
```